

# JavaScript REPL

# REPL

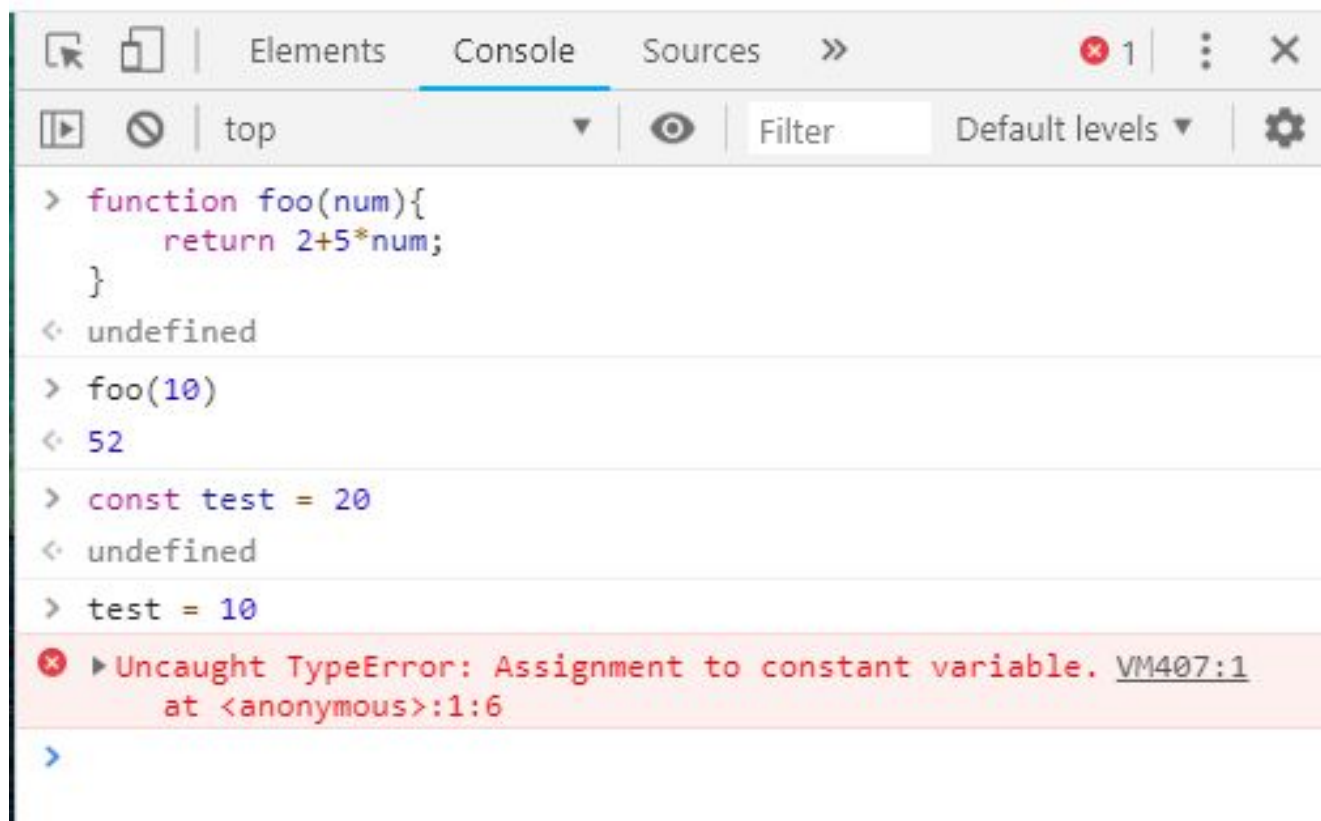
## *Read-Eval-Print Loop*

An interactive interpreter to a programming language. It does 3 things:

1. **Reads** input from the keyboard
2. **Evaluates** code passed to it
3. **Formats** and **displays** results
4. **Loops** the three previous commands until termination

# REPL : demo

Chrome devtools  
(F12)



The screenshot shows the Chrome DevTools Console with the 'Console' tab selected. The console displays a series of commands and their results in a REPL-like fashion. The commands are: a function definition, a function call, a constant declaration, and an attempt to reassign the constant. The final command results in a red error message.

```
> function foo(num){  
    return 2+5*num;  
}  
< undefined  
  
> foo(10)  
< 52  
  
> const test = 20  
< undefined  
  
> test = 10  
✖ ▶ Uncaught TypeError: Assignment to constant variable. VM407:1  
    at <anonymous>:1:6  
  
>
```

# JavaScript Scope

# Scope (programming)

**Scope** is a **region of computer program** where the variable name can be used to refer to the entity.

Scope determines the **accessibility (visibility)** of variables.

# Scope (programming) - how those lines will work?

## Scope A

var foo = "hello";

var testA = bar;

## Scope B

var bar = "world";

var testB = foo;

## Scope C

var baz = "hello world";

var testA = foo;

var testB = bar;

# Scope (programming) - answer

## Scope A

var foo = "hello";

Error!

var testA = bar;

## Scope B

var bar = "world";

Ok!

var testB = foo;

## Scope C

var baz = "hello world";

Error!

var testA = foo;  
var testB = bar;

# Functions have their own scope

**function A()**

var foo = "hello";

Error!

var testA = bar;

**function B()**

var bar = "world";

Ok!

var testB = foo;

**function C()**

var baz = "hello world";

Error!

var testA = foo;  
var testB = bar;



# Sharing a scope

**function B()**

var bar = "world";

C(); *// Calling function "C"*

var testC = baz; ?

**function C()**

var baz = "hello world";

var testB = bar; ?

# Can I share scope?

You can't, but...

...you can pass variables from one scope to another

# Sharing a scope

**function B()**

var bar = "world";

**var baz = C(bar);**

*// Calling function "C" with  
"bar" argument*

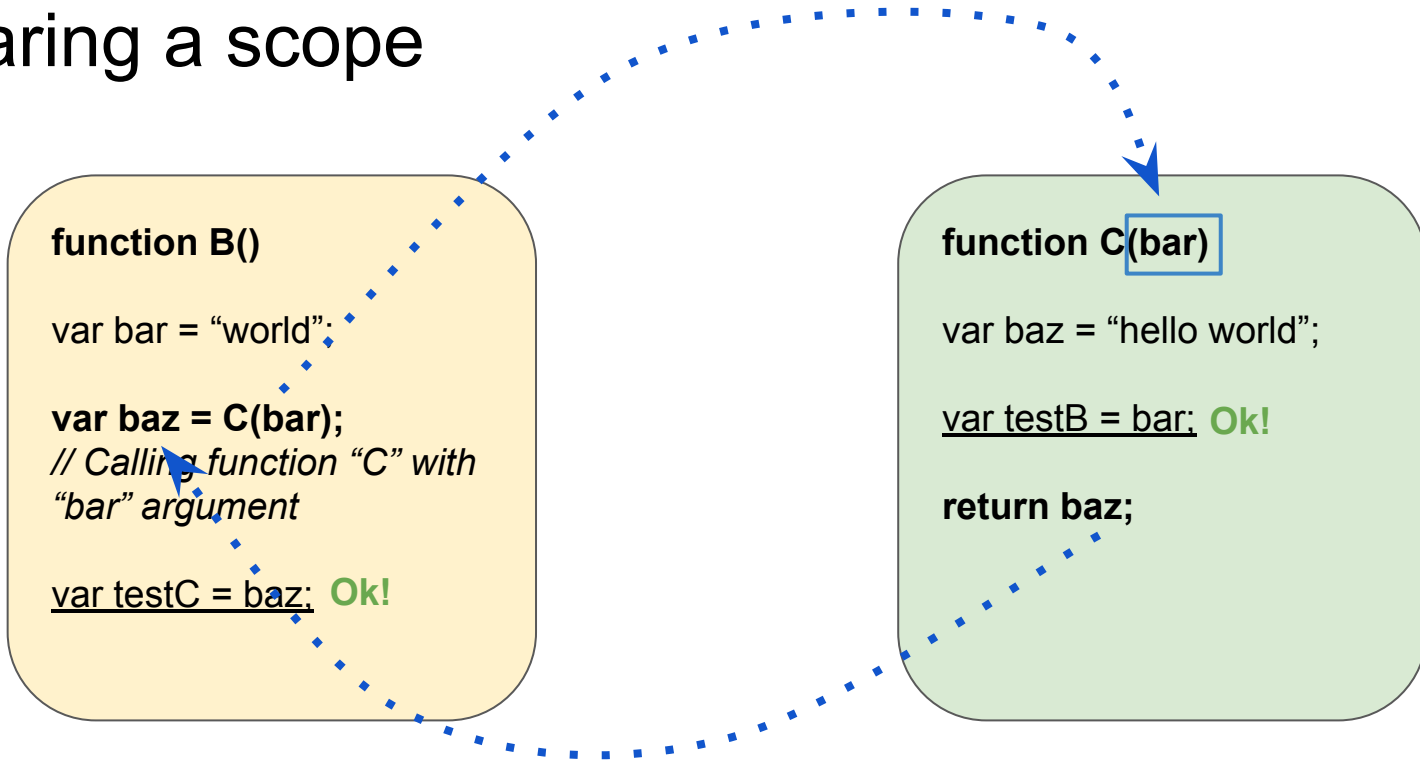
var testC = baz; **Ok!**

**function C(bar)**

var baz = "hello world";

var testB = bar; **Ok!**

**return baz;**



# Why scope is important?

*Why don't we just use global variables everywhere?*

We did before ALGOL (1958)!

In the early days amount of data processed **was small**. Data was kept in one place and could be accessed by any part of the program.

Eventually amount of data (and variables) grew. It is hard to keep track of big amount of variables.

# Why scope is important? #2

- One part of a program can't mess up the data needed by another part of the program
- Easy to determine which parts of the program are using the data correctly and which were not.
- Possible to re-use variable names.
- Easier navigation for developer.
- Easier to re-use program parts.

# JavaScript Scope

JavaScript has **function scope**: Each function creates a new scope.

Scope determines the accessibility (visibility) of variables.

Variables defined inside a function are not accessible (visible) from outside the function.

# JavaScript Scope #2

*Global scope? It has it!*

2 types of scope in JavaScript.

1. **Local scope** - scope inside function
2. **Global scope** - everything else

# JavaScript Scope

```
var a = 4;
```

1

global: a, foo

```
function foo(x) {
```

```
  var b = a * 4;
```

2

foo: x, b, bar

```
  function bar(y) {
```

```
    var c = y * b;
```

```
    return c;
```

3

bar: y, c

```
  }
```

```
  return bar(b);
```

```
}
```

```
console.log(foo(a));
```

```
// 256
```



# JavaScript Local Scope

Variables declared **within** a JavaScript function, are **Local** to the function, are located in **Local** scope of a function.

Local variables are created when a function starts, and deleted when the function is completed.

// code here can **NOT** use name

```
function myFunction() {  
  var name = "Dmitrijs";
```

// code here **CAN** use name

```
}
```

# JavaScript Global Scope

Variables declared **outside** a JavaScript function are located in **Global** scope.

All scripts and functions on a web page can access it.

What happens if 2 scripts declare **Global** variable “name” and are loaded on the same page?

```
var name = "Dmitrijs";
```

```
// code here CAN use name
```

```
function myFunction() {
```

```
    // code here also CAN use name
```

```
}
```

# JavaScript Global Scope and HTML

Global scope in JavaScript is JavaScript environment itself.

In HTML this environment is a window (browser window). All global variables belong to the current window.

1. You can't use variables from one browser window in another
2. All global variables are deleted when you close the browser window, and are assigned again when you open a new window.
3. In any JavaScript place it is possible to access global scope!

# JavaScript “window” scope

In HTML global scope is sometimes also called “window” scope.

It can be accessed by “window” global variable.

```
var name = "Dmitrijs";
```

```
// code here can use window.name
```

```
var globalName = window.name // Same as name
```

# JavaScript “window” scope #2

“window” scope is “initialized” when you open new or reload a page.

“window” scope automatically have a lot of useful stuff initialized, such as `window.document` or `window.history`

- `document` - object to access HTML DOM (Document Object Model)
- `history` - to access current browsing history
  - `back()` / `forward()` - functions to go back or forward in browsing history
  - `pushState()` - function to change current URL
- `alert()` - function to show an alert box to a user
- `location` - to access current URL, domain name, port, e.t.c.

Understanding scope

# JavaScript blocks and functions

```
function myFunction() {  
  var foo = "hello"  
}
```

```
if (true) {  
  var bar = "world"  
}
```

```
console.log(foo)  
console.log(bar)
```

?

# JavaScript blocks and functions

```
function myFunction() {  
  var foo = "hello"  
}
```

Function scope

```
if (true) {  
  var bar = "world"  
}
```

Block scope

console.log(foo)

**Error!**

console.log(bar)

**Ok!**



# JavaScript blocks and functions

JavaScript has **function scope**: Each function creates a new scope.

**Blocks** do **NOT** create a new scope! Variable defined in a block will be accessible outside it.

Blocks are:

- if
- while / for
- switch

# JavaScript var, let, const

In JavaScript variable can be defined using either `var`, `let` or `const`.

What is the difference between `var` and `let`?

Contrary to the `var` keyword, the `let` and `const` keywords support the declaration of **local scope inside block statements**.

# Local block scope example

```
if (true) {  
  // 'if' block doesn't create a scope  
  
  // name is in the global scope because of the 'var' keyword  
  var name = 'Dmitrijs';  
  // surname is in the local scope because of the 'let' keyword  
  let surname = 'Minajevs';  
  // employer is in the local scope because of the 'const' keyword  
  const employer = 'Visma Labs';  
}  
  
console.log(name); // logs 'Dmitrijs'  
console.log(surname); // Uncaught ReferenceError: likes is not defined  
console.log(employer); // Uncaught ReferenceError: skills is not defined
```

# JavaScript var, let, const

`var` variables are **LOCAL** only for functions, in **function scope**.

`let` and `const` are **LOCAL** both for functions in **function scope AND** block, in **block scope**.

# Recap

Scope determines the **accessibility** (**visibility**) of variables.

In JavaScript there are 2 types of scope: **Local** (inside functions) and **Global** (everything else)

In browser **Global** scope is accessible using **window** object and contains useful browser features.

**Blocks** in JavaScript define its own scope, however it is not considered to be **Local** UNLESS you use `let` or `const`