5. Ternary vs. If-Else

Ternary Operator:

- **Syntax:** condition ? expressionIfTrue : expressionIfFalse
- **Usage:** Assigns a value based on a condition.
- **Benefits:** Concise for simple conditions and assigning values.
- **Drawbacks:** Can become hard to read with complex logic or multiple assignments.

Example:

JavaScript

```
const isAdult = age >= 18;
const message = isAdult ? "You are an adult." : "You are not an adult.";
console.log(message); // Output: "You are an adult." (assuming age >= 18)
```

If-Else Statement:

- **Syntax:**
  JavaScript
  ```
  if (condition) {
    // code to execute if condition is true
  } else {
    // code to execute if condition is false (optional)
  }
  ```
- **Usage:** Controls the flow of execution based on conditions and can execute multiple statements within each block.
- **Benefits:** More readable for complex logic and multiple statements.
- **Drawbacks:** Less concise than a ternary operator for simple assignments.

Choosing Between Them:

- For simple conditions and assigning a single value, the ternary operator can be a good choice.
- For complex logic, multiple statements, or better readability, an if-else statement is often preferred.

6. Type Conversion in JavaScript

Type Coercion (Implicit Conversion): JavaScript automatically converts values from one type to another in certain situations. Here are common examples:

- Numbers to strings (e.g., 10 + "hello" becomes "10hello")
- Strings to numbers (e.g., "10" * 2 becomes 20)
- Booleans to numbers (e.g., true + false becomes 1)

Explicit Conversion: You can use functions like parseInt(), parseFloat(), toString(), and others to explicitly convert between data types.

Example:

JavaScript

```
const numString = "123";
const num = parseInt(numString); // num becomes 123 (number)
```

7. Scope in JavaScript

Scope determines the accessibility of variables and functions within your code. There are two main types of scopes in JavaScript:

a. Global Scope:

- Variables and functions declared outside of any function or block have global scope.
- They are accessible from anywhere in your code, which can lead to naming conflicts and make code harder to maintain.

Example:

JavaScript

```
let globalVar = "This is global!"; // Accessible from anywhere

function sayHello() {
  console.log(globalVar); // Can access globalVar
}

sayHello(); // Output: "This is global!"
```

b. Local Scope:

- Variables and functions declared inside a function or block have local scope.
- They are only accessible within that function or block.

Example:

JavaScript

```
function sayGoodbye() {
  let localVar = "Goodbye from inside the function!";
  console.log(localVar); // Accessible here
}

sayGoodbye(); // Output: "Goodbye from inside the function!"
  // console.log(localVar); // Uncommenting this will cause an error
(localVar not defined)
```

Block Scope (ES6 and later):

- Introduced in ES6 (ECMAScript 2015), let and const declarations create block scope.
- Variables declared with let or const within a block (e.g., if statement, for loop) are only accessible within that block.

Example:

JavaScript

```
if (true) {
  let blockVar = "This is only accessible within the if block!";
}

console.log(blockVar); // Uncommenting this will cause an error
(blockVar not defined)
```

Function Scope (var):

- Variables declared with var (prior to ES6) inside a function have function scope.
- They are accessible from anywhere within the function, even within nested blocks. However, their use is discouraged due to potential issues with hoisting (variables accessible before their declaration).

Key Points:

- Use let and const for variable declarations to create clear and predictable scope.
- Avoid using global variables except for very specific scenarios.
- Be mindful of block scope when using let and const within loops or conditional statements.