Sure! Here are 20 tasks/exercises designed to help you deepen your understanding of datatypes in JavaScript (including primitives, objects, and Symbols):

**Basic Datatypes**

1. **Primitive Types Practice**: Create variables of each primitive type (string, number, boolean, null, undefined, bigint, and symbol) and log their types using typeof.
2. **Type Conversion**: Write a function that converts a given value to different types (e.g., string to number, boolean to string) and logs the results.
3. **Checking Equality**: Write examples demonstrating the difference between == and === with different types (e.g., 1 == '1' vs. 1 === '1').
4. **Null vs. Undefined**: Create a function that takes a parameter and logs whether it is null or undefined. Call this function with different values.
5. **Arithmetic Operations**: Perform arithmetic operations with different types (e.g., number + string, boolean + number) and explain the results.

**Objects and Arrays**

1. **Object Creation and Manipulation**: Create an object with various properties (including nested objects). Add, update, and delete properties, and iterate over the object’s keys and values.
2. **Array Methods**: Create an array of numbers and perform various operations using array methods (push, pop, shift, unshift, splice, map, filter, reduce).
3. **Cloning and Merging Objects**: Demonstrate shallow vs. deep cloning of objects using the spread operator and a deep cloning function (e.g., JSON.parse(JSON.stringify(obj))).
4. **Handling Dates**: Create a Date object and perform operations like formatting the date, calculating the difference between dates, and manipulating dates.
5. **Set and Map**: Create a Set and a Map, perform operations (add, delete, check existence), and iterate over their elements.

**Advanced Topics**

1. **Symbol Usage**: Create a unique Symbol and use it as a property key in an object. Demonstrate how it prevents property name conflicts.
2. **Well-Known Symbols**: Use well-known symbols (e.g., Symbol.iterator, Symbol.toStringTag) to customize object behavior.
3. **Type Assertions**: Write a function that accepts any type and uses type assertions to safely perform operations based on the type of input.
4. **Custom Iterables**: Create a custom iterable object using the Symbol.iterator method and iterate over it with a for...of loop.
5. **Prototype and Inheritance**: Create a constructor function, define methods on its prototype, and demonstrate inheritance by creating a subclass.

**Type Checking and Validation**

1. **Type Guards**: Write functions that use type guards to check the type of input parameters and handle them accordingly.
2. **TypeScript Basics**: (Optional if using TypeScript) Create a simple TypeScript project and define different types, interfaces, and type aliases. Compile it to JavaScript.
3. **Type Coercion**: Write examples demonstrating implicit type coercion in JavaScript (e.g., == vs. ===, + operator with different types) and explain the results.
4. **Custom Type Conversion**: Implement custom toString and valueOf methods in an object to control its string and numeric conversion.
5. **Type Checking Libraries**: Explore a type checking library like prop-types (for React) or TypeScript for stricter type enforcement in a project.

**Practical Application**

1. **JSON Handling**: Write functions to convert objects to JSON strings and parse JSON strings back to objects. Handle errors gracefully.
2. **Function Overloading**: (If using TypeScript) Demonstrate function overloading to handle multiple types of input with different behaviors.
3. **Proxy Objects**: Create a Proxy to intercept and log operations on an object, such as property access and assignment.
4. **Immutable Data Structures**: Use a library like Immutable.js to create immutable data structures and demonstrate their usage.
5. **Data Validation**: Write a function that validates an object's properties based on expected types (e.g., using a schema validation library like Joi).

**Debugging and Testing**

1. **Debugging Types**: Use console.log and typeof statements to debug type-related issues in a piece of code.
2. **Unit Tests**: Write unit tests for functions that involve type conversions and validations using a testing framework like Jest.
3. **Error Handling**: Write a function that performs various type operations and includes comprehensive error handling for invalid inputs.
4. **Type Performance**: Measure the performance of different datatype operations (e.g., object property access vs. array index access) using the console.time method.
5. **Real-World Project**: Apply your knowledge of datatypes in a small real-world project, such as a todo list application, where you handle various data inputs and ensure type safety throughout the application.

By completing these exercises, you'll gain a solid understanding of JavaScript datatypes and how to effectively use them in real-world scenarios.

JavaScript provides a variety of methods for manipulating and working with strings. Here is a comprehensive list of string methods available in JavaScript:

**Basic Methods**

1. **charAt(index)**: Returns the character at the specified index.

let str = "Hello";

console.log(str.charAt(1)); // "e"

1. **charCodeAt(index)**: Returns the Unicode of the character at the specified index

console.log(str.charCodeAt(1)); // 101

1. **concat(str1, str2, ..., strN)**: Joins two or more strings and returns a new concatenated string.

console.log(str.concat(" ", "World")); // "Hello World"

1. **includes(substring, start)**: Checks if the string contains the specified substring, starting from a given position.

console.log(str.includes("ell")); // true

1. **endsWith(substring, length)**: Checks if the string ends with the specified substring.

console.log(str.endsWith("lo")); // true

1. **indexOf(substring, start)**: Returns the index of the first occurrence of the specified substring, starting from a given position.

console.log(str.indexOf("l")); // 2

1. **lastIndexOf(substring, start)**: Returns the index of the last occurrence of the specified substring, starting from a given position.

console.log(str.lastIndexOf("l")); // 3

1. **localeCompare(compareString, locales, options)**: Compares two strings in the current locale.

console.log(str.localeCompare("Hello")); // 0 (equal)

**Case Conversion**

1. **toLowerCase()**: Converts the entire string to lowercase.

console.log(str.toLowerCase()); // "hello"

1. **toUpperCase()**: Converts the entire string to uppercase.

console.log(str.toUpperCase()); // "HELLO"

**Trimming and Padding**

1. **trim()**: Removes whitespace from both ends of the string.

let strWithSpaces = " Hello ";

console.log(strWithSpaces.trim()); // "Hello"

1. **trimStart() / trimLeft()**: Removes whitespace from the beginning of the string.

console.log(strWithSpaces.trimStart()); // "Hello "

1. **trimEnd() / trimRight()**: Removes whitespace from the end of the string.

console.log(strWithSpaces.trimEnd()); // " Hello"

1. **padStart(targetLength, padString)**: Pads the string with another string (repeated, if needed) so that the resulting string reaches a given length.

console.log(str.padStart(10, '\*')); // "\*\*\*\*\*Hello"

1. **padEnd(targetLength, padString)**: Pads the string with another string (repeated, if needed) so that the resulting string reaches a given length.

console.log(str.padEnd(10, '\*')); // "Hello\*\*\*\*\*"

**Extracting Substrings**

1. **slice(start, end)**: Extracts a section of the string and returns it as a new string.

console.log(str.slice(1, 4)); // "ell"

1. **substring(start, end)**: Returns the part of the string between the start and end indexes.

console.log(str.substring(1, 4)); // "ell"

1. **substr(start, length)**: Returns the part of the string from the start index with the specified length (deprecated but still used in some codebases).

console.log(str.substr(1, 3)); // "ell"

**Splitting and Replacing**

1. **split(separator, limit)**: Splits the string into an array of substrings based on the specified separator.

let csv = "a,b,c,d";

console.log(csv.split(",")); // ["a", "b", "c", "d"]

1. **replace(searchValue, newValue)**: Replaces the first occurrence of a substring or a regex match with a new string.

console.log(str.replace("l", "x")); // "Hexlo"

1. **replaceAll(searchValue, newValue)**: Replaces all occurrences of a substring or a regex match with a new string.

javascript

Copy code

console.log(str.replaceAll("l", "x")); // "Hexxo"

**Template Literals**

1. **Template Literals**: Uses backticks (`) to allow for easier string interpolation and multi-line strings.

let name = "World";

console.log(`Hello, ${name}!`); // "Hello, World!"

**Searching**

1. **match(regex)**: Retrieves the matches of a string against a regular expression.

let sentence = "The quick brown fox jumps over the lazy dog";

console.log(sentence.match(/quick/)); // ["quick"]

1. **matchAll(regex)**: Returns an iterator of all matched substrings against a regular expression.

let matches = sentence.matchAll(/o/g);

for (const match of matches) {

console.log(match); // logs each match

}

1. **search(regex)**: Searches for a match between a regular expression and the string, and returns the index of the match.

console.log(sentence.search(/fox/)); // 16

1. **startsWith(substring, start)**: Checks if the string starts with the specified substring.

console.log(sentence.startsWith("The")); // true

**Miscellaneous**

1. **fromCharCode(...codes)**: Static method that returns a string created from the specified sequence of Unicode values.

console.log(String.fromCharCode(72, 101, 108, 108, 111)); // "Hello"

1. **fromCodePoint(...codePoints)**: Static method that returns a string created from the specified sequence of code points.

console.log(String.fromCodePoint(128512)); // "😀"

1. **codePointAt(pos)**: Returns a non-negative integer that is the Unicode code point value at the given position.

console.log("😀".codePointAt(0)); // 128512

1. **repeat(count)**: Returns a new string with a specified number of copies of the original string.

console.log("abc".repeat(3)); // "abcabcabc"