Types

Primitive Types

Misconception: In Javascript, everything is an object. This is false. False is not an object.

The primitive types are:

- undefined
- string
- number
- boolean
- object
  - function (callable object)
  - array
  - null
  - etc
- symbol (obscure)
- bigint

typeof

`typeof` returns a string.

It has the *expected* behavior for strings, numbers, booleans.

In [1]:
```
typeof 101
```

Out[1]:
```
'number'
```

In [2]:
```
typeof true
```

Out[2]:
```
'boolean'
```

In [3]:
```
typeof 'false'
```

Out[3]:
```
'string'
```

In [4]:
```
typeof []
```

Out[4]:
```
'object'
```

In [5]:
```
typeof typeof 69
```

Out[5]:
```
'string'
```

In [6]:
```
typeof 69n
```

Out[6]:
```
'bigint'
```

typeof null

`typeof` returns the string `'object'` for null.

In [7]:
```
typeof null
```

Out[7]:
```
'object'
```

undefined

`typeof` returns `'undefined'` regardless of whether the variable is initialized or not.

In [8]:
```
typeof uninit
```

Out[8]:
```
'undefined'
```

In [9]:
```
var undef
typeof undef
```

Out[9]:
```
'undefined'
```

functions

While functions are objects, `typeof` returns `'function'` for functions.

In [10]:
```
var function1 = function hello(){
    return 1;
}
typeof function1
```

Out[10]:
```
'function'
```

In [11]:
```
typeof (() => 0)
```

Out[11]:
```
'function'
```

In [12]:
```
let uninitializedlet
typeof uninitializedlet
```

Out[12]:
```
'undefined'
```

In [13]:
```
typeof new String("1")
```

Out[13]:
```
'object'
```

In [14]:
```
typeof new Boolean(false)
```

Out[14]:
```
'object'
```

NaN

NaN is better thought of as **invalid number** as opposed to not a number.

In [15]:
```
Number(0)
```

Out[15]:
```
0
```

In [16]:
```
Number(-1)
```

Out[16]:
```
-1
```

In [17]:
```
Number("0o105")   //octal
```

Out[17]:
```
69
```

In [18]:
```
Number("0xabc")   //hex
```

Out[18]:
```
2748
```

In [19]:
```
Number("0o8")
```

Out[19]:
```
NaN
```

In [20]:
```
Number("whatevs")
```

Out[20]:
```
NaN
```

In [21]:
```
1 - 1
```

Out[21]:
```
0
```

isNaN vs Number.isNaN

`isNaN` coerces, `Number.isNaN` does not.

In [22]:
```
var tmp = Number("0")
console.log([tmp, Number(tmp), isNaN(tmp), Number.isNaN(tmp)])
```

```
[ 0, 0, false, false ]
```

In [23]:
```
var tmp = "0"
console.log([tmp, Number(tmp), isNaN(tmp), Number.isNaN(tmp)])
```

```
[ '0', 0, false, false ]
```

In [24]:
```
var tmp = null
console.log([tmp, Number(tmp), isNaN(tmp), Number.isNaN(tmp)])
```

```
[ null, 0, false, false ]
```

In [25]:
```
var tmp = 'zero'
console.log([tmp, Number(tmp), isNaN(tmp), Number.isNaN(tmp)])
```

```
[ 'zero', NaN, true, false ]
```

Operations with NaN return NaN

In [26]:
```
Number("1") - 1
```

Out[26]:
```
0
```

In [27]:
```
Number("whatevs") + 1
```

Out[27]:
```
NaN
```

NaNs are not equal to each other

NaN is the only value in Javascript which is not equal to itself.

In [28]:
```
var tmp = Number("0")
console.log(tmp)
tmp === tmp
```

Out[28]:
```
0
true
```

In [29]:
```
var tmp = Number("zero")
console.log(tmp)
tmp === tmp
```

Out[29]:
```
NaN
false
```

In [30]:
```
var undefExample
undefExample === undefExample
```

Out[30]:
```
true
```

In [31]:
```
null === null
```

Out[31]:
```
true
```

Negative Zero

As one can expect, `-0` is equal to itself.

In [32]:
```
-0 == -0
```

Out[32]:
```
true
```

In [33]:
```
-0 === -0
```

Out[33]:
```
true
```

Using toString returns 0

In [34]:
```
var tmp = -0
tmp.toString()
```

Out[34]:
```
'0'
```

Negative zero and inequalities

In [35]:
```
-0 < 0
```

Out[35]:
```
false
```

In [36]:
```
-0 > 0
```

Out[36]:
```
false
```

In [37]:
```
-0 <= 0
```

Out[37]:
```
true
```

Negative zero equals zero

In [38]:
```
-0 == 0
```

Out[38]:
```
true
```

In [39]:
```
-0 === 0
```

Out[39]:
```
true
```

Object.is

`Object.is` can be thought of as the quadruple equals because it fixes the `0` not equal to `-0` issue.

In [40]:
```
Object.is(0,-0)
```

Out[40]:
```
false
```

In [41]:
```
Object.is(-0,-0)
```

Out[41]:
```
true
```

Math.sign

Returns `1`, `-1`, `0` or `-0`.

In [42]:
```
Math.sign(2022)
```

Out[42]:
```
1
```

In [43]:
```
Math.sign(-2022)
```

Out[43]:
```
-1
```

In [44]:
```
Math.sign(0)
```

Out[44]:
```
0
```

In [45]:
```
Math.sign(-0)
```

Out[45]:
```
-0
```

Abstract Operations

ToPrimitive

`ToPrimitive` is an abstract operation takes a hint that is either `string` or `number`.

- If the hint is `string`, then `ToPrimitive` does a `ToString` first then `ToNumber`.
- If the hint is `number`, then `ToPrimitive` does a `ToNumber` first then `ToString`.

ToString

If `x` is a string, `ToString` returns `x`.

ToNumber and Some Primitive Values

In [47]:
```
String("x")
```

Out[47]:
```
'x'
```

In [48]:
```
String(null)
```

Out[48]:
```
'null'
```

In [49]:
```
String(undefined)
```

Out[49]:
```
'undefined'
```

In [50]:
```
String(false)
```

Out[50]:
```
'false'
```

ToString and Numbers

In [51]:
```
String(-0)
```

Out[51]:
```
'0'
```

In [52]:
```
String(0)
```

Out[52]:
```
'0'
```

In [53]:
```
String(9999999999999999999999999999999999999999999999999999999999999999999999999999999
```

Out[53]:
```
'1e+69'
```

In [54]:
```
String(9999999999999999999999999999999999999999999999999999999999999999999999999999999
```

Out[54]:
```
'99999999999999999999999999999999999999999999999999999999999999999999999999'
```

In [55]:
```
var tmp = Number("tmp")
console.log(tmp)
String(tmp)
```

Out[55]:
```
NaN
'NaN'
```

ToString and Arrays

In [56]:
```
String([])
```

Out[56]:
```
''
```

In [57]:
```
String([null,undefined,""])
```

Out[57]: ',,'

In [58]:
```
String([[[]],[[],[]]])
```

Out[58]: ',,'

ToString and Objects

In [59]:
```
String({})
```

Out[59]: '[object Object]'

In [60]:
```
String({'a': 1, 'b': 2})
```

Out[60]: '[object Object]'

In [61]:
```
var tmp = { 'a':1, 'b': 2, toString(){return(this.b)} }
String(tmp)
```

Out[61]: '2'

In [62]:
```
var tmp = { 'a':1, 'b': 2, toString(){return(this.b)} }
tmp['b'] = 3
String(tmp)
```

Out[62]: '3'

In [63]:
```
var tmp = { 'a':1, 'b': 2, toString(){return(JSON.stringify(this))}}
String(tmp)
```

Out[63]: '{"a":1,"b":2}'

ToNumber

ToNumber and Some Primitive Values

In [64]:
```
Number(true)
```

Out[64]: 1

In [65]:
```
Number(false)
```

Out[65]: 0

In [66]:
```
Number(null)
```

Out[66]:
```
0
```

In [67]:
```
Number(undefined)
```

Out[67]:
```
NaN
```

ToNumber and Strings

Empty string returns `0` when coerced to a number.

In [68]:
```
Number("")
```

Out[68]:
```
0
```

In [69]:
```
Number("       \n \t \n \t")
```

Out[69]:
```
0
```

In [70]:
```
Number("0")
```

Out[70]:
```
0
```

In [71]:
```
Number("-0")
```

Out[71]:
```
-0
```

In [72]:
```
Number("--0")
```

Out[72]:
```
NaN
```

In [73]:
```
Number("  007   ")
```

Out[73]:
```
7
```

In [74]:
```
Number("0.")
```

Out[74]:
```
0
```

In [75]:
```
Number(".")
```

Out[75]:
```
NaN
```

In [76]:
```
Number("1,23")
```

Out[76]:
```
NaN
```

In [77]:
```
Number("0x45   ")
```

Out[77]:
```
69
```

In [78]:
```
Number("0x   45")
```

Out[78]:
```
NaN
```

ToNumber and Objects

For any array or object, defaults to ToString.

In [79]:
```
Number([])
```

Out[79]:
```
0
```

`Number([])` :

- `ToPrimitive([], 'number')`
  - `valueOf([])` simply returns `[]` . That's not a number.
  - `toString([])` returns `""` . That's not a number. Recurse.
- `ToPrimitive(""), 'number')`
  - `valueOf("")` returns `0` . That's a number. Return.

In [80]:
```
Number([1])
```

Out[80]:
```
1
```

In [81]:
```
Number(["0"])
```

Out[81]:
```
0
```

In [82]:
```
Number(["1","2"])
```

Out[82]:
```
NaN
```

`Number([])` :

- `ToPrimitive(["1","2"], 'number')`
  - `valueOf(["1","2"])` simply returns `["1","2"]` . That's not a number.
  - `toString(["1","2"])` returns `"[Object object]"` . That's not a number. Recurse.
- `ToPrimitive("[Object object]"), 'number')`
  - `valueOf("[Object object]")` returns `NaN` . That's a number. Return.

In [83]:
```
Number([undefined])
```

Out[83]: 0

In [84]:
```
Number({valueOf(){return(2)}})
```

Out[84]: 2

In [85]:
```
Number([[[[[]]]]])
```

Out[85]: 0

ToBoolean

Falsy values:

- "" (empty string)
- 0, -0, 0n
- null
- undefined
- NaN
- `false*

Everything else is truthy.

In [86]:
```
Boolean(0)
```

Out[86]: false

In [87]:
```
Boolean(0n)
```

Out[87]: false

In [88]:
```
Boolean([])
```

Out[88]: true

In [89]:
```
Boolean("")
```

Out[89]: false

In [90]:
```
Boolean("0")
```

Out[90]: true

In [91]:
```
Boolean("false")
```

Out[91]: true

In [92]:
```
Boolean("    0    ")
```

Out[92]:
```
true
```

In [93]:
```
Boolean(new Boolean(false))
```

Out[93]:
```
true
```

In [94]:
```
1 < 2 < 3
```

Out[94]:
```
true
```

- 1 < 2 < 3
- (1 < 2) < 3
- true < 3
- 1 < 3
- true

In [95]:
```
3 > 2 > 1
```

Out[95]:
```
false
```

- 3 > 2 > 1
- (3 > 2) > 1
- true > 1
- 1 > 1
- false

Cases of Coercion

Addition prefers string concatenation.

In [96]:
```
1 + '1'
```

Out[96]:
```
'11'
```

Multiplication prefers numbers.

In [97]:
```
5 * '    4    '
```

Out[97]:
```
20
```

In [98]:
```
'       23                ' * '    0x3    '
```

Out[98]:
```
69
```

A plus sign coerces to a number.

In [99]:
```
+'0x45'
```

Out[99]: 69

In [100…
```
1 + + '1'
```

Out[100]: 2

In [101…
```
400 + + + + + '20'
```

Out[101]: 420

An exclamation mark coerces to a boolean.

In [102…
```
!null
```

Out[102]: true

In [103…
```
![]
```

Out[103]: false

In [104…
```
!"false"
```

Out[104]: false

Boxing

Coercion from primitive to object.

Strings are not objects a priori. Adding `.length` coerces it into its object counterpart.

In [105…
```
"abcde".length
```

Out[105]: 5

Equality

Equality is **not** simply:

- `==` checks value, and
- `===` checks type.

When types match, the result of `==` and `===` are **always** the same. When types don't match, the result of `===` is **always** false.

Things to Avoid

- **Avoid** using `==` with `0` or `""` or `"  "` .
- **Avoid** using `==` with non-primitives.
- **Avoid** using `== true` and `== false` .

In [106…

```
var tmp1 = { 'a': 1 }
var tmp2 = { 'a': 1 }
tmp1 == tmp2
```

Out[106]: `false`

Double Equals aka isLooselyEqual

If the types are the same, use `===` .

If both `null` or `undefined` , return `true` .

If non-primitive, turn to primitive.

String vs number: prefers number.

Double equals checks types first.

If they match, it does a strict equality check.

Null double equals undefined.

Double equals makes sure that `null` is equal to `undefined` .

In [107…

```
null == undefined
```

Out[107]: `true`

In [108…

```
undefined == null
```

Out[108]: `true`

In [109…

```
console.log(typeof null)
console.log(typeof undefined)
null === undefined
```

```
object
undefined
```

Out[109]: false

In [110…
```
var tmp1 = { 'a': null }
var tmp2 = { }
tmp1.a == null && tmp2.a == null
```

Out[110]: true

Double equals between strings and numbers

If exactly one is a number and the other is a string, then we coerce the string to a number.

In [111…
```
42 == "42"
```

Out[111]: true

In [112…
```
"-0" == 0
```

Out[112]: true

In [113…
```
var tmp1 = "5"
var tmp2 = 5
tmp1 == tmp2
```

Out[113]: true

In [114…
```
"5" == " 5"
```

Out[114]: false

Double equals and booleans

If exactly one of the two operands are a boolean, use `ToNumber` on the boolean and return the double equals of that.

In [115…
```
true == 1
```

Out[115]: true

In [116…
```
false == ""
```

Out[116]: true

In [117…
```
NaN == false
```

Out[117]: false

Double equals and objects

If one is an object and the other is String, Number or Symbol, then apply `ToPrimitive` on the object.

Double equals only compares **primitives**.

In [118…]
```
10 == [10]
```

Out[118]: true

In [119…]
```
[""] == 0
```

Out[119]: true

Corner Cases

In [120…]
```
[] == ![]
```

Out[120]: true

- `[] == ![]`
- Evaluate `![]`.
  - The right hand side coerces into a boolean.
  - `Boolean([])` is `true`.
  - And so `![]` returns `false`.
- We now have a double equals between an object `[]` and a boolean `false`.
  - We must coerce `[]` into a primitive via `ToPrimitive`.
  - `ToPrimitive` turns `[]` into the empty string `""`.
  - Double equals prefers numbers so `""` evaluates to `0` and `false` evaluates to `0` as well.
- So, return `true`.

In [121…]
```
[] != []
```

Out[121]: true

In [122…]
```
emptyArray = []
console.log(Boolean(emptyArray))
console.log(emptyArray == true)
```

```
true
false
```

The Case for Double Equals

- Knowing the types is better.
- Double equals is **not** about comparisons of unknown types.
- Double equals is about comparisons of **known** types (optionally when coercion is helpful).

If you know the types:

- and they are **the same**, use `==` because `==` uses `===` anyway.
- and they are **different**, a triple equals will always return `false` .

Scope

Use the metaphor of colored marbles.

Scopes Example

For example:

```
1 var person = "Jose Rizal"
2
3 function whoYou() {
4    person = "J-Riz"
5    console.log(person)
6 }
7
8 whoYou()
```

- line 1 declares the variable `person`. This variable is in **Scope 1**.
- line 1 also assigns the value `"Jose Rizal"` to it.
- line 3 declares the function `whoYou`. This function is in **Scope 1**.
- lines 3-6 also gives the function definition.
- the code inside lines the brackets in lines 3-6 belong to **Scope 1.1**.
- as of this moment, the code inside lines 3-6 are not yet executed.
- line 8, part of **Scope 1** calls the function `whoYou`.
  - we enter **Scope 1.1**
  - there is a function called `whoYou` in **Scope 1** and so we follow its definition.
  - the first (non-empty) line of definition of `whoYou` is line 4.
  - line 4 references a variable / function called `person`.
    - **Scope 1.1** does not have a variable nor a function called `person`. We search it on the scope above.
    - **Scope 1** does have a variable with identifier `person`. We use that and assign its value to be `"J-Riz"`.
  - line 5 references a variable / function called `console`.
    - **Scope 1.1** does not have a variable nor a function called `console`. We search it on the scope above.
    - **Scope 1** does have a variable `console` (implicitly)
      - `console` has an attribute `log`.

In [123…

```
var person = "Jose Rizal"

function whoYou() {
    person = "J-Riz"
}

whoYou()
console.log(person)
```

```
J-Riz
```

In [124…

```
var person = "Jose Rizal"

function whoYou() {
    var person = "J-Riz"
}

whoYou()
console.log(person)
```

```
Jose Rizal
```

Function Expressions

Function expressions put their identifier on their own scope.

In [125…

```
function whereYou() {
    console.log(whereYou)
}

var saanYou = function whereKa() {
    console.log(whereKa)
}

whereYou()
saanYou()
// whereKa()  # ReferenceError
```

```
[Function: whereYou]
[Function: whereKa]
```

Anonymous function expression

In [126…

```
var saanYou = function() {
    console.log("Hello!")
}
```

Named function expression

In [127…

```
var saanYou = function whereKa() {
    console.log("Hello!")
}
```

Why named function expressions?

1. Reliable self-reference for recursion, etc.
2. More debuggable track traces.
3. More self-documenting code.

Arrow functions

Not recommended.

In [128…

```
colors = ["blue", "red", "fuchsia"]
colors.map( color => `I'm ${color} dabadee dabada...` )
```

Out[128]:

```
[
  "I'm blue dabadee dabada...",
  "I'm red dabadee dabada...",
  "I'm fuchsia dabadee dabada..."
]
```

Preferences (in order)

- (Named) Function Declarations
- Named Function Expressions
- Anonymous Function Expression

Lexical Scope vs Dynamic Scope

- **Lexical scope** is determined at compile time. (Use **ES Levels** to look at lexical scope.)
- **Dynamic scope** depends on where functions is called from.

Least Exposure Principle

In the context of scoping: Keep everything private, and only expose what is necessary.

1. Avoids name colissions.
2. Someone else can misuse it.
3. Prevents difficult refactoring.

IIFE

Immediately-invoked functions.

In [129…

```
var person = (function whoYou(){
    try {
        return "Jose Rizal"
    }
    catch {
        return "J-Riz";
    }
})()
```

Block Scoping

Curly braces don't necessarily define a scope.

They only become a scope if there is a `let` or a `const` inside it.

In [130…

```
var person = "Jose Rizal"
{
    let person = "J-Riz"
    console.log(`I'm ${person}. Sheesh!`)
}
console.log(`Ako si ${person}.`)
```

```
I'm J-Riz. Sheesh!
Ako si Jose Rizal.
```

The following thing wrapped in curly braces is not a scope.

In [131…

```
var person = "Jose Rizal"
{
    var person = "J-Riz"
    console.log(`I'm ${person}. Sheesh!`)
}
console.log(`Ako si ${person}.`)
```

```
I'm J-Riz. Sheesh!
Ako si J-Riz.
```

Explicit `let` block

If you will be using a variable shortly just after it is declared, think of putting it in a block scope and using `let` .

let vs var

If you have a variable that belongs to the entire scope of a function, use `var` .

For block scopes occuring in functions, use `let` .

Function declarations start their own scope.

In [132…

```
function outside() {

    var person = "Jose Rizal"

    function inside() {
        var person = "J-Riz"
    }

    console.log(person)
}
outside()
```

```
Jose Rizal
```

var can be useful sometimes

In [133…

```
var person = "J-Riz"

try{
    Sheesh
}
catch{
    var person = "Jose Rizal"
}

console.log(person)
```

```
Jose Rizal
```

In [1]:

```
var person = "J-Riz"

try{
    Sheesh
}
catch{
    person = "Jose Rizal"
}

console.log(person)
```

```
Jose Rizal
```

In [134…

```
var person = "J-Riz"

try{
    Sheesh
}
catch{
    let person = "Jose Rizal"
}

console.log(person)
```

```
J-Riz
```

var can be used several times in a scope but not let

In [135…

```
var tmp1 = "a"
var tmp1 = "b"
tmp1
```

Out[135]:

```
'b'
```

In [136…

```
let tmp1 = "a"
let tmp1 = "b"
tmp1 //error
```

```
evalmachine.<anonymous>:2
let tmp1 = "b"
    ^

SyntaxError: Identifier 'tmp1' has already been declared
    at new Script (node:vm:100:7)
    at createScript (node:vm:257:10)
    at Object.runInThisContext (node:vm:305:10)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:527:28)
    at emit (node:internal/child_process:936:14)
    at process.processTicksAndRejections (node:internal/process/task_queu
es:83:21)
```

const

Only useful for **primitives**.

In [137…

```
var person1 = "Jose Rizal"
person1 = "J-Riz"
```

Out[137]:

```
'J-Riz'
```

In [138…

```
const person2 = "Jose Rizal"
person2 = "J-Riz" // ERROR
```

```
evalmachine.<anonymous>:2
person2 = "J-Riz" // ERROR
        ^

TypeError: Assignment to constant variable.
    at evalmachine.<anonymous>:2:9
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:305:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:527:28)
    at emit (node:internal/child_process:936:14)
    at process.processTicksAndRejections (node:internal/process/task_queu
es:83:21)
```

In [139…

```
const person3 = ["J-Riz"]
person3.push("Andy B")
```

Out[139]:

```
2
```

Hoisting

Hoisting is not a thing in the ECMAScript documentation (until recently).

It is a short-hand for the "two passes" that JavaScript does.

Function **declarations** hoist, so you can put your main code on top and define the functions below.

In [140…

```
totallyNewPerson = "Jose Rizal"
totallyNewGreeting = "Mabuhay"
var totallyNewPerson
var totallyNewGreeting
```

Out[140]: 'Mabuhay'

In [141…

```
var person = "Jose Rizal"

alterEgo()

function alterEgo() {
    console.log(person);
    var person = "J-Riz"
}
```

undefined

`let` doesn't "hoist"?

- When `var` creates the plan for the scopes, it gives the variable declared an initial value of `undefined` .
- When `let` and `const` creates the plan for the scopes, it does not give the variable an initial value (i.e. the variable is uninitialized).

In [142…

```
var person = "Jose Rizal"
{
    console.log(person)   //TDZ error
    let person = "J-Riz"
}
```

```
evalmachine.<anonymous>:3
    console.log(person)   //TDZ error
                ^

ReferenceError: Cannot access 'person' before initialization
    at evalmachine.<anonymous>:3:17
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:305:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:527:28)
    at emit (node:internal/child_process:936:14)
    at process.processTicksAndRejections (node:internal/process/task_queu
es:83:21)
```

Why does TDZ exist?

It is for `const`.

If `const` were assigned the primitive `undefined` first, and then finally assigned to something else, then it encountered two different values during its lifetime.

Hence, there had to be a notion of *uninitialized*.

Closure

Closure is when a function **remembers** its lexical scope even when the function is executed outside that lexical scope.

That is a practical definition. Academic definition of closure apparently is not very useful practically.

In [143…

```
function waitAndSpeak(statement) {
    setTimeout(function waitASec(){
        console.log(statement)
    }, 1)
}

waitAndSpeak("Thank you for waiting!")
```

```
Thank you for waiting!
```

In the function below, the function `printGreeting` is closed over the `greeting`.

In [144…

```
function speak(greeting) {
    return function printGreeting() {
        console.log(greeting)
    }
}

var formalGreeting = ask("Kamusta ka?")
var familiarGreeting = ask("Yo! How are you? Sheesh!")

formalGreeting()
familiarGreeting()
```

```
evalmachine.<anonymous>:7
var formalGreeting = ask("Kamusta ka?")
                     ^

ReferenceError: ask is not defined
    at evalmachine.<anonymous>:7:22
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:305:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:527:28)
    at emit (node:internal/child_process:936:14)
    at process.processTicksAndRejections (node:internal/process/task_queu
es:83:21)
```

Closure does not capture a value.

You close over a variable, not a value.

In [145…

```
var person = "Jose Rizal"

var whoYou = function() {
    console.log(person)
}

person = "J-Riz"
whoYou()
```

```
J-Riz
```

for-var and for-let difference

In [146…

```
for(var i=1; i<=5; i++){
    setTimeout(function() {
        console.log(`i: ${i}`);
    }, i*1);
}
i
```

Out[146]: 6

```
i: 6
i: 6
i: 6
i: 6
i: 6
```

In [147…

```
for(let i=1; i<=5; i++){
    setTimeout(function() {
        console.log(`i: ${i}`);
    }, i*1);
}
i
```

Out[147]: 6

```
i: 1
i: 2
i: 3
i: 4
i: 5
```

Modules

A namespace is not a module.

In [148…

```
var person = {
    name: "Jose Rizal",
    greet(statement) {
        console.log(`Ako si ${this.name}. ${statement}`)
    }
}

person.greet("Mabuhay!")
```

```
Ako si Jose Rizal. Mabuhay!
```

Modules encapsulate data and behavior (methods) together.

The state (data) of a module is held by its methods via closure.

Module Factory

In [149…

```
function personModule(name) {
    var pubAPI = {greet,}
    return(pubAPI)

    function greet(greeting) {
        console.log(`${name}: ${greeting}`)
    }
}

var jose = personModule("Jose Rizal")
var jriz = personModule("J-Riz")

jriz.greet("Sheesh!")
jose.greet("Hey shawty!")
```

```
J-Riz: Sheesh!
Jose Rizal: Hey shawty!
```

ES6 Modules

- ES6 modules are file-based. You can't have more than one ES6 module in the same file.
- Everything in a module is private.
- Use export to make it public.
- Modules, once imported, are referenced to one object.
- You have to make an explicit **Module Factory** in modules.

Use the `.mjs` extension for the ES6 module pattern

`bayani.mjs` :

```
var person = "Jose Rizal"

export default function greet(greeting) {
     console.log(person, greeting)
};
```

Importing modules

default import

```
import greet from "bayani.mjs"
greet("Mabuhay!")
```

namespace import

```
import * as bayani from "bayani.mjs"
bayani.greet("Mabuhay!")
```

Objects

`this` keyword

A function's `this` references the execution context for that call, determined entirely by **how the function was called**.

In other words, you cannot look at a function definition and be sure what the `this` keyword means.

A `this` -aware function can thus have a different context each time it's called, which makes it more flexible and reusable.

If you want things to be more predictable, why not write a module instead?

four different ways to call a function

- Implicit binding
- Explicit binding
- `new` keyword
- Default binding

implicit binding

`this` is the object that calls it.

This is how other languages treat this.

In [150…

```
var bayani = {
    person: "Jose Rizal",
    greet(greeting) {
        console.log(`${this.person}, ${greeting}`)
    }
}

bayani.greet("sheesh!") // this is the bayani object
```

```
Jose Rizal, sheesh!
```

dynamic binding

Reusing a function to invoke in a different context.

In [151…

```
function greet(greeting) {
    console.log(`${this.person}, ${greeting}`)
}

var bayani1 = {
    person: "J-Riz",
    greet: greet
}

var bayani2 = {
    person: "Andy B",
    greet: greet
}

bayani1.greet("sheesh!")
bayani2.greet("yo!")
```

```
J-Riz, sheesh!
Andy B, yo!
```

explicit binding (using `.call`)

With implicit binding, `this` is less predictable.

If you need to explicitly say what `this` is, use explicit binding.

But, use sparingly. Otherwise, you don't need `this`.

In [152…

```
function greet(greeting) {
    console.log(`${this.person}, ${greeting}`)
}

var bayani1 = {
    person: "J-Riz",
}

var bayani2 = {
    person: "Andy B",
}

greet.call(bayani1, "sheesh!")
greet.call(bayani2, "yo!")
greet.call({person: "MaJoJa"}, "hello mga bro!")
```

```
J-Riz, sheesh!
Andy B, yo!
MaJoJa, hello mga bro!
```

hard binding

Removes flexibility and adds more predictability.

This is an example of explicit binding.

In [153…

```
var bayani = {
    person: "Jose Rizal",
    greet(greeting) {
        console.log(`${this.person}, ${greeting}`)
    }
}

setTimeout( bayani.greet.bind(bayani), 10, "sheesh")
console.log();
```

In [154…

```
var bayani = {
    person: "Jose Rizal",
    greet(greeting) {
        console.log(`${this.person}, ${greeting}`)
    }
}

setTimeout( bayani.greet, 10, "sheesh")
console.log();
```

```
Jose Rizal, sheesh
```

In [155…

```
function greet(greeting) {
    console.log(`${this.person}, ${greeting}`)
}

function otherGreet() {
    var customContext = {
        person: "Andy B"
    }
    greet.call(customContext, "sheesh!")
}

otherGreet()
```

```
undefined, sheesh
Andy B, sheesh!
```

new keyword

In [156…

```
function greet(greeting) {
    console.log(`${greeting}`)
}

var newEmptyObject = greet("Sheesh!")
```

```
Sheesh!
```

what are the four things the `new` keyword does

1. Create a new empty object.
2. Link* that object to a new object.
3. Call the function with `this` set to the new object.
4. If function does not return an object, return `this` .

default binding

In non-strict mode, it uses the global object for `this` .

In [157…

```
var person = "J-Riz"

function greet(greeting) {
    console.log(`${this.person}, ${greeting}`)
}

greet("Sheesh!")
```

```
J-Riz, Sheesh!
```

It is a terrible idea to invoke a `this` keyword using the default binding.

In strict mode, doing this is not allowed.

In [158…

```
var person = "J-Riz"

function strictGreet(question) {
    "use strict"
    console.log(`${this.person}, ${greeting}`)
}

strictGreet("Sheesh!")
```

```
evalmachine.<anonymous>:5
    console.log(`${this.person}, ${greeting}`)
                        ^

TypeError: Cannot read properties of undefined (reading 'person')
    at strictGreet (evalmachine.<anonymous>:5:25)
    at evalmachine.<anonymous>:8:1
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:305:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:527:28)
    at emit (node:internal/child_process:936:14)
    at process.processTicksAndRejections (node:internal/process/task_queu
es:83:21)
```

order of precedence for understanding `this`

1. Is the function called by `new`? If so, the newly created object will be bound to `this`.
2. Is the function called with `call` or `apply` (`bind` uses apply). Then the context in `call` will be used.
3. Is the function called on a context object, such as `bayani.greet`?
4. Use default binding except in strict mode.

arrow functions

`this` is resolved lexically with arrow functions.

Recommendation: Use arrow functions only if you need the lexical `this`.

An arrow function is not a `this`-bound function to its parent function.

In [159…

```
var person = "J-Riz"

var bayani = {
    person: "Andy B",
    greet(greeting) {
        setTimeout( () => console.log(`${this.person}, ${greeting}`), 100
    }
}

bayani.greet("sheesh!")
```

Recall that the curly brace defining `bayani` below is **not** a scope.

So, `this.person` resolves to the global scope.

In [160…

```
var person = "J-Riz"

var bayani = {
    person: "Andy B",
    greet: (greeting) => {
        console.log(`${this.person}, ${greeting}`)
    }
}

bayani.greet("sheesh!")
```

```
J-Riz, sheesh!
```

In [161…

```
var person = "J-Riz"

var bayani = {
    person: "Andy B",
    greet: (greeting) => {
        console.log(`${this.person}, ${greeting}`)
    }
}

bayani.greet.call(bayani, "sheesh!")
```

```
J-Riz, sheesh!
```

`class` keyword

In [162…

```
class Greeter {
    constructor(person) {
        this.person = person
    }
    greet(greeting) {
        console.log(`I'm ${this.person}. ${greeting}`)
    }
}
```

In [163…

```
var greeter1 = new Greeter("J-Riz")
var greeter2 = new Greeter("Andy B")
greeter1.greet("Sheesh!")
greeter2.greet("Yo!")
```

```
I'm J-Riz. Sheesh!
I'm Andy B. Yo!
```

`extends` on classes

In [164…

```
class formalGreeter extends Greeter {
    formallyGreet(greeting) {
        console.log(`${greeting} Ako si ${this.person}.`)
    }
}
```

In [165…

```
var greeter3 = new formalGreeter("Jose Rizal")
greeter3.formallyGreet("Magandang araw!")
greeter3.greet("Magandang araw!")
```

```
Magandang araw! Ako si Jose Rizal.
I'm Jose Rizal. Magandang araw!
Andy B, sheesh!
```

`super` keyword

In [166…

```
class shoutingGreeter extends Greeter {
    greet(greeting) {
        super.greet(greeting.toUpperCase())
    }
}
```

In [168…

```
var greeter4 = new shoutingGreeter("J-Riz")
greeter4.greet("Sheesh!")
```

```
I'm J-Riz. SHEESH!
```

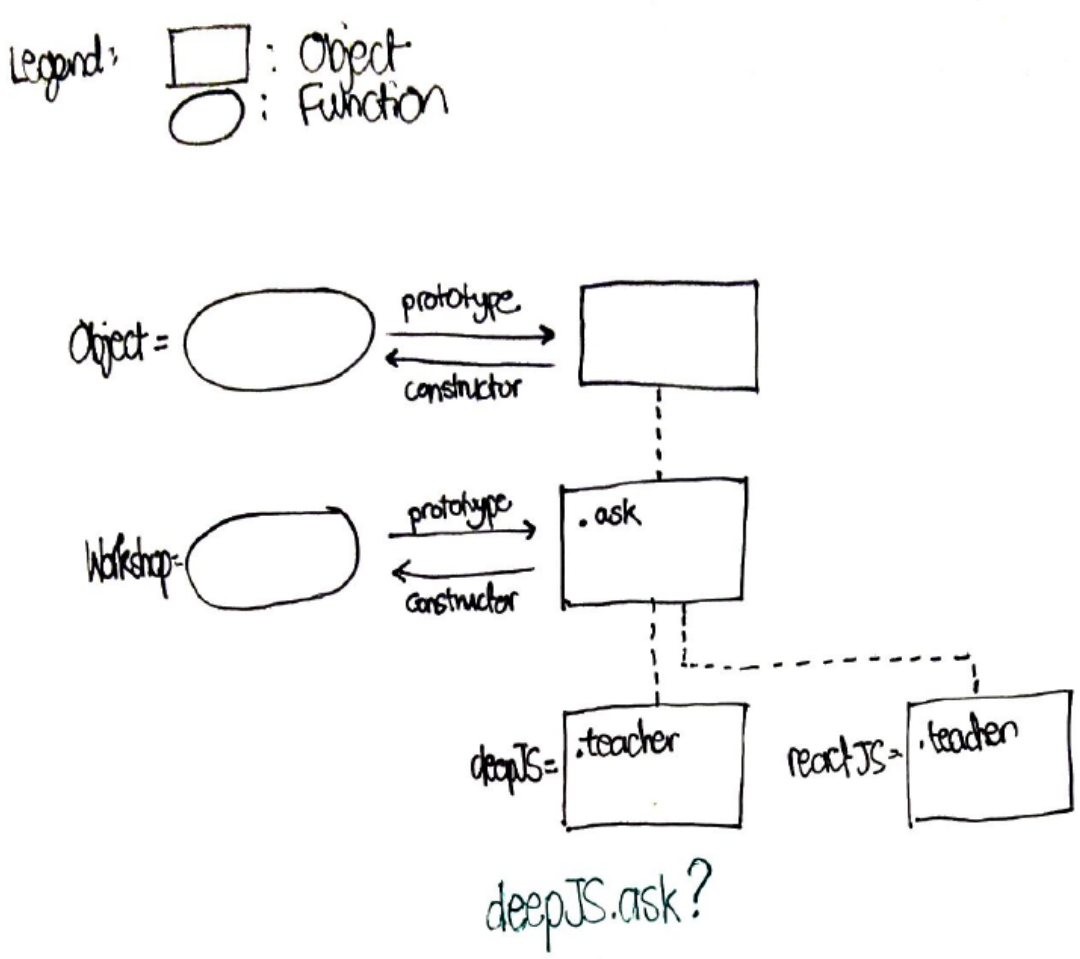
Protoypes

Objects are build by *constructor calls* (via `new`).

The usual analogy for classes is blueprints and buildings.

The relationship between the *blueprint* (class) and the *building* (instance) existed during the time it was being instantiated.

A *constructor call* makes an object ~based on~ linked to its own prototype.



In [1]:

```
function Greeter(person) {
    this.person = person
}

Greeter.prototype.greet = function(greeting) {
    console.log(`${greeting} I am ${this.person}.`)
}

var person = new Greeter("J-Riz")

person.greet("Sheesh!")
```

```
Sheesh! I am J-Riz.
```

In [10]:

```
function Greeter(person) {
    this.person = person
}

Greeter.prototype.greet = function greet(greeting) {
    console.log(`${greeting} I am ${this.person}.`)
}

var person = new Greeter("J-Riz")

console.log( person.constructor === Greeter )
console.log( person.__proto__ === Greeter.prototype )
console.log( Object.getPrototypeOf( person ) === Greeter.prototype )

Greeter.prototype
```

Out[10]:

```
true
true
true
{ greet: [Function: greet] }
```

Shadowing

In [20]:

```
function Greeter(person) {
    this.person = person
}

Greeter.prototype.greet = function greetOnGreeterPrototype(greeting) {
    console.log(`${greeting} I am ${this.person}.`)
}

var person = new Greeter("J-Riz")

person.greet = function(greeting) {
    this.greet(greeting.toUpperCase())
}

person.greet("Woah!") //Infinite recursion.
```

```
evalmachine.<anonymous>:12
    this.greet(greeting.toUpperCase())
         ^

RangeError: Maximum call stack size exceeded
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
    at person.greet (evalmachine.<anonymous>:12:10)
```

This works, but is ugly. You can't do shadowing properly without classes.

In [25]:

```
function Greeter(person) {
    this.person = person
}

Greeter.prototype.greet = function greetOnGreeterPrototype(greeting) {
    console.log(`${greeting} I am ${this.person}.`)
}

var person = new Greeter("J-Riz")

console.log(person.__proto__.greet)

person.greet = function(greeting) {
    this.__proto__.greet.call(this, greeting.toUpperCase())
}

person.greet("Woah!")
```

```
[Function: greetOnGreeterPrototype]
WOAH! I am J-Riz.
```

Object.create

1. Creates a new empty object.
2. Links the object to another object.

What happens when `person.louder("Sheesh!")` on the code below?

- There is no `louder` method on `person`. Go one level up the prototype chain.
- There is a `louder` method on `AnotherGreeting`. Use this.
- In this `louder` method, `this.greet` is called.
- `this` in this context is still `person`.
- There is no `greet` method on `person`. Go one level up the prototype chain.
- There is no `greet` method on `AnotherGreeting`. Go one level up the prototype chain. Note that `Greeting` is the next level, thanks to `Object.create`.
- There is a `greet` method on `Greeting`. Use this.

In [29]:

```
function Greeting(person) {
    this.person = person
}

Greeting.prototype.greet = function greetOnGreeterPrototype(greeting) {
    console.log(`${greeting} I am ${this.person}.`)
}

function AnotherGreeting(person) {
    Greeting.call(this, person)
}

AnotherGreeting.prototype = Object.create(Greeting.prototype)
AnotherGreeting.prototype.louder = function(greeting) {
    this.greet(greeting.toUpperCase());
}

var person = new AnotherGreeting("J-Riz")

person.louder("Sheesh!")
```

```
SHEESH! I am J-Riz.
```

JavaScript "inheritance" is *behavior delegation*

A prototypal system can implement a class system.

OLOO: Objects Linked to Other Objects

You can create an object without any class in Javascript (and Lua).

In [40]:

```
var Greeter = {
    setPerson(person) {
        this.person = person
        console.log(`${this.person} in da haus!`)
    },
    greet(greeting) {
        console.log(`${greeting} I am ${this.person}.`)
    }
};

var AnotherGreeter = Object.assign(
    Object.create(Greeter),
    {
        louder(greeting) {
            this.greet(greeting.toUpperCase())
        }
    }
);

var omniGreeter = Object.create(AnotherGreeter)

omniGreeter.setPerson("J-Riz")
omniGreeter.greet("Yo shawty!")
omniGreeter.louder("Sheesh!")
omniGreeter.setPerson("Andy B")
omniGreeter.louder("Wazzup!")
```

```
J-Riz in da haus!
Yo shawty! I am J-Riz.
SHEESH! I am J-Riz.
Andy B in da haus!
WAZZUP! I am Andy B.
```

`Object.create` polyfill

```
if(!Object.create) {
    Object.create = function(o) {
        function F() {}
        F.prototype = o
        return new F();
    }
}
```