EXPERT VIEWS

Master of Javascript Errors

Resolve Mistakes Faster Than ChatGPT

Dragoslav Ivkovic

First Edition

Contents

- Explanation
- Examples with Solutions
- Ways to Avoid Mistakes

Behavior of null

- Explanation
- Examples with Solutions
- Ways to Avoid Mistakes

Strict comparison of objects

- Explanation
- Examples with Solutions
- Ways to Avoid Mistakes

Implicit Type Conversion in JavaScript

- Understanding
- Ways to Avoid These Errors
- Example & Solution

Floating Point Precision Error in JavaScript

- Understanding
- Examples & Solution

Strict comparison within Switch() statements

- Explanation
- Examples with Solutions
- Ways to Avoid Mistakes

The specificity of the replace() method

- Explanation
- Examples with Solution
- How to Avoid These Mistakes

Array constructor

- Explanation
- Examples with Solutions

- How to Avoid This Mistake

JavaScript Date object

- Explanation
- Examples with Solutions
- How to Avoid This Mistake

This

- Explanation
- Examples with Solutions
- How to Avoid This Mistake

Shadowing a variable

- Explanation
- Examples with Solutions
- How to Avoid This Mistake

Function arguments

Array.prototype.sort()

Problem with callback function in a loop

Problem with callback function in a loop - Unexpected behavior of a callback function in a loop can be attributed to the fact that it does not get invoked immediately while the loop is running, but is always invoked with a "delay" when the loop has already completed and reached the last

Master of Javascript Errors: Resolve Mistakes Faster Than ChatGPT First Edition Your pathway to swiftly and efficiently resolving JavaScript errors Author: Dragoslav Ivkovic

Dragoslav Ivkovic GLOBAL

First Published: July 2023

Production reference: 1210421

Published by Dragoslav Ivkovic Publishing Ltd. Global

ISBN: 9798853891531

www.dragoslavivkovic.com

About the Author

Dragoslav Ivkovic is an experienced web developer who has dedicated over six years to mastering JavaScript and its associated frameworks, including React and Next.js. His expertise has been employed across an array of web applications, ranging from small interactive websites to large-scale enterprise solutions.

When Dragoslav is not diving deep into code, he enjoys exploring the latest tech trends, unwinding with video games, or appreciating nature's beauty during a hike. He is also an active contributor to the developer community and never misses an opportunity to share his knowledge and insights.

With "Master of Javascript Errors", Dragoslav aims to share his passion for coding and help others navigate the often intimidating landscape of JavaScript errors. He believes that with the right guidance, these errors can become valuable learning opportunities rather than roadblocks. So get ready to dive deep into the world of JavaScript, conquer those persistent errors, and code like never before!

Master of Javascript Errors: Resolve Mistakes Faster Than ChatGPT

Ever been faced with such a JavaScript error that made you scratch your head? Or does some part of your code lead you into the labyrinths where there are no results to solve them? Then, this is definitely the right place.

'Master of Javascript Errors' isn't just another programming manual. Imagine it as a close friend who helps you navigate the tricky terrain of JavaScript errors designed to turn you into a swift and efficient problem-solver.

Whether you're a newbie trying to get the hang of JavaScript or a pro looking for an excuse to deal with code glitches more effectively, this book is your trusty sidekick. With each page, you'll dig deeper into JavaScript errors—a knowledge that will help you conquer them head-on.

This guide is bursting with easy-to-understand explanations, hands-on instructions, and real-life

examples making sure every problem turns into a valuable learning experience. Learn to find and fix both basic and complex errors – learn how to sidestep them as well – and come up with practical solutions. Our book is crisp, direct, doesn't bore you with unnecessary detaling...concretely offers you solutions—helps you recognize error types quickly directing towards potential fixes.

'Master of Javascript Errors' teaches you the knowledge to face JavaScript errors confidently, knowing you won't be just good at debugging—you'll outrun most AIs when it comes to speed and accuracy.

'Master of Javascript Errors' will serve as a jumpboard on your journey towards mastering code errors. Discover how you can fix, enhance and speed up your JavaScript code like never before. Soon, JavaScript errors won't be roadblocks but stepping stones towards better coding skills.

What Are JavaScript Errors?

When a program encounters a situation it doesn't know how to handle, it results in an error. This could happen in different scenarios, like when the program tries to open a file that doesn't exist, or when it attempts to connect to a website, but there's no Internet connection.

These situations cause the program to give an error message to the user. It's like the program saying, "I'm sorry, but I can't do what you're asking me to do." The program tries to give as much information as possible about what went wrong. Clever programmers try to guess these situations in advance. They do this so that the user gets a message that's easy to understand, instead of a confusing code like "404". Instead, the user might see a friendly message like "The page could not be found."

In JavaScript, errors are presented as objects when there's a mistake in the code. These objects have lots of information about the error, like what kind

of error it is, which part of the code caused it, and a track of the code's operation when the error happened. JavaScript also lets programmers make their own error messages. This can be really helpful when they're trying to figure out what's wrong.

Understanding JavaScript Error Characteristics

After understanding what a JavaScript error is, let's look closer at what these errors are made of.

JavaScript errors have specific built-in and customizable traits that help figure out why the error happened and what it affects. Usually, JavaScript errors have three main traits:

"message": This is a text message that explains the error.

"name": This tells us the kind of error that happened. (We'll learn more about this later.)

"stack": This shows the path of the code that was running when the error happened.

Besides these, errors can have other traits like "columnNumber", "lineNumber", "fileName", and so on. These give more information about the error. However, these extra traits are not always there and might not be included in every error object created by your JavaScript app.

Decoding the Error Trail

A "stack trace" is like a breadcrumb trail showing the steps a program took when an error happened. Here's what it typically looks like:

First, it shows the error type and message. After that, it lists the steps the program took, one by one. Each step shows where in the source code it happened and the line number. You can use this information to find the exact spot in your code where the error occurred.

The steps are listed in a stacked way. It starts from where the error happened and goes back through the steps the program took. If you "catch" the error, it won't go back through all the steps and stop your program. But sometimes, you might want to let serious errors go uncaught to stop the program on purpose.

Mistakes vs Unexpected Events

Sometimes, people think that errors and exceptions are the same thing. But there's a small but important difference between them.

An "exception" is an error that has been thrown.

Here's an example to make it clearer. This is how you can make an error in JavaScript:

```
const wrongTypeError = TypeError("Wrong type found, expected character")
```

And this is how you can turn that error into an exception:

```
throw wrongTypeError
```

Many people like to use a shortcut that makes an error and throws it at the same time:

```
throw TypeError("Wrong type found, expected character")
```

This is a common way to do it. **But because it's so common, sometimes people get mixed up between exceptions and errors.** So, it's important to understand the difference, even if you use the shortcut.

Understanding JavaScript and Its Errors

Hello, fellow code explorer! If you're here, you're likely standing face to face with the wild world of JavaScript errors. They can seem like unclimbable mountains or bottomless pits, often appearing at the least opportune times, painting your console with intimidating red text. But here's a comforting truth: it's part of our journey. As JavaScript developers, we all stumble into these errors, whether you're fresh out of boot-camp or a seasoned code warrior.

But fear not! This book will transform these confusing, daunting errors into something understandable and manageable. We'll dissect each one, peeling back the layers of jargon and complexity to reveal their true nature. You'll learn to look beyond the initial shock and view them as constructive dialogs between you and your code.

Overview of Error Types in JavaScript

So, let's talk about the various 'creatures' inhabiting the JavaScript error 'zoo'. From **SyntaxError** and **ReferenceError**, to **TypeError** and **RangeError**, not forgetting our friends **EvalError**, **URIError**, **InternalError**, and **AggregateError**. Each has its own habitat and behavior, presenting unique challenges to the unwary programmer.

However, don't be overwhelmed. You're not stepping into this jungle alone. In the chapters ahead, we'll embark on an expedition, meeting each of these error types up close. We'll decipher their languages, understand their behaviors, and discover effective strategies to deal with them. By the end of this journey, you'll find these wild creatures more like familiar companions, aiding rather than hindering your coding progress.

Importance of Proper Error Handling

Before we set off, let's shine a spotlight on why proper error handling matters. Errors aren't just hurdles in our path; they're stepping stones guiding us towards better, more resilient code. Dismissing them or haphazardly silencing them with quick fixes only covers up the underlying issues. It's akin to applying a plaster to a deep wound—it might hide the injury, but it doesn't foster healing.

Proper error handling lets us address the root causes, fortifying our code against future errors of the same nature. In doing so, we not only 'clean' our code but also make it robust and bulletproof, something we can confidently stand behind.

With this book, you're not just learning to solve problems—you're enhancing your skills as a JavaScript developer, enabling you to tackle tasks more efficiently and effectively. You'll discover how to turn roadblocks into expressways, transforming your ap-

proach towards JavaScript errors, and most importantly, growing in your coding journey.

So, are you ready to dive in, embrace the challenges, and emerge as a m

Understanding SyntaxError

At its core, JavaScript is a language, and like any other language, it has a set of rules governing its structure - this is known as its syntax. When these rules are broken, you might encounter a SyntaxError.
A SyntaxError in JavaScript is thrown when the JavaScript interpreter comes across tokens or a token order in your code that does not conform to the syntax of the language. This usually happens when there's something that's either missing or misplaced in your code.

Common Causes of SyntaxError

Some common reasons for a **SyntaxError** to be thrown include:

1. *Missing or extra parentheses*: Forgetting to close a function call or condition with a parenthesis, or having an unnecessary parenthesis can trigger a **SyntaxError**.
2. *Misplaced or missing symbols*: This can be a missing {} in a block of code, or a missing : in object literals, or forgetting to use ; at the end of a statement.
3. *Improper string usage*: Strings need to be enclosed by either single quotes ', double quotes ", or backticks ``` . **If these aren't used correctly, such as a missing closing quote, it could lead to a** SyntaxError `.
4. *Incorrect usage of reserved keywords*: JavaScript reserves a set of keywords for its syntax. Using these reserved keywords incorrectly, such as

declaring a variable with the name **function**, would cause a **SyntaxError**.

Examples and Solutions for SyntaxError

Consider the following piece of code:

```
let name = 'John Doe
```

This will cause a **SyntaxError** because the string **John Doe** was not properly closed with a matching single quote. The correct version of this code would be:

```
let name = 'John Doe';
```

Here's another common example:

```
if (name = 'John Doe') {
```

```
  console.log('Hello, John!');


}
```

This will cause a **SyntaxError** because the assignment operator (=) was used instead of the equality operator (== or ===). The correct version should look like this:

```
if (name === 'John Doe') {

  console.log('Hello, John!');


}
```

Best Practices to Avoid SyntaxError

Avoiding **SyntaxError** mostly requires a keen eye, attention to detail, and a good understanding of JavaScript's syntax. Here are some best practices:

1. *Always pair your symbols*: Every opening symbol in JavaScript has a matching closing symbol. Always ensure that every **{** has a corresponding **}**, every **(** has a corresponding **)**, and every **[** has a corresponding **]**.
2. *Mind your semicolons*: While JavaScript can often infer where semicolons should be placed, it's best not to rely on this. Always add a **;** at the end of your statements.
3. *Properly quote your strings*: Remember to open and close your strings properly. Use a linter to help spot any unclosed or incorrectly quoted strings.
4. *Use a linter or an IDE*: Linters and modern IDEs can catch most syntax errors before the code is even run. They check your code against a set of

rules and can highlight potential issues in real-time.

5. *Understand reserved keywords*: Avoid using reserved keywords for variable or function names to prevent confusion and potential syntax errors.

As your journey into JavaScript continues, remember that every **SyntaxError** is an opportunity to better understand the language's rules and syntax. With every mistake made, a new lesson is learned. Happy coding!

Understanding ReferenceError

A **ReferenceError** is thrown when you attempt to reference a variable that does not exist. In other words, it occurs when you try to use a variable that has not been declared or is not within the current scope. While it may seem like a simple mistake, dealing with **ReferenceError** requires a solid understanding of how variable scoping works in JavaScript.

Common Causes of ReferenceError

Some of the common causes of a **ReferenceError** include:

1. *Using an undeclared variable*: This is the most common cause of a **ReferenceError**. Trying to access a variable that has not been declared will result in this error.
2. *Variable scoping issues*: In JavaScript, variables declared with **let** and **const** are block-scoped, meaning they only exist within the block they are declared in. If you try to access these variables outside of their block, a **ReferenceError** will be thrown.
3. *Typographical errors*: Misspelling a variable name can cause a **ReferenceError** as the misspelled variable name would not have been declared.

Examples and Solutions for ReferenceError

Consider the following piece of code:

```
console.log(age);
```

If **age** has not been previously declared, a **ReferenceError** will be thrown. You can solve this by declaring the variable before using it:

```
let age = 25;

console.log(age); // Logs: 25
```

Here's another example demonstrating variable scoping:

```
if (true) {

  let message = 'Hello, world!';



}



console.log(message);  // ReferenceError: message is not defined
```

In this case, **message** is block-scoped, and doesn't exist outside of the **if** block. To fix this, you could declare **message** in a scope that is accessible to both the **if** block and the **console.log** statement:

```
let message;

if (true) {
  message = 'Hello, world!';
}

console.log(message);  // Logs: Hello, world!
```

Best Practices to Avoid ReferenceError

Avoiding **ReferenceError** requires understanding how and when to declare variables, as well as how variable scope works in JavaScript. Here are some best practices:

1. *Always declare your variables*: Before you use a variable, make sure it has been declared.
2. *Understand scope*: Familiarize yourself with how scope works in JavaScript, particularly the difference between block scope (for **let** and **const**) and function scope (for **var**).
3. *Avoid typos*: A good IDE or text editor can help catch typos in your variable names, preventing many potential **ReferenceError**s.
4. *Use strict mode*: Enabling strict mode (**'use strict'**) changes JavaScript's behavior in several ways to help you catch mistakes. In strict mode, trying to use a variable without declaring it will throw a **ReferenceError**.

Remember, **ReferenceError**s are not just errors, they're clues that can help you understand the intricacies of variable declaration and scope in JavaScript. With each **ReferenceError** you solve, you're one step closer to mastering JavaScript. Keep coding!

Understanding TypeError

A **TypeError** is an error that occurs in JavaScript when a value is not of the expected type, and thus, an operation could not be performed on that value. This means you are trying to perform an operation on a value whose type does not support that operation.

Common Causes of TypeError

1. *Invalid use of a function*: Functions in JavaScript are designed to work with certain types of arguments. If a function receives an argument of the wrong type, it may throw a TypeError.
2. *Incorrect context*: The **this** keyword in JavaScript refers to the object that a function is a method of. If you call a method with a context that it wasn't expecting, it may throw a TypeError.
3. *Undefined is not an object*: This error occurs when you try to access a property or method on **undefined** or **null**.

Examples and Solutions for TypeError

Let's consider some examples:

1. *Invalid use of a function*:

```
let str = "Hello, world!";

str.push('!');  // TypeError: str.push is not a function
```

In this case, the **push** method is not a valid method for string data types. To solve this error, you could instead use the **concat** method which is a valid method for strings:

```
let str = "Hello, world!";
```

```
str = str.concat('!');  // "Hello, world!"
```

1. *Incorrect context*:

```
let obj = {
  value: 'Hello, world!',

  printValue: function() {

    console.log(this.value);

  }
```

```
};
```

```
let print = obj.printValue;
```

```
print();  // TypeError: Cannot read properties of undefined (reading 'value')
```

Here, **this** in **print()** refers to the global object, not **obj**. To fix this, you could use the **call**, **apply**, or **bind** method to specify the context:

```
let obj = {
```

```
  value: 'Hello, world!',

  printValue: function() {

    console.log(this.value);

  }

};

let print = obj.printValue;
```

```
print.call(obj);  // "Hello, world!"
```

1. *Undefined is not an object*:

```
let obj = undefined;

console.log(obj.value);  // TypeError: Cannot read properties of undefined
```

(reading 'value')

You can avoid this by checking if the object is defined before trying to access its properties:

```
let obj = undefined;

if (obj) {


  console.log(obj.value);


}
```

Best Practices to Avoid TypeError

To avoid **TypeError**s, keep these best practices in mind:

1. *Use the correct types*: Make sure to provide the correct types of arguments when calling a function or method.
2. *Check for undefined or null*: Before accessing a property or method of an object, make sure the object is not **undefined** or **null**.
3. *Use strict mode*: Strict mode (**'use strict'**) can help you catch potential problems in your code, as it makes JavaScript more strict in what it allows.

With these best practices and a better understanding of **TypeError**, you'll be well-equipped to write robust, error-free JavaScript code. Keep coding, and remember, every error is a step towards becoming a better developer!

Understanding RangeError

A RangeError happens when you try to give a variable a value that it can't have. This often happens when you give a function a value that's not allowed. This can be tricky to fix, especially when using third-party libraries that don't explain what values are allowed.

Here are some common situations where a RangeError can happen:

- When you try to create an array with a length that's not allowed.
- When you give wrong values to number methods like toExponential(), toPrecision(), toFixed(), etc.
- When you give wrong values to string functions like normalize().

Now, let's see an example of how a RangeError can be made:

```
let arr = new Array(-1);  // This will cause a RangeError
```

Output: RangeError: Invalid array length

Common Causes of RangeError

A `RangeError` in JavaScript is typically caused when a value is not within an expected range. Here are some common causes:

1. **Invalid array length:** In JavaScript, an array's length cannot be negative or larger than a certain maximum value. If you try to create an array with an invalid length, it will cause a `RangeError`. For example, `new Array(-1)` or `new Array(Number.MAX_SAFE_INTEGER + 1)` will both result in a `RangeError`.

2. **Incorrect usage of number methods:** Certain number methods in JavaScript, such as `toFixed()`, `toExponential()`, and `toPrecision()`, expect arguments within specific ranges. If you pass a value outside of these ranges, it will cause a `RangeError`. For instance, `Number(1).toFixed(101)` will cause a `RangeError` because the `toFixed()` method only accepts values between 0 and 100 inclusive.

3. **Invalid usage of String methods:** Some String methods like `String.prototype.normalize()` accept certain string values representing Unicode normalization forms. Passing an unsupported form will result in a `RangeError`. For example, `'some string'.normalize('unsupportedForm')` will cause a `RangeError`.

4. **Recursion limits:** JavaScript has a maximum call stack size. If a recursive function exceeds this limit, it will result in a `RangeError`. This is often the case when a function ends up calling itself indefinitely due to a missing or incorrect base case.

To avoid these common causes of `RangeError`, ensure you understand the expected ranges of the methods and constructors you're using, validate your inputs, and carefully manage recursive function calls.

Here's an example showing some of these principles in action:

```
function createArray(length) {

  // Validate input


  if (typeof length !== 'number' || length < 0) {


    console.log('Invalid length, defaulting to 0');


    length = 0;  // Use a default value
```

```
}


// Try to create array and handle potential RangeError

try {

  return new Array(length);

} catch (error) {
```

```
    if (error instanceof RangeError) {

      console.log('Failed to create array: ' +
error.message);

    } else {

      throw error;  // Rethrow unexpected errors

    }

  }
```

```
}

createArray(-1); // Invalid length, defaulting to 0
```

Here's what each part of the code does:

- **function createArray(length)** { ... }: This is a function called **createArray** that takes one argument, length.
- **if (typeof length !== 'number' || length < 0) { ... }:** This is an if statement that checks if length is not a number or if it's less than 0. If either of these conditions is true, it means that length is not a valid array length.

- **console.log('Invalid length, defaulting to 0');: If length is not valid**, this line prints a message to the console.
- **length = 0;:** This line sets length to **0**, a default value that's always a valid array length.
- **try { ... } catch (error) { ... }**: This is a try/catch block. The code inside the try part is attempted, and if it throws an error, the code inside the catch part is executed.
- **return new Array(length);**: Inside the try part, the function tries to return a new array with length length. If length is not a valid array length (even after our previous checks), this line will throw a RangeError.
- **if (error instanceof RangeError) { ... } else { ... }**: Inside the catch part, this if/else statement checks if the error is a RangeError. If it is, it prints a message to the console. If it's not, it throws the error again.
- **createArray(-1);**: This line calls the createArray function with -1 as an argument. Because -1 is

not a valid array length, the function will end up using the default length of 0.

Best Practices to Avoid RangeError

To avoid a **RangeError**, you should ensure that the values you're using fall within the expected or valid range. Here are some strategies to help you avoid this error:

1. **Know your functions**: Understanding the functions you're using is the first step in avoiding **RangeError**s. Make sure you know what kind of values the function expects and what it considers out of range. For instance, in JavaScript, the **Array** constructor expects a non-negative integer. Providing anything else will lead to a **RangeError**.
2. **Validate inputs**: Before passing values to a function, validate them to ensure they're within the expected range. This is especially important when the values come from user input or external sources.
3. **Exception handling**: Use try/catch blocks to handle potential **RangeError**s gracefully. While

it's better to prevent an error than to handle it after it occurs, this can provide a safety net and prevent your program from crashing.

4. **Test your code**: Write tests for your functions, especially edge cases that might cause **RangeErrors**. This can help you catch potential errors before they become a problem.
5. **Use default values or constraints**: If a function receives a value outside of its valid range, consider using a default value or applying a constraint to bring the value back into range.

Understanding EvalError

An **EvalError** is a specific type of error in JavaScript that is thrown as a result of an error occurring in the **eval()** function. The **eval()** function is used in JavaScript to evaluate a string of code. However, the use of **eval()** is often discouraged due to security and efficiency concerns.

Common Causes of EvalError

The main cause of an **EvalError** is the misuse of the **eval()** function. This can occur when:

1. You try to use **eval()** in a "strict mode" context. In "strict mode", using **eval()** in a way that creates new variable or function declarations will lead to an **EvalError**.
2. Passing invalid code to the **eval()** function.

Examples and Solutions for EvalError

1. *Use of eval() in strict mode*:

```
'use strict';
  eval('var x = 5'); // EvalError: Strict mode code may not include a
  direct call to 'eval' or certain 'eval'-like functions
```

In this case, you should avoid creating new variables within **eval()**. Instead, declare variables outside of **eval()**, like so:

```
'use strict';

var x;
```

```
eval('x = 5'); // x is now 5
```

1. *Passing invalid code to eval()*:

```
eval('console.lo("Hello, world!")'); // EvalError: Invalid code
```

To fix this error, correct the JavaScript code that you're passing to **eval()**:

```
eval('console.log("Hello, world!")'); // Prints: "Hello, world!"
```

Best Practices to Avoid EvalError

To avoid **EvalError**s, keep the following best practices in mind:

1. *Avoid using eval() when possible*: **eval()** is a potential security risk and is often inefficient compared to other JavaScript operations. Always look for an alternative before resorting to **eval()**.
2. *Use strict mode*: While it can cause **EvalError**s when misused, "strict mode" can also help catch errors that might otherwise be missed.
3. *Validate input*: If you must use **eval()**, ensure that the input is valid JavaScript code. This is especially important if the input comes from an external source.

Understanding **EvalError** and using **eval()** responsibly can greatly enhance the reliability and security of your JavaScript code.

Understanding URIError

A **URIError** is an object representing an error that occurs when the global URI handling functions are used in a wrong manner. It's thrown when the functions **encodeURI()**, **decodeURI()**, **encodeURIComponent()**, **decodeURIComponent()**, **escape()** or **unescape()** are passed an invalid parameter, such as a URI that is incorrectly structured.

Common Causes of URIError

The main cause of a **URIError** is the improper use of URI handling functions. For example, this error is typically thrown when:

1. **decodeURI()** or **decodeURIComponent()** are passed a URI that is not correctly encoded.
2. Incorrectly using **encodeURIComponent()** or **encodeURI()** on a URI that is already encoded.

Examples and Solutions for URIError

1. *Incorrect decoding*:

```
decodeURIComponent('%');  // URIError: URI malformed
```

This error occurs because the "%" symbol is expecting two hexadecimal digits to follow it. To fix the error, make sure that any encoded URI you're trying to decode is properly formed:

```
decodeURIComponent('%20');  // Returns: " "
```

1. *Double encoding*:

```
encodeURI('https://example.com/?q=' +
encodeURIComponent(encodeURICompo-
nent('JavaScript Errors')));

// Returns: "https://example.com/?q=JavaScript
%2520Errors"
```

This results in double encoding of the spaces (**%2520** instead of **%20**). This can be avoided by only encoding the URI component once:

```
encodeURI('https://example.com/?q=' +
encodeURIComponent('JavaScript Errors'));

// Returns: "https://example.com/?q=JavaScript
%20Errors"
```

Best Practices to Avoid URIError

To prevent **URIError**s, keep these practices in mind:

1. *Validate input*: Always check the inputs to your URI encoding and decoding functions to ensure they are correctly formatted.
2. *Handle errors*: Use try/catch blocks around your URI handling code to catch and handle **URIError**s when they occur.
3. *Avoid double encoding or decoding*: Ensure you are not encoding or decoding a URI component more than once.

By taking the time to understand **URIError** and following best practices for handling URIs, you can write more robust JavaScript code.

Understanding InternalError

InternalError is an error type that is thrown when JavaScript's internal algorithms fail, typically due to exceeding the JavaScript engine's maximum recursion depth. This error is not often seen by developers as it is primarily used internally by the JavaScript engine.

Common Causes of InternalError

The most common cause of an **InternalError** is a function that calls itself, known as a recursive function, without an adequate terminating condition. This can result in the function calling itself indefinitely, or until the maximum recursion depth is exceeded.

Examples and Solutions for InternalError

1. *Infinite Recursion*:

```
function recurse() {
  recurse();

}

recurse();  // InternalError: too much
```

recursion

In this example, the **recurse** function calls itself without any terminating condition, leading to an **InternalError**. To fix this, you should add a condition to terminate the recursion:

```
function recurse(num) {
  if (num <= 0) {

    return;

  }

  recurse(num - 1);
```

```
}

recurse(100); // No error
```

In this adjusted function, recursion stops when **num** is not greater than 0, preventing an infinite loop of function calls.

Best Practices to Avoid InternalError

To prevent **InternalError**s, it's important to keep the following best practices in mind:

1. *Properly terminate recursive functions*: Ensure that every recursive function has a clear and correct termination condition to prevent infinite recursion.
2. *Limit the depth of recursion*: If you're dealing with a potentially high level of recursion, it might be better to use an iterative solution instead.
3. *Exception handling*: As with all error types, it's a good practice to handle potential errors with try/catch blocks to prevent an unexpected application crash.

In general, you won't encounter **InternalError**s often, but understanding them can help you write safer, more effective JavaScript code.

Understanding AggregateError

The **AggregateError** object takes multiple errors and wraps them into a single error. This is especially helpful when dealing with multiple promises running in parallel where one or more may fail, as with **Promise.allSettled()** or **Promise.any()** methods.

Common Causes of AggregateError

AggregateError comes into play when multiple errors need to be thrown at once. It's commonly associated with operations that handle multiple promises, where individual promises may fail and throw their own errors.

Examples and Solutions for AggregateError

Here's an example where several promises fail:

```
let promise1 = Promise.reject(new Error("An error occurred with promise1"));

let promise2 = Promise.reject(new Error("An error occurred with promise2"));

let promise3 = Promise.resolve("Promise3 resolved");

let promises = [promise1, promise2, promise3];
```

```
Promise.allSettled(promises).then((results) =>
results.forEach((result) =>
console.log(result.status)));
```

```
// Output: rejected, rejected, fulfilled
```

In this scenario, we don't see an **AggregateError** because **Promise.allSettled()** does not throw errors for rejected promises.

But, if we use **Promise.any()**, we will see an **AggregateError**:

```
Promise.any(promises).catch((error) =>
console.log(error));
```

```
// Output: AggregateError: All promises were rejected
```

The **Promise.any()** function rejects with an **AggregateError** when all promises are rejected.

To handle this error, we should analyze the individual errors contained in the **AggregateError**.

```
Promise.any(promises).catch((error) => {

  if (error instanceof AggregateError) {


    for (let e of error.errors) {
```

```
        console.log(e);

    }

}

});

// Output: Error: An error occurred with promise1

//       Error: An error occurred with promise2
```

Best Practices to Avoid AggregateError

While avoiding an **AggregateError** isn't always possible due to its nature, here are some practices that could minimize the likelihood of encountering one:

1. *Error handling in promises*: Try to handle errors within each promise to ensure they resolve successfully.
2. *Validate inputs before execution*: If promises depend on user input or other changeable data, validate this data before executing the promise.
3. *Use the right Promise function*: Consider whether functions like **Promise.all()**, **Promise.allSettled()**, or **Promise.any()** are best for your needs. Each function has different error-handling characteristics.

Understanding the try...catch Statement

Examples and Uses of try...catch

Best Practices for Using try...catch

Explanation: In JavaScript, primitive types are not objects and thus, they don't have methods or properties. However, JavaScript provides built-in constructor functions or wrapper objects (String, Number, and Boolean) that wrap these primitives and provide them with object-like behavior, such as methods and properties. But this can sometimes lead to unexpected results, especially when these are used with the strict equality operator. This is because the strict equality operator compares both the type and the value. So, even if the value is the same, if one side is a primitive and the other is a wrapper object, they are considered different types, and the comparison returns false.

Examples with Solutions:

1. **Example of strict equality comparison between a number and its corresponding Number object:**

```
(function(n) {

    return n === new Number(n);

})(10);  // Outputs: false
```

Solution: Use the valueOf() method to get the primitive value of the Number object for comparison:

```
(function(n) {
    return n === new Number(n).valueOf();

})(10);  // Outputs: true
```

1. **Example of strict equality comparison between a string and its corresponding String object:**

```
(function(x) {
    return new String(x) === x;
```

```
})('a');  // Outputs: false
```

Solution: Use the valueOf() method to get the primitive value of the String object for comparison:

```
(function(x) {

    return new String(x).valueOf() === x;

})('a');  // Outputs: true
```

To avoid these kinds of errors, be aware of the difference between primitives and their corresponding wrapper objects in JavaScript. When you need to compare a primitive with its corresponding wrapper object, it's safer to convert the object to a primitive using the valueOf() method, which returns the primitive value of a wrapper object. This way, you ensure that you're comparing the same types, and the strict equality comparison will work as expected.

Explanation: In JavaScript, null is considered an object when evaluated using the typeof operator, which can lead to confusion. Additionally, when checking if an object exists or has been defined, null and undefined need to be handled correctly to avoid false positives or potential errors.

Examples with Solutions:

1. **Example of null being of type 'object':**

```
console.log(typeof null); // Outputs: "object"
```

1. **Example of false positive when checking for 'undefined':**

```
var obj = null;

if (obj !== "undefined") {

  console.log("Object exists"); // This would erroneously execute
```

```
}
```

*Solution: Use **typeof** operator to correctly check if **obj** is undefined:*

```
if (typeof obj !== "undefined") {

  console.log("Object exists"); // This wouldn't execute if obj is null

}
```

1. **Example of false positive when checking for 'null' and 'undefined':**

```
var obj = null;

if (obj !== null && typeof obj !== "undefined") {


  console.log("Object exists"); // This would not execute


}
```

Solution: Swap the order of the conditions to handle ***undefined*** *first:*

```
if (typeof obj !== "undefined" && obj !== null) {

  console.log("Object exists"); // This would not execute if obj is null or undefined


}
```

To avoid these kinds of mistakes, it's important to understand how JavaScript handles null and undefined. When checking for the existence of a variable, always use the typeof operator first to check if the variable is undefined. Then, check for null to ensure the variable has been both declared and assigned a non-null value. Additionally, be aware that typeof null returns 'object', which is a quirk of JavaScript. If you need to specifically check for null, do so explicitly.

In JavaScript, using the strict equality operator (===) to compare two objects doesn't compare their content; instead, it compares their references. Even if two objects have the same properties and values, if they are not the same instance or do not reference the same object, they are considered different.

Example of false comparison between similar objects:

```
console.log({a: 1} === {a: 1});  // Outputs: false


// Solution: A common way to compare objects is by JSON stringifying them:


console.log(JSON.stringify({a:          1})          ===
JSON.stringify({a: 1}));  // Outputs: true
```

Example of false comparison between similar arrays:

```
console.log([1,2,3] === [1,2,3]); // Outputs: false
```

// Solution: Same as with objects, a common way to compare arrays is by JSON stringifying them:

```
console.log(JSON.stringify([1,2,3])                    ===
JSON.stringify([1,2,3])); // Outputs: true
```

Example of true comparison when objects share the same reference:

```
var obj1 = {a: 1};

var obj2 = obj1;
```

```
console.log(obj1 === obj2);  // Outputs: true
```

The key way to avoid this kind of mistake is by understanding that objects, arrays, functions, and other non-primitive types in JavaScript are compared by reference, not by value. If you need to compare these types of values, consider using JSON.stringify() or writing a custom comparison function that examines each property of the objects. It's also worth noting that JSON.stringify() has limitations, as it won't correctly compare objects that have different orders of properties or that contain JavaScript-specific objects like Date or RegExp.

In JavaScript, implicit (hidden) type conversions refer to conversions that aren't immediately obvious, carried out by the JavaScript engine as a side effect of other operations. This is most common when the value of a particular type is used in a way that automatically results in its conversion.

Explanation

Associativity of operations and logical value conversion to number: The JavaScript '<' operator is a comparison operator with its own set of rules. For instance, if we consider this operation '3 < 2 < 1', from a mathematical point of view, it's incorrect, but in JavaScript, it returns TRUE. This is because the associativity of the '<' operator is from left to right, causing implicit conversion of FALSE (the result of '3 < 2') to the number '0', so the expression becomes '0 < 1' which returns TRUE.

Implicit Conversion to Logical Type in Comparison: When logical operators (&& or ||) are used to compare operands that are not of logical type, there is implicit conversion of operands to a logical type due to the conditions of the operator. However, these

operators return the value of one of the two operands, not a logical value. The resulting condition is determined according to these rules:

For the || operator, if the condition is TRUE, it returns the value of the first operand; if FALSE, it returns the value of the second operand.

For the && operator, if the condition is FALSE, it returns the value of the first operand; if TRUE, it returns the value of the second operand.

Implicit Conversion of Object to String: When attempting to print an object in the console along with some accompanying text, you get an unexpected result because of implicit conversion of the object to a string. This can be solved by converting the JavaScript object to a string using the JSON.stringify() method before printing.

Implicit Conversion of Array to String: When an array is implicitly converted to a string as a result of concatenation, you get a slightly different result. For example, using the '+" operator with operands that are not numbers results in string concatenation, not

addition. When arrays are involved, the elements are joined with a comma separator.

Ways to Avoid These Errors

1. **Understand JavaScript Type Conversion:** Understanding the way JavaScript handles type conversions can help you avoid related errors. Be aware of how different types of values are implicitly converted in different situations, and test your code thoroughly to make sure it behaves as expected.
2. **Explicit Conversion:** When possible, it's best to convert types explicitly. This way, you have control over the conversion and can ensure it behaves as you expect.
3. **Use Strict Comparison Operators:** JavaScript has strict comparison operators ('===' and '!==') that do not convert operand types. This can be useful if you want to avoid implicit type conversion.
4. **Using JSON.stringify() for Objects:** When needing to convert an object to a string, instead of relying on implicit conversion, use

JSON.stringify() to convert the object into a JSON string.

5. **Avoid Using + Operator with Non-Numeric Types:** If you're trying to concatenate strings, it's best to use the String.concat() method or template literals, which don't cause implicit type conversion.

Sure, here are some expanded examples with solutions demonstrating the concepts of implicit type conversion in JavaScript:

1. Associativity of operations and logical value conversion to number:

```
console.log(3 < 2 < 1); // Outputs: true

// This can be resolved by adding parentheses:

console.log(3 < (2 < 1)); // Outputs: false
```

2. Implicit Conversion to Logical Type in Comparison:

```
var x = 0;

var y = "0";



var z = [];



console.log(x || y || z); // Outputs: "0"



// To ensure logical comparison works correctly, explicit conversion can be done:
    console.log(Boolean(x) || Boolean(y) || Boolean(z)); // Outputs: true
```

3. Implicit Conversion of Object to String:

```
var obj = {a:1, b:2};
```

```
console.log('The object is: ' + obj);  // Outputs: "The object is: [object Object]"



// Solution:

var objString = JSON.stringify(obj);

console.log('The object is: ' + objString);  // Outputs: "The object is: {"a":1,"b":2}"
```

4. Implicit Conversion of Array to String:

```
var array = [1, 2, 3];

console.log("The array contains: "+ array);     // Outputs "The array contains: 1,2,3"



// Solution: explicitly convert the array to string using join() method

console.log("The array contains: "+ array.join(" "));  // Outputs "The array contains: 1 2 3"
```

5. Implicit Conversion of null to other types:

```
console.log(null + 1); // Outputs: 1


// Solution: handle null cases explicitly


console.log((null ? 0 : null) + 1); // Outputs: 1
```

6. Implicit Conversion with equality operators:

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


// Solution: use the strict equality operator
```

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

7. Implicit conversion with boolean values:

```
console.log(true + false); // Outputs: 1

// Solution: convert boolean values to numbers explicitly

console.log(Number(true) + Number(false)); // Outputs: 1
```

8. Implicit conversion with Date objects:

```
var d = new Date();

console.log(d + 1); // Outputs: "Tue Jul 25 2023 10:01:52 GMT+0300 (Eastern European Summer Time)1"



// Solution: convert the Date object to a string or number explicitly



console.log(d.toString() + 1); // Outputs: "Tue Jul 25 2023 10:01:52 GMT+0300 (Eastern European Summer Time)1"



console.log(d.getTime() + 1); // Outputs: 1690430513001
```

Each of these examples illustrates a different aspect of implicit type conversion in JavaScript, and how it can be mitigated by explicit conversions or appropriate coding practices.

This is known as a floating point precision error. It's not an "error" in the typical sense, but rather a limitation of how computers represent decimal numbers in binary.

In the JavaScript programming language, and many others, numbers are usually represented in a format known as "double-precision floating point" according to the IEEE 754 standard for floating point arithmetic. This standard represents numbers as binary fractions, which can cause unexpected behavior when dealing with precise decimal values.

For example, while numbers like 0.5, 0.75, or 0.1 can be represented exactly in binary, others like 0.1 cannot. Instead, the binary representation is an approximation of the actual number, which can lead to precision errors when you perform operations like addition or multiplication.

In the case of adding 0.1 and 0.2 in JavaScript, the result is not exactly 0.3 but slightly more due to this

binary floating-point precision issue. When comparing this result to the expected result, it can lead to unexpected behaviors if you are not aware of this limitation.

Solutions

To mitigate this kind of issue, you can implement certain strategies in your code:

Round the results of your operations to a certain precision that makes sense in your specific use case.

```
var result = 0.2 + 0.1;

if (result.toFixed(4) == expectedResult){


  // Some code


}
```

Instead of comparing two numbers for equality, check if they are "close enough" by comparing the absolute difference to some small number.

```
var precision = 0.00001;

if (Math.abs(result - expectedResult) < precision){


  // Some code


}
```

Keep in mind these are just workarounds, the underlying floating point precision issue still exists in the language and is a fundamental aspect of how modern computers work. Understanding this can help

you avoid potentially confusing results in your calculations.

In JavaScript, the **switch** statement compares the expression inside the switch parenthesis with the case values using strict comparison (===). If the types do not match, the comparison is considered false, and the case is skipped. In the given example, because the number 5 and the string '5' are of different types, the case '5' is not executed, even though their values are the same when converted.

Examples with Solutions

1. **Example of a mismatched type in a switch statement:**

```
var myVar = 5;

switch(myVar){


  case '5':


  alert("hi"); // This will never be executed


}
```

*Solution: Use the **toString()** method to convert the number to a string before comparing:*

```
var myVar = 5;

switch(myVar.toString()){


  case '5':


  alert("hi"); // This will be executed now


}
```

Ways to Avoid Mistakes: When you're using a switch statement, remember that it uses strict comparison. This means that the type of your switch expression should match the type of your case values. If your case values are strings, make sure to convert your switch expression to a string before comparing, for example by using the toString() method, to ensure the correct case is executed

Explanation:

The **replace()** method in JavaScript replaces only the first match of the provided substring. If we want to replace all occurrences of the substring in the original string, we must use a global Regular Expression.

Examples with Solutions

1. **Example of replace() replacing only the first occurrence:**

```
var a = "bob or bob";
var word = "bob";
console.log(a.replace(word, "lol")); // Returns: "lol or bob"
```

pythonCopy code
Solution: Use a global Regular Expression to replace all occurrences:

```
var a = "bob or bob";
var pattern = /bob/g;
console.log(a.replace(pattern, "lol")); // Returns: "lol or lol"
```

Remember that the replace() method replaces only the first occurrence of the provided substring. To replace all occurrences, you should use a global Regular Expression. Ensure that the 'g' flag is added to your regular expression to enable global search and replace all matches, not just the first one.

For case-insensitive search and replace, you can add the 'i' flag along with the 'g' flag, like this: /bob/gi where 'g' stands for global, and 'i' for case-insensitive. This ensures that all instances, regardless of case, will be replaced.

If you use the **Array()** constructor with a single numerical argument, JavaScript interprets that argument as the desired length of the new array, not as the first element in that array. Thus, **new Array(3)** creates an empty array of length 3, not an array containing the single element 3. This can lead to unexpected results if you're not aware of this behavior.

Examples with Solutions:

Example of the issue:

```
new Array(3); // Returns an array [undefined, undefined, undefined]
    new Array(1, 2, Array(3)); // Returns an array [1, 2, [undefined, undefined, undefined]]
```

Solution: When you want to create an array with a single numerical element, use array literal notation instead of the Array constructor:

```
[3]; // Returns an array [3]
```

Example of the issue:

```
let arr = new Array(3);
    arr[0] = "zero";
    console.log(arr); // Logs ["zero", undefined, undefined]
```

Solution: If you want to initialize an array to a specific length but also populate it, consider using Array.from or Array.fill:

```
let arr = Array.from({length: 3}, (_, i) => i); // Returns [0, 1, 2]
```

How to Avoid This Mistake

Always use array literal syntax [] when creating arrays, especially when you want to initialize the array with values. If you need to create an array of a specific length, consider using Array.from({length: n}), which creates an array of length n, or Array.fill(), which creates a new array of a provided length and populates it with a provided value.

The JavaScript Date object counts months from 0 (January) to 11 (December). This is contrary to the day and year properties, which count from 1. This discrepancy can lead to confusion when creating dates. For example, **new Date(2023, 5, 20)** will result in June 20, 2023, not May 20, 2023, as might be expected.

Examples with Solutions:

1. **Example of the issue:**

new Date(2023, 5, 20); // Returns June 20, 2023

new Date(2023, 5, 31); // Returns July 1, 2023

Solution: Subtract 1 from the month number when creating a new Date object:

new Date(2023, 4, 20); // Returns May 20, 2023

When using the Date object in JavaScript, remember that months are zero-indexed. This means January is 0, February is 1, and so on, until December which is 11. If you find this confusing or counterintuitive, consider creating a helper function to adjust the month numbers, or use a date library like moment.js or date-fns that abstracts these inconsistencies away.

Explanation

A method in JavaScript is simply a property of an object that references a function. When we assign a method of an object to a variable, we're actually assigning a reference to the function to the variable. Therefore, when invoking this variable, we're not calling the object's method, but rather a normal function from the global scope. The keyword **this** inside a function being called from the global scope, according to the rules for **this**, points to the global object (window).

Example:

```
var osoba = {
    ime: "Pera",
    prezime: "Peric",
    punoIme: function () {
    return this.ime + " " + this.prezime;
    }
    }
    osoba.punoIme(); // Returns: Pera Perić

var prikazPunogImena = osoba.punoIme // Assigning the function reference to a variable
    prikazPunogImena(); // Returns: undefined undefined
```

Looking at the previous example from the perspective of this, we can insert the function in place of the method reference:

```
var prikazPunogImena = function () {
    return this.ime + " " + this.prezime;
```

```
}
prikazPunogImena(); // Returns: undefined un-
defined
```

From the above, we can see that the function prikazPunogImena() is being invoked, not an object's method. Therefore, the "default rule" is applied where this points to the global window object which doesn't have the ime and prezime properties, thus returning undefined, undefined.

To resolve this issue, you can use the bind() method to explicitly bind this in the function to the osoba object, after which it no longer matters from where the function is called.

```
var osoba = {
ime: "Pera",
prezime: "Peric",
punoIme: function () {
return this.ime + " " + this.prezime;
}
}
```

```
var prikazPunogImena = osoba.punoIme.bind(osoba) // Now `this` is bound to the `osoba` object
prikazPunogImena(); // Returns: Pera Peric
```

Understanding how this behaves in JavaScript is crucial to avoid this kind of problem. One way to make sure this refers to the correct context is by using methods like .bind(), which allows you to explicitly set what this should refer to.

Additionally, the use of arrow functions, which do not have their own this context, but rather inherit this from the parent scope, can be a useful strategy in certain contexts, especially if you're working in an environment that supports ES6. However, keep in mind that arrow functions may not always be the right solution for methods where you want this to behave as it does in traditional functions.

Variable shadowing occurs when a variable declared within a certain scope shares the same name as a variable declared in an outer scope. When the code is run, the inner variable "shadows" the outer variable, which can lead to unexpected behavior, as demonstrated in the provided code example. In this case, plata is declared in both the global scope and within the anonymous function scope.

There's also a related concept known as "hoisting", where JavaScript moves declarations (both variable and function) to the top of their respective scopes during the compile phase, before the code has been executed.

In the example you provided, the plata variable inside the function is hoisted to the top of the function scope and initialized as undefined. When the first console.log is run, it attempts to log the value of plata in the closest scope, which is undefined at that point. The variable plata is then assigned the value of "5000", and the next console.log correctly logs that value.

Example: Variable Shadowing and Hoisting

```
var salary = "1000";
(function () {
console.log("Initial salary is " + salary);
var salary = "5000";
console.log("New salary is " + salary);
})(); // Returns: "Initial salary is undefined" and "New salary is 5000"
```

How JavaScript Interprets This After Hoisting:

```
var salary = "1000";
(function () {
var salary; // undefined due to hoisting
console.log("Initial salary is " + salary);
salary = "5000";
console.log("New salary is " + salary);
})(); // Returns: "Initial salary is undefined" and "New salary is 5000"
```

To avoid such issues, always ensure that variables within a function have unique names that don't conflict with variables in outer scopes. If you need to use an outer variable inside a function, simply don't redeclare it inside the function. This allows you to access the outer variable without issue. It is considered good practice to limit the use of the same variable names across different scopes. Moreover, understanding the concept of hoisting can help you better understand why variables behave in certain ways in JavaScript.

JavaScript functions have two special kinds of objects available to them. One is the **arguments** object, which is an array-like object that holds all the arguments passed to the function. The other is the **length** property of the function itself, which tells us how many parameters are defined for the function.

1. **Length of a function**: The **length** property of a function in JavaScript returns the number of formal parameters (parameters that are written in the function definition) of that function. It doesn't care about how many arguments were passed in when calling the function.
2. **The arguments object**: Inside any JavaScript function, we can use a special object called **arguments**. This object contains an entry for each argument passed to the function. Importantly, **arguments.length** will return the number of arguments actually passed to the function, not the number of arguments that the function was declared with. This object is array-like but it's not a

real array, so it doesn't have typical array methods like **push** or **pop**.

Examples with Solutions:

Example 1: Length of a function

```
(function nekaFunkcija(a,b){}).length // Returns 2
```

Example 2: The **arguments** object

```
var result = (function(a, b, c) {
  delete arguments[0];

  return arguments.length;

})(5,7,9);
```

```
console.log(result); // Returns 3
```

In the above example, even though **delete arguments[0];** removes the first argument, it doesn't change the **length** of the **arguments** object, which is still 3 because three arguments were passed when the function was called. This shows how **arguments** is not a true array, because deleting an element in a true array would change the array's length.

Remember that arguments is not a true array and does not behave like an array in all ways. Always use the length property of a function to get the number of defined parameters, and use arguments.length to get the number of arguments passed to the function. Keep in mind that deleting an entry from the arguments object won't change arguments.length.

Explanation:

By default, **Array.prototype.sort()** converts each element in the array to a string and then sorts them based on the string's Unicode code point value. This is why **[10, 1, 5].sort()** results in **[1, 10, 5]**—because as strings, **"10"** comes after **"1"** and before **"5"**.

To sort an array of numbers in ascending order, you can pass a comparator function to the **sort()** method. The comparator function should take two arguments and return a negative, zero, or positive value depending on whether the first argument is considered smaller than, equal to, or larger than the second argument.

Examples with Solutions:

1. **Example of the issue:**

```
[10,1, 5].sort() // Returns [1, 10, 5]
```

*Solution: Provide a comparator function to **sort():***

```
[10,1, 5].sort((a, b) => a - b) // Returns [1, 5, 10]
```

1. **Example of the issue:**

```
[50, 2, 9, 1].sort(); // Returns [1, 2, 50, 9]
```

*Solution: Provide a comparator function to **sort():***

```
[50, 2, 9, 1].sort((a, b) => a - b); // Returns [1, 2, 9, 50]
```

How to Avoid This Mistake

Always remember that the default **sort()** method sorts elements as strings. When you need to sort an array of numbers, always provide a comparator function. For example:

```
let arr = [100, 200, 10, 20];
arr.sort((a, b) => a - b); // sorts arr in ascending numerical order
arr.sort((a, b) => b - a); // sorts arr in descending numerical order
```

Example

```
for (var i = 0; i < 5; i++){
    setTimeout(function(){
    console.log(i);
    }, 1000);
    } // Returns: 5 5 5 5 5
```

Explanation The setTimeout method is invoked while the loop is running. At the start of the loop, the initial value of "i=0", then the setTimeout function is invoked. The setTimeout function will call the callback function only after 10 seconds. While waiting for the callback function to execute, the loop continues to run, and now "i=1". The setTimeout function is immediately invoked again and will call the callback function again after 10 seconds. But, while waiting for it to execute, the loop continues and now "i=2". This procedure repeats until "i" gets the value 5. This entire previously mentioned process is executed in a very short period (measured in milliseconds).

The callback function is invoked after the loop ends After the first 10 seconds, the loop has already iterated over the variable "i" and "invokes" the first callback function, which therefore takes the value of the variable "i", which is 5. The next callback function activates after an additional 10 seconds, and uses the same variable, so the result is the same...

Solution: We can solve this problem by invoking the callback function at each loop iteration, so that it "grabs" the value of "i" at that moment. In this way, we create a new function for each pass through the loop, which, thanks to the closure characteristics, is able to "remember" the assigned value of "i" at that round.

Method I: In this example, we invoke the function at each loop iteration using an Immediately Invoked Function Expression (IIFE), providing it with unique values for each loop.

Example

```
for (var i = 0; i < 5; i++){
    (function(i){
```

```
setTimeout(function(){
console.log(i);
}, 1000);
})(i);
} // Returns: 0 1 2 3 4
```

Method II: This method uses the property of the ES6 keyword let, which defines a new variable "i" at each iteration of the loop:

Example

```
for (let i = 0; i < 5; i++){
setTimeout(function(){
console.log(i);
}, 1000);
} // Returns: 0 1 2 3 4
```

Method III: In this example, we extract the callback function so that we can invoke it in the setTimeout() function syntax at each pass of the loop. The closure will remember that value at the moment of invocation, and subsequent changes to the variable value will not affect it, i.e., "redeclaring the variable"

Example

```
function someFunction (i){
    return function(){
    console.log(i);
    };
    };

for (var i = 0; i < 5; i++){
    setTimeout(someFunction(i), 10000);
    }
```

One possible solution is to place all the code in an IIFE that will be invoked at each loop iteration, and the closure will remember the passed value.