PwSkill Assignment

1. compare var, let and const in javascript. when would you use each?

In JavaScript, `var`, `let`, and `const` are used to declare variables, but they have different behaviors and use cases. Here's a detailed comparison:

- The var is the oldest keyword to declare a variable in Java Script. It has the Global scoped or function scoped which means variables defined outside the function can be accessed globally, and variables defined inside a particular function can be accessed within the function.

Example :

```
var a = 10
function f() {
    var b = 20
    console.log(a, b)
}
f();
console.log(a);
```

Output :

```
10 20
10
```

- The let keyword is an improved version of the var keyword. It is introduced in the ES6 or EcmaScript 2015. These variables has the block scope. It can't be accessible outside the particular code block.

Example :

```
let a = 10;
function f() {
    let b = 9
    console.log(b);
    console.log(a);
}
f();
```

output :

```
9

10
```

- The const keyword has all the properties that are the same as the let keyword, except the user cannot update it and have to assign it with a value at the time of declaration. These variables also have the block scope. It is mainly used to create constant variables whose values can not be changed once they are initialized with a value.

Example :

```
const a = 10;
function f() {
    a = 9
    console.log(a)
}
f();
```

Output :

```
TypeError:Assignment to constant variable.
```

---

2. what is hoisting in JavaScript? how does it affect on var let and const?

Hoisting is a JavaScript mechanism where variable and function declarations are moved to the top of their containing scope during the compile phase before the code execution. This means that you can use functions and variables before they are declared in the code. However, the way hoisting works differs between `var`, `let`, and `const`.

Hoisting of `var`

Variables declared with `var` are hoisted to the top of their containing function or global scope, but their initialization remains in place. As a result, the variable is undefined until the code execution reaches the initialization.

Example :

```
console.log(x); // undefined, because the declaration is hoisted var x = 5;
console.log(x); // 5

The JavaScript engine interprets this code as:
javascript var x;
console.log(x); // undefined
x = 5;
console.log(x); // 5
```

Hoisting of `let`

Variables declared with `let` are hoisted to the top of their block scope, but they are not initialized. This creates a "temporal dead zone" (TDZ) from the start of the block until the declaration is encountered. Accessing the variable in the TDZ results in a `ReferenceError`.

Example :

```
console.log(y); // ReferenceError: Cannot access 'y' before initialization let
y = 10;
console.log(y); // 10

The JavaScript engine interprets this code as:
javascript // TDZ starts
console.log(y); // ReferenceError let y = 10; // TDZ ends console.log(y); //
10
```

Hoisting of `const`

Variables declared with `const` are also hoisted to the top of their block scope, but like `let`, they are not initialized until their declaration is reached. This also creates a temporal dead zone (TDZ). Additionally, `const` variables must be initialized at the time of declaration.

Example :

```
console.log(z); // ReferenceError: Cannot access 'z' before initialization
const z = 20;
console.log(z); // 20

The JavaScript engine interprets this code as:
javascript // TDZ starts
console.log(z); // ReferenceError const z = 20; // TDZ ends
console.log(z); // 20
```

Summary of Hoisting Behavior

1. `var`:
Declarations are hoisted to the top of their scope.
Initialization remains in place.
Can be accessed before declaration, but will be `undefined`.

2. `let`:
Declarations are hoisted to the top of their block scope.
Initialization remains in place, and accessing the variable before declaration causes a `ReferenceError` due to the temporal dead zone.

3. `const`:
Declarations are hoisted to the top of their block scope.
Initialization remains in place, and accessing the variable before declaration causes a `ReferenceError` due to the temporal dead zone.
Must be initialized at the time of declaration.

3. Explain JavaScript's dynamic typing and the typeof operator with examples.

Javascript uses dynamic typing, meaning that variables are not explicitly declared with specific data types. This means that JavaScript assumes what the data type is based on what is assigned to the variable.

```
1.  Variable Reassignment:
   javascript
   let myVar = 42;         // Initially a number
   console.log(typeof myVar); // "number"

   myVar = "Hello, world!"; // Now a string
   console.log(typeof myVar); // "string"

   myVar = true;          // Now a boolean
   console.log(typeof myVar); // "boolean"
```

```
2.  Function Parameters:    javascript    function add(a, b) {
    return a + b;
```

```
}

console.log(add(5, 10));         // 15 (number)
console.log(add("Hello, ", "world!")); // "Hello, world!"
(string)    console.log(add(5, "10"));       // "510" (string, due to type
coercion)
```

The typeof operator in JavaScript is used to check the type of the operand passed to it. It will return the type of the passed operand that can be used to compare the types of the two different variables declared in the code. The typeof operator is used by specifying it just before the name of the operand to get its type. It returns the type of the operand in the form of a string. So, if you are trying to compare the type manually by specifying it, you should use the string format to specify it.

```
// Define variables with different primitive data types
const num = 10;
const str = "Hello";
const bool = true;
const undef = undefined;
const nul = null;
const sym = Symbol("symbol");
const bigInt = 9007199254740991n;

// Use typeof operator to determine the data type
console.log(typeof num);    // Output: "number"
console.log(typeof str);    // Output: "string"
console.log(typeof bool);   // Output: "boolean"
console.log(typeof undef); // Output: "undefined"
console.log(typeof nul);    // Output: "object" (typeof null is an oddity,
                            // it returns "object")
console.log(typeof sym);    // Output: "symbol"
console.log(typeof bigInt);// Output: "bigint"
```

output :

```
number
string
boolean
undefined
```

```
object
symbol
bigint
```

4. How can you manipulate strings in JavaScript? Provide examples of common methods.

Manipulating strings in JavaScript is a common task, and there are numerous methods available to help you handle strings efficiently. Here are some of the most commonly used string manipulation methods, along with examples:

Length Property

Returns the length of a string.

javascript
let str = "Hello, World!";
console.log(str.length); // Output: 13

Accessing Characters

You can access individual characters using bracket notation or the `charAt` method.

javascript
console.log(str[0]); // Output: H
console.log(str.charAt(0)); // Output: H

indexOf() and lastIndexOf()

Find the index of a substring within a string.

javascript
console.log(str.indexOf('o')); // Output: 4
console.log(str.lastIndexOf('o')); // Output: 8

slice()

Extracts a part of a string and returns it as a new string.

javascript
console.log(str.slice(7, 12)); // Output: World

substring()

Similar to `slice()`, but does not accept negative indices.

javascript
console.log(str.substring(7, 12)); // Output: World

substr()

Extracts a substring from a string, starting at a specified index and extending for a given number of characters.

javascript
console.log(str.substr(7, 5)); // Output: World

toUpperCase() and toLowerCase() Convert a string to upper or lower case.

javascript console.log(str.toUpperCase()); // Output: HELLO, WORLD!
console.log(str.toLowerCase()); // Output: hello, world!

replace()

Find a substring and replace it with another substring.

javascript console.log(str.replace('World', 'JavaScript')); // Output: Hello, JavaScript!

trim()

Remove whitespace from both ends of a string.

javascript
let strWithWhitespace = "   Hello, World!   ";
console.log(strWithWhitespace.trim()); // Output: "Hello, World!"

split()

Split a string into an array of substrings.

javascript
let strArray = str.split(', ');
console.log(strArray); // Output: ["Hello", "World!"]

concat()

Concatenate two or more strings.

javascript let str1 = "Hello"; let str2 = "World"; console.log(str1.concat(", ", str2, "!")); // Output: Hello, World!

includes()

Check if a string contains a certain substring.

javascript
console.log(str.includes("World")); // Output: true

startsWith() and endsWith()

Check if a string starts or ends with a certain substring.

javascript
console.log(str.startsWith("Hello")); // Output: true
console.log(str.endsWith("!")); // Output: true

repeat()

Repeat a string a specified number of times.

javascript
let repeatedStr = "abc".repeat(3);
console.log(repeatedStr); // Output: abcabcabc

These methods cover a wide range of string manipulation tasks you might need in JavaScript. Understanding and using these methods will help you effectively manage and transform strings in your JavaScript applications.

5. What are the differences and similarities between null and undefined in JavaScript?

In JavaScript, `null` and `undefined` are both used to represent the absence of a value, but they have different meanings and uses. Here's a detailed comparison:

Similarities

1. Type:
- Both `null` and `undefined` are primitive values.

2. Falsy Values:
- Both are considered falsy values in a boolean context, meaning they evaluate to `false` when used in a conditional statement.

javascript
if (!null) console.log("null is falsy"); // Output: null is falsy
if (!undefined) console.log("undefined is falsy"); // Output: undefined is falsy

Differences

1. Meaning:

- `undefined`: Indicates that a variable has been declared but not assigned a value, or a function has been called but has not returned a value.
- `null`: Explicitly represents the intentional absence of any object value. It is often used to reset or clear a variable.

```
javascript
let a;
console.log(a); // Output: undefined

let b = null;
console.log(b); // Output: null
```

2. Type:
- `undefined` is its own type.

```
javascript
console.log(typeof undefined); // Output: undefined
```

- `null` is of type `object`. This is a long-standing bug in JavaScript, and it was never fixed for backward compatibility reasons.

```
javascript
console.log(typeof null); // Output: object
```

3. Default Values:
- Variables that are declared but not initialized are `undefined`.

```
javascript
let x;
console.log(x); // Output: undefined
```

- Function parameters that are not provided default to `undefined`.

```
javascript
function
foo(arg) {
    console.log(arg);
  }
  foo(); // Output: undefined
```

4. Usage Contexts:
- `undefined` is used by JavaScript itself, e.g., for variables that are declared but not assigned, or when accessing object properties that do not exist.

```
javascript
let obj = {};
console.log(obj.property); // Output: undefined
```

- `null` is used by programmers to indicate that a variable should have no value.

```
javascript
let data = null; // No value assigned on purpose
```

5. Equality Comparison:
- `undefined` and `null` are loosely equal (`==`), but not strictly equal (`===`).

```
javascript
console.log(null == undefined); // Output: true
console.log(null === undefined); // Output: false
```

Practical Examples

- Default Parameter Values:

```
   javascript
function greet(name){
if (name ===
undefined){
      console.log("Hello, stranger!");
    } else {
      console.log(`Hello, ${name}!`);
    }
  }
  greet(); // Output: Hello, stranger!
greet("Alice"); // Output: Hello, Alice!
```

- Resetting Variables:

```
   javascript
   let person = { name: "John" };
   person = null; // person is now explicitly set to have no value
```

Understanding the differences and appropriate use cases for `null` and `undefined` is important for writing clear and bug-free JavaScript code.