Exploration — Variables, Data Types and Simple Functions

Introduction

In this exploration, we will study how to declare variables and define functions. We will also identify the different data types provided in JavaScript. We will look into the details of the primitive data types in this exploration and study objects in the next exploration. To start our study of JavaScript, let us examine the following simple JavaScript program.

Run

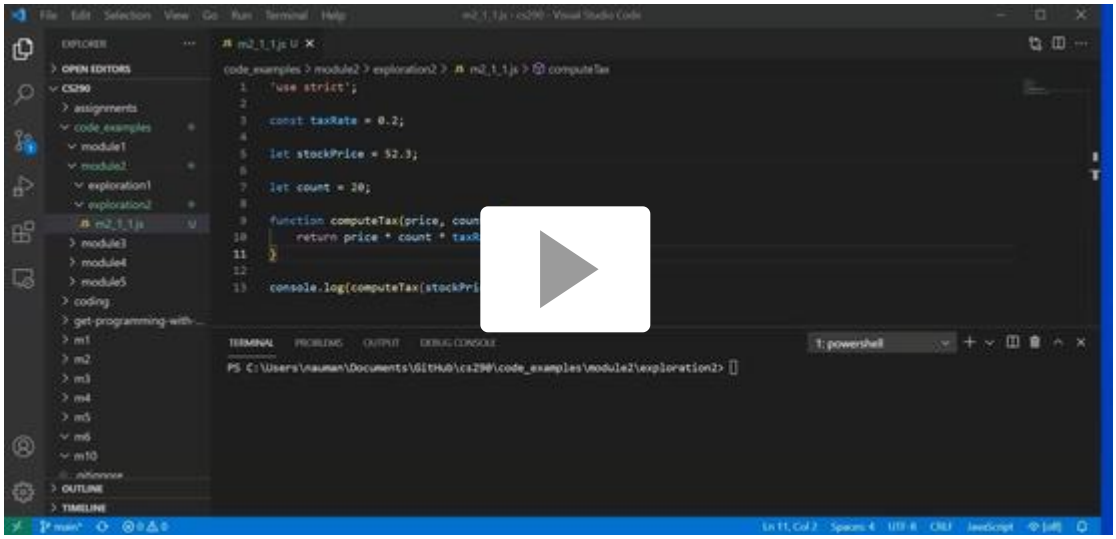
open in replit

index.js

```
1  'use strict';
2
3  const taxRate = 0.2;
4
5  let stockPrice = 52.3;
6
7  let count = 20;
8
9  function computeTax(price, count) {
```

Console Shell

In this program, we declares three variables `taxRate`, `stockPrice` and `count` and a function `computeTax`. We then call the function and print the value returned by the function to the console.



Types

A value in JavaScript has one of the following types:

- A number.
- A Boolean value, either `false` or `true`.
- A string.
- A symbol.
- The special values `undefined` and `null`.
- An object.

Values other than object type are called ***primitive types***. We will not discuss the symbol type further in this course. If interested, see Chapter 11 of the textbook to learn about the use of this type.

An Interpreted Language

We didn't need to compile our JavaScript program before running it. This is because JavaScript is an interpreted language and we do not need to compile it. The JavaScript interpreter scans the code on the fly, interprets it and executes it. This is in contrast to compiled languages, such as Java, C, C++, Rust. To run a program written in a compiled language, we first use a compiler that processes our code to create an executable file which we can then run.

Dynamic Typing

A variable in JavaScript doesn't have a type. We can store a value of any type in any variable and we don't declare the type of a variable. At runtime, the JavaScript interpreter infers the current type

of the variable based on the value that has been assigned to it. In the same program, we can assign a value of a different type to the same variable. For example, the following statements in a program are valid:

```
// Assign a number value to the variable age
let age = 10.0;

// Assign a string value to the variable age
age = 'ten';
```

Declaring Variables

In modern JavaScript, there are two forms of declaring variables, using `let` and using `const`. If the value of a variable will never change, then we should declare it using `const`, otherwise we should use `let`. Any attempt to changes the value of a variable declared as `const` will be rejected.

```
PS C:\Users\nauman> node
> const kmsInMile = 1.61;
undefined
> kmsInMile = 1.60;
Uncaught TypeError: Assignment to constant variable.
>
```

Obsolete Forms of Declaring Variables

There are two other forms of declaring variables that should avoided.

1. Declaring variables with `var`, e.g., `var count = 20;`.
   - We should avoid this due to issues with scoping that can occur when variables are declared with `var`.
   - We will discuss the reason for this briefly in the next exploration. If interested in the details, read the section on `let` in the **W3 Schools JavaScript Tutorial (https://www.w3schools.com/js/js_let.asp)** .
2. Declaring variables without a keyword, e.g., `count = 20;`.
   - This is also called "create upon first assignment."
   - Declaring variables in this form can result in a new variable being create due to misspelling.
   - For example

```
// We declare a variable count
let count = 21;

// We attempt to update the value of the variable count, but due to misspelling we end up creating a ne
w variable coun
coun = 22;
```

Fortunately, modern JavaScript can catch this issue if we use something called **strict mode**. To enable strict mode, we need to place the following line (in single or double quotes, with or without the semicolon) as the first non-comment line in a JavaScript file.

```
'use strict';
```

To use strict mode in the interactive Node REPL, we need to start Node as follows

```
node --use-strict
```

We have thus encountered the first two golden rules specified in the previous exploration.

Golden Rule 1: Declare variables with `let` or `const`, not `var`.

Golden Rule 2: Use strict mode.

Unless we clearly specify otherwise, we will also follow these, and other, golden rules in this course.

Declaring Functions

Function declarations in JavaScript use the `function` keyword followed by

1. The name of the function
2. The names of the parameters, enclosed in parentheses.
3. The body of the function in curly braces.

Unlike some other languages, we don't specify the return type of the function. If the function has a `return` statement, then the function returns immediately when that `return` statement is executed. A function which doesn't have a `return` statement or in which a `return` statement is not followed by an expression, returns the special value `undefined`.

Optional Semicolons

JavaScript doesn't require that we enter semicolons at the end of statements because it automatically inserts semicolons. However, in some cases this can be problematic! Specifically, we must not put a `return` statement by itself on a line unless we want to return the special value `undefined`. Otherwise, an unneeded semicolon will be inserted after the line containing just the keyword `return` and the function will return the value `undefined`.

For example, the following function returns the value undefined, because JavaScript will automatically insert a semicolon at the end of the line with the `return` keyword.

```
function dontDoThis(){
    // The following line is equivalent to
    //      return;
    return
        10
}
```

As opposed to this, the following function will return the value `10` since `10` is on the same line as the `return` keyword.

```
function doThisInstead() {
    return 10
}
```

For more details, see Chapter 2.2 and Chapter 3.1 of the textbook.

Naming Identifiers

Names of variables and functions should start with a letter and should contain letters and digits. JavaScript also allows `_` and `$` in identifier names. But the general JavaScript convention is to use camelCase, instead of snake_case, so we will not use `_` in identifiers. Additionally, `$` is conventionally used in libraries and tools, so we will not use that either.

Numbers

JavaScript has just one number type which is double-precision floating-point numbers. For some arithmetic operations, JavaScript automatically converts strings to numbers. The details are provided in Chapter 1.6 and 1.7 of the text book. But we must not rely on this automatic conversion, and instead we must follow Golden Rule 3:

Golden Rule 3: Know your types and avoid automatic type conversion.

We should use functions provided by JavaScript to explicitly do the conversion. Here are the relevant functions:

- **parseFloat()** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseFloat) and **parseInt()** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt) to convert a string to a number
- **toString()** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toString) to convert a number to a string
- **Math.trunc()** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/trunc) to discard the fractional part of the number
- **Math.round()** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/round) to round the number (up or down) to the nearest integer.

There is also a special number value `NaN`, which is a constant denoting that the expected number value is "not a number." For example, `0/0` evaluates to `NaN` and parsing a string that doesn't contain a number, e.g., `parseInt('')` also returns `NaN`.

Exercise

Read Chapter 1.6 & 1.7 in the book. What is the result of adding `x` and `y` without conversion in the following example? Why do we get this result?

Run

open in replit

index.js

```
1  'use strict';
2
3  // A string holding the characters 10
   let x = '10';
4
5  let y = 5.2;
6
7  // To sum them up, convert x to int
8  let res = parseInt(x) + y;
```

Console Shell

Boolean values

There are just two values of Boolean type, `true` and `false`. When we use a value of any other type in a condition, JavaScript automatically converts the value into `true` or `false`, sometimes leading to **Wat!**. We must again apply the golden rule and only use Boolean values in conditions.

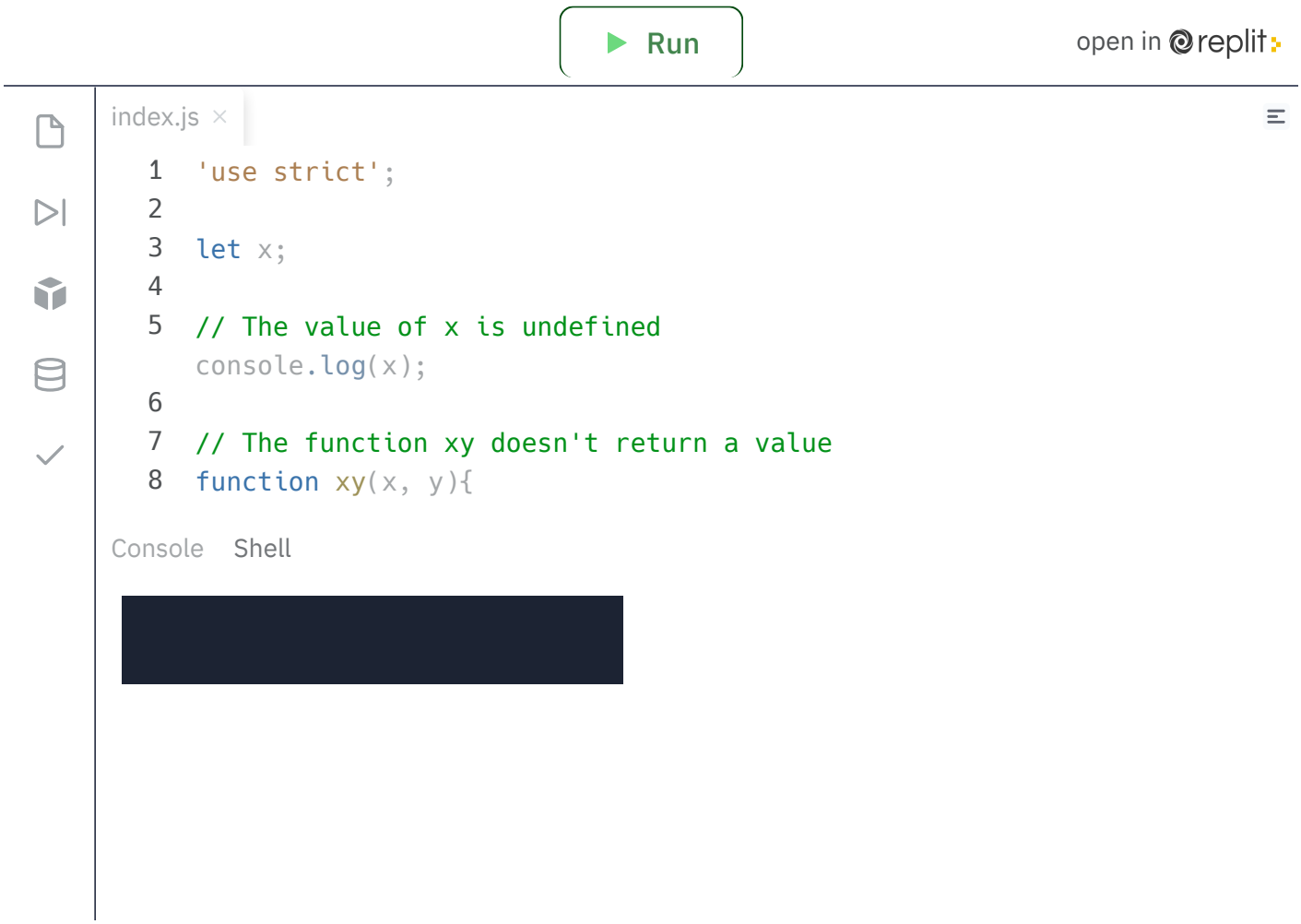
Special Values: null and undefined

There are 2 special values in JavaScript for indicating the absence of a value, `undefined` and `null`. Here are some examples, where JavaScript will automatically use `undefined`:

1. If we declare a variable but don't initialize it, JavaScript assigns it the value `undefined`.
2. If a function is called with fewer arguments than the number of parameters the function expects, the parameters for which a value has not been provided are initialized to `undefined`.
3. If a function doesn't return a value, its returned value is considered to be `undefined`.

We recommend (but don't require) that you always use `undefined` in your code and don't use `null` as a return value or assignment value in your code. Read Section 1.9 in the book for a discussion of this point.

Example



Strings

Strings in JavaScript can be enclosed in either single quote or double quote. This means that 'Hello World' and "Hello World" are both the same string. Such strings that contain only characters are also called **String Literals.**

JavaScript also supports strings that can contain expressions. Such strings are called **Template Literals**. Template literals are enclosed in **backticks**, i.e., `` ` ``, with expressions embedded within `${...}`. The value of the expression is evaluated, is converted to a string and is spliced into the template.

Example

Run

open in replit

index.js

```
1  'use strict';
2
3  let name = 'Nauman';
4
5  // ${name} is replaced by the string 'Nauman'
   let helloWorld = `Hello ${name}`;
6
   console.log(helloWorld);
7
```

Console Shell

Summary

At this point we should be able to write JavaScript programs that use primitive types and simple functions. We also explained 3 of the golden rules and identified their use in our programs.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- MDN's pages on **JavaScript (https://developer.mozilla.org/en-US/docs/Web/JavaScript)** are very informative and a good resource for learning more about the topics in this exploration.
- Values, types and operators are discussed in **Chapter 1 (https://eloquentjavascript.net/01_values.html)** of *Eloquent JavaScript*.