Day 3: Try, Catch, and Finally

All topics

String Basics

Table Of Contents

Go to Top

JavaScript Strings

These are chains of zero or more Unicode characters (i.e., letters, digits, and punctuation marks) used to represent text.

We denote string literals by enclosing them in single (`'`) or double (`"`) quotation marks. Double quotation marks can be contained in strings surrounded by single quotation marks (e.g., `'"'` evaluates to `"`), and single quotation marks can be contained in strings surrounded by double quotation marks (e.g., `"'"` evaluates to `'`). In addition, you can also enclose a single or double quotation within another quotation of its same type by preceding the quotation you wish to have interpreted literally with the escape character (`\`).

Each string has a property called `String.length` denoting the length of, or number of characters in, the string. For example, if we write `var s = "Hello."`, the value of `s.length` is 6.

- EXAMPLE

```
var input = "";
process.stdin.on('data', function (data) {
    input = String(data).trim();
    main();
});
/** Ignore above this line. **/

function main() {
    console.log("\"" + input + "\"");
    console.log('\'' + input + '\'');
    console.log(input.length);
```

Input

```
HackerRank
```

Run

Output

String Constructor

To create a new string, we use the syntax `String(value)` where $value$ denotes the data we want to turn into a string.

Click *Run* below to see the string constructor in code.

```
var myNumber = 4;
var myString = String(myNumber);

console.log(myNumber + " is a " + typeof myNumber);
```

Output

Run

Observe that the value `4` assigned to the $myNumber$ variable is typed as a *number*, and that the $myString$ variable is typed as a string because we passed $myNumber$ to the string constructor.

Methods

Here are some frequently used methods that operate on strings:

`String.charAt()`

Returns the character at the specified index. For example:

```
var input = "";
process.stdin.on('data', function (data) {
    input = data;
    main();
});
function readLine() { return input; }
/** Ignore above this line. **/
var s = "HackerRank";
var i = +(readLine());
```

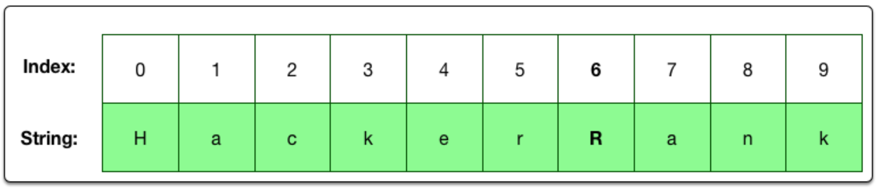
Input

```
6
```

Run

Output

Go to Top



Index:	0	1	2	3	4	5	**6**	7	8	9
String:	H	a	c	k	e	r	**R**	a	n	k

`String.concat()`

Returns a new string consisting of the calling string concatenated with another string passed as an argument. For example:

```javascript
var s = "Hacker";
var t = "Rank";
var u = s.concat(t);
console.log(s + " " + t);
```

Output

Run

`String.includes()`

Returns a boolean denoting whether a string passed as an argument exists within the calling string. For example:

```javascript
var s = "HackerRank";
console.log(s.includes() + " " + s.includes(""));
```

Output

Run

`String.endsWith()`

Returns a boolean denoting whether the calling string ends with the characters of another string passed as an argument. For example:

```javascript
var s = "HackerRank";
console.log(s.endsWith() + " " + s.endsWith(""));
```

Output

Run

`String.indexOf()`

Returns an integer denoting the index within the calling String object of the *first* occurrence of the given argument:

- If the argument isn't found in the string or no argument was passed to the function, it returns `-1`.
- If an integer is passed as a second argument, it will start searching the string from the index denoted by the integer.

Go to Top

For example:

```
var s = "HackerRank";
console.log(
    s.indexOf("a") + " "
    + s.indexOf("a", s.length) + " "
    + s.indexOf("a", 6)
);
console.log(
    s.indexOf("an") + " "
    + s.indexOf("x") + " "
    + s.indexOf("")
```

Output

Run

`String.lastIndexOf()`

Returns an integer denoting the index within the calling String object of the *last* occurrence of the given argument:

- If the argument isn't found in the string or no argument was passed to the function, it returns `-1`.
- If an integer is passed as a second argument, it will search the string backward starting from the index denoted by the integer.

For example:

```
var s = "HackerRank";
console.log(
    s.lastIndexOf("a") + " "
    + s.lastIndexOf("a", s.length) + " "
    + s.lastIndexOf("a", 6)
);
console.log(
    s.lastIndexOf("ac") + " "
    + s.lastIndexOf("x") + " "
    + s.lastIndexOf("")
```

Output

Run

`String.match()`

Match a regular expression passed as an argument against the calling string. If a match is found, it returns an object with three properties: the matched substring, the `index` it was found at, and the `input` (i.e., the initial string); if no match is found, it returns null. For example:

```
var s = "HackerRank";
console.log(s.match());
```

Output

Run

`String.normalize()`

Returns a string containing the Unicode Normalization Form of the calling string's value.

Go to Top

The argument must be one of the following:

- `"NFC"`: Normalization Form Canonical Composition. This is the default in the event that no argument is given.
- `"NFD"`: Normalization Form Canonical Decomposition.
- `"NFKC"`: Normalization Form Compatibility Composition.
- `"NFKD"`: Normalization Form Compatibility Decomposition.

For example:

```
var s = "HackerRank";
console.log(s.normalize());
```

Output

Run

`String.repeat()`

Returns a string consisting of the elements of the calling String object repeated some number of times (given as an integer argument). If no argument or a 0 argument are given, then it returns the empty string. For example:

```
var s = "HackerRank";
console.log(s.repeat () + "x" + s.repeat(0));
```

Output

Run

`String.replace()`

Finds a match between a regular expression and a string, then returns a string where the first matched substring is replaced with a new substring. If no match is found, the returned string is the same as the original string. For example:

```
var s = "HackerRank";
```

Output

Run

`String.search()`

Executes the search for a match between a regular expression and a specified string, then returns the index of the first character of the first match. For example:

```
var s = "HackerRank";
console.log(s.search() + " " + s.search("[a-z]"));
```

Output

Run

`String.slice()`

Go to Top

Extracts a section of a string and returns it as a new string. The extracted section depends on the arguments passed to the function:

- If no arguments are passed to the function, it returns the entire string.
- If one integer argument, i, is passed to the function, it returns a substring starting at index i and ending at the end of the string.
- If two integer arguments, i and j, are passed to the function, it returns a substring consisting of characters in the range $[i, j)$; in other words, this is a substring starting at index i and ending at $j - 1$. For example:

If one (or both) of the arguments passed to this function is negative, then the index corresponding to that argument is calculated as `String.length` minus the given argument. For example:

```
var s = "HackerRank";
console.log(s.slice(0, 6) + " " + s.slice(6));
console.log(
    s.slice() + " "
    + s.slice(-4, 8) + " "
    + s.slice(-4, -2)
```

Output

Run

`String.split()`

Splits a String object into an array of strings by separating the string into substrings:

- If no argument is given, it returns an array containing the original string.
- If the empty string is passed as an argument, it returns an array of the string's individual letters.
- If a string consisting of one or more letters is passed as an argument, it splits the string at each occurrence of that string and returns an array of the split substrings.

For example:

```
var s = "HackerRank";
console.log(s.split(""));
console.log(s.split("k"));
```

Output

Run

`String.startsWith()`

Returns a boolean denoting whether a string begins with the characters of another string passed as an argument. For example:

```
var s = "HackerRank";
console.log(s.startsWith("Hack"));
```

Output

Run

`String.substr()`

Go to Top

Returns a substring consisting of characters in a given range, depending on the arguments passed to the function:

- If no arguments are passed to the function, it returns the entire string.
- If one integer argument, i, is passed to the function, it returns a substring starting at index i and ending at the end of the string.
- If two integer arguments, i and j, are passed to the function, it returns a substring consisting of characters in the range $[i, j)$; in other words, this is a substring starting at index i and ending at $j - 1$.

For example:

```
var s = "HackerRank";
console.log(s.substr());
```

Output

Run

`String.substring()`

Returns a substring consisting of characters in a given range, depending on the arguments passed to the function:

- If no arguments are passed to the function, it returns the entire string.
- If one integer argument, i, is passed to the function, it returns a substring starting at index i and ending at the end of the string.
- If two integer arguments, i and j, are passed to the function, it returns a substring consisting of characters in the range $[i, j)$; in other words, this is a substring starting at index i and ending at $j - 1$.

For example:

```
var s = "HackerRank";
console.log(s.substring());
```

Output

Run

`String.toLowerCase()`

Returns the calling string's value, converted to lowercase letters. For example:

```
var s = "HackerRank";
s = s.toLowerCase();
```

Output

Run

`String.toUpperCase()`

Returns the calling string's value, converted to uppercase letters. For example:

```
var s = "HackerRank";
s = s.toUpperCase();
```

Go to Top

Output

Run

`String.trim()`

Returns the string with the whitespace trimmed from its beginning and end. This is part of the *ECMAScript 5* standard. For example:

```
var s = "    HackerRank    ";
console.log("Original Length: " + s.length);
s = s.trim();
```

Output

Run

`String.trimLeft()`

Returns the string with the whitespace trimmed from its left side. For example:

```
var s = "    HackerRank    ";
console.log("Original Length: " + s.length);
s = s.trimLeft();
```

Output

Run

`String.trimRight()`

Returns the string with the whitespace trimmed from its right side. For example:

```
var s = "    HackerRank    ";
console.log("Original Length: " + s.length);
s = s.trimRight();
```

Output

Run

- DATA TYPES Recommended Article

Data Types

View

Related challenge for **String Basics**

Day 2: Conditional Statements: Switch

Success Rate: **98.61%** Max Score: **10** Difficulty:

Solve Challenge

Error Handling

Go to Top

JavaScript Errors

There are three types of errors in programming:

1. Syntax Error (Parsing Error)

In a traditional programming language, this type of error occurs at *compile time*; because JavaScript is an *interpreted* language, this type of error arises when the code is interpreted. When a syntax error occurs in JavaScript, only the code contained within the same *thread* as the syntax error is affected; independent code running in other threads will still be executed, as nothing in them depends on the code containing the error. For example, consider the following code containing a syntax error:

```
console.log("Hello"
```

This produces the following error: `SyntaxError: missing ) after argument list`. This is because we failed to add a closing parenthesis to our call to *console.log*.

2. Runtime Error (Exception)

Commonly referred to as *exceptions*, this type of error occurs during execution (i.e., after compilation or interpretation). Once a runtime error is encountered, an exception is *thrown* in the hope that it will be *caught* by a subsequent section of code containing instructions on how to recover from the error. Much like syntax errors, these affect the thread where they occured but allow other, independent threads to continue normal execution. For example, consider the following code containing a runtime error:

```
function sum(a, b) {}
add(2, 3)
```

This produces the following error: `ReferenceError: add is not defined`. This is because we attempted to call the *add* function without ever declaring and defining it.

3. Logical Error

These are some of the most difficult errors to isolate because they cause the program to operate without terminating or crashing, but the operations the code performs are not correct. Unlike syntax and runtime errors which arise due to some violation of the rules of the language, these errors occur when there is a mistake in your the code's logic.

- EXAMPLE

Click *Run* below to see this in code for a program that prints the *sum* of two integers. Can you spot the logical error?

Input Format

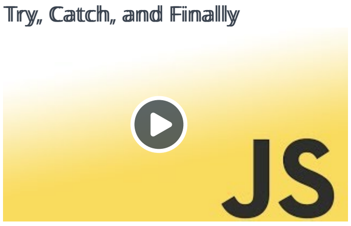
The first line must contain an integer denoting a.
The second line must contain an integer denoting b.

```
var input = "";
var inputLines = "";
var lineNumber = 0;
process.stdin.on('data', function (data) {
    input += data;
});
process.stdin.on('end', function () {
    inputLines = input.split("\n");
    main();
});
function readLine() { return inputLines[lineNumber++]; }
/**** Ignore above this line. ****/

function sum(a, b) {
    return a - b;
}
```

Table Of Contents

Try, Catch, and Finally



Throw

Go to Top

```
function main() {
    var a = +(readLine());
    var b = +(readLine());

    console.log( sum(a, b) );
```

Input

```
2
3
```

Run

Output

Solution

In the code above, we're reading two integers from stdin and passing them to our *sum* function. We expect the function to return the sum of two integers, a and b; however, due to a logical error (used the `-` operator instead of the `+` operator), it's actually returning the *difference* between a and b. This program is functional in that it runs, but it is broken in that it does not properly calculate the sum of two integers.

Tip: When trying to isolate logical errors in code, it's often helpful to print the contents of your variables to *stderr* (*standard error*) at various stages in the logic using `console.warn()` or `console.error()`. For example, if we used this version of the *sum* function instead:

```
function sum(a, b) {
    var result = a - b;
    console.error("The sum of " + a + " and " + b + " is " + result);
    return result;
}
```

The following line would be printed to our error console during execution: `The sum of 2 and 3 is -1`. This makes it obvious during debugging that there is a logical issue with how our function calculates the sum of a and b.

Try, Catch, and Finally

The *try* block is the first step in error handling and is used for any block of code that is likely to raise an exception. It should contain one or more statements to be executed and is typically followed by at least one *catch clause* and/or the optional *finally clause*. In other words, the *try* statement has three forms:

- *try-catch*
- *try-finally*
- *try-catch-finally*

The *catch* block immediately follows the *try* block and is executed only if an exception is thrown when executing the code within the *try* block. It contains statements specifying how to proceed and recover from the thrown exception; if no exception is thrown when executing the *try* block, the *catch* block is skipped. If any statement within the *try* block (including a function call to code outside of the block) throws an exception, control immediately shifts to the catch clause.

It's important to note that we always want to avoid throwing an exception. It's best if the contents of the *try* block execute without issue but, if an exception is unavoidable, control passes to the *catch* block which should contain instructions that report and/or recover from the exception.

The *finally* block is optional. It executes after the *try* and *catch* blocks, but before any

Go to Top

subsequent statements following these blocks. The *finally* block always executes, regardless of whether or not an exception was thrown or caught.

- EXAMPLE

In the code below, the call to `getElement(arr, 4)` inside the *try* block will throw an exception because the code declaring ***arr*** was commented out. It's immediately followed by a *catch* block that catches the exception and prints the *message* associated with it (`arr is not defined`). Because the exception was caught, the program continues executing, printing the next line after the *catch* block (`The program continued executing!`).

```
"use strict"

function getElement(arr, pos) {
    return arr[pos];
}


//let arr = [1, 2, 3, 4, 5];

try {
    console.log(getElement(arr, 4));
}
catch (e) {
    console.log(e.message);
}
```

Output

Run

If we remove the *catch* block and add the *finally* block shown below, it will instead print `Finally Block` and then terminate due to the uncaught exception (`ReferenceError: arr is not defined`).

```
//let arr = [1, 2, 3, 4, 5];

try {
    console.log(getElement(arr, 4));
}
finally {
    console.log("Finally Block");
}
console.log("The program continued executing!");
```

- EXAMPLE

In this example, we create a constant variable, ***arr***, that's an array of integers. We then *try* to reassign the value of ***arr*** to a different array of numbers; this throws an exception because the value of a constant cannot be reassigned or re-declared. Click *Run* below to execute the code.

```
"use strict";

const arr = [1, 2, 3, 4, 5];

try {
    arr = [4, 2];
    console.log(arr.sort());
}
catch (e) {
    console.log(e.message);
```

The code above produces the following output:

Go to Top

Output

Run

```
Assignment to constant variable.
```

Let's say we want to make sure that the contents of ***arr*** are printed to stdout regardless of whether or not an exception is thrown or caught; we can do this by adding a *finally* block:

```
"use strict";

const arr = [1, 4, 3, 4, 5];

try {
    arr = [4, 2];
    console.log(arr.sort());
}
catch (e) {
    console.log(e.message);
}
finally {
    console.log(arr.sort());
```

Output

Run

Throw

We use the *throw* statement, denoted by the `throw` keyword, to throw an exception. There are two ways to do this, shown below.

1. `throw value`

We can throw an exception by following the keyword `throw` with some ***value*** that we wish to use for the exception being thrown. Click *Run* below to see this in code.

```
function throwString() {
    // Generate an exception with a String value
    throw "some exception";
}

function throwFalse() {
    // Generate an exception with a boolean value of false
    throw false;
}

function throwNumber() {
    // Generate an exception with a Number value of -1
    throw -1;
}

try {
    throwString();
}
catch (e) {
    console.log(e);
}

try {
    throwFalse();
}
catch (e) {
    console.log(e);
}

try {
```

Go to Top

```
    throwNumber();
}
catch (e) {
    console.log(e);
```

Output

Run

2. `throw new Error(customError)`

We can throw an exception by following the keyword `throw` with `new Error(customError)`, where *customError* is the value we want for the *message* property of the exception being thrown. Click *Run* below to see this in code.

```
var input = "";
process.stdin.on('data', function (data) {
    input = String(data).trim();
    main();
});
/** Ignore above this line. **/

function throwMyError() {
    // Generate an exception with a value read from stdin
    throw new Error(input);
}

function main() {
    try {
        throwMyError();
    }
    catch (e) {
        console.log(e.message);
    }
```

Input

```
This is my fancy error.
```

Run

Output

- EXAMPLE

In this example, we wrote a simple program that throws an exception if the given integer argument is outside of the bounds of an array. Click *Run* below to execute the code.

Input Format

Three space-separated integers denoting indices in an array.

```
"use strict";
var input = "";
var index = 0;
process.stdin.on('data', function (data) {
    input = String(data).split(" ");
    main();
});
function readLine() { return +(input[index++]); }
/**** Ignore above this line. ****/

/*
 * This function returns the value at index pos (i.e., arr[pos]).
 *
 * If the index 'pos' is outside of the bounds of the array (i.e., v
 */
```

Go to Top

```
function getValue(arr, pos) {
    if (pos < 0) {
        throw new Error("Index Underflow: " + pos);
    }

    let len = arr.length;

    if (pos >= len) {
        throw new Error("Index Overflow: " + pos);
    }

    return arr[pos];
}

function main() {
    var index;
    const arr = [1, 2, 3, 4, 5];
    try {
        index = readLine()
        console.log(getValue(arr, index));
    }
    catch (e) {
        console.log(e.message);
    }

    try {
        index = readLine();
        console.log(getValue(arr, index));
    }
    catch (e) {
        console.log(e.message);
    }

    try {
        index = readLine();
        console.log(getValue(arr, index));
    }
    catch (e) {
        console.log(e.message);
    }
}
```

Input

```
5 2 -3
```

Run

Output

Related challenge for Error Handling

Day 3: Throw

Success Rate: 99.48% Max Score: 15 Difficulty:

Solve Challenge