JavaScript

STUDIA PODYPLOMOWE
POLITECHNIKA BIAŁOSTOCKA


#5

Internal type coercion

coercion.js

```
1  console.log(20 * 0); // 0
2  console.log(20 * -0); // -0
3  console.log(20 * '0'); // 0
4  console.log(20 * '-0'); // -0
5  console.log(20 * `${0}`); // 0
6  console.log(20 * `${-0}`); // -0
7
8  console.log(10 - null); // 10
9  console.log(10 - undefined); // NaN
10 console.log(10 - [null]); // 10
11 console.log(10 - [undefined]); // 10
12
13 console.log(1 < 2 < 3); // true
14 console.log(3 > 2 > 1); // false
15
```

toString

PRIMITIVES

1	'1'
1.23	'1.23'
Infinity	'Infinity'
NaN	'NaN'
0	'0'
-0	'0'
true	'true'
false	'false'
null	'null'
undefined	'undefined'

7.1.12 ToString (*argument*)

The abstract operation ToString converts *argument* to a value of type String according to Table 11:

Table 11: ToString Conversions

Argument Type	Result
Undefined	Return **"undefined"**.
Null	Return **"null"**.
Boolean	If *argument* is **true**, return **"true"**. If *argument* is **false**, return **"false"**.
Number	Return NumberToString(*argument*).
String	Return *argument*.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ? ToPrimitive(*argument*, hint String). 2. Return ? ToString(*primValue*).

toNumber

PRIMITIVES

'hello'	NaN
'1'	1
'1.23'	1.23
''	0
'0'	0
'-0'	-0
'0001'	1
null	0
undefined	NaN
true	1
false	0

7.1.3 ToNumber (*argument*)

The abstract operation ToNumber converts *argument* to a value of type Number according to Table 10:

Table 10: ToNumber Conversions

Argument Type	Result
Undefined	Return **NaN**.
Null	Return **+0**.
Boolean	If *argument* is **true**, return 1. If *argument* is **false**, return **+0**.
Number	Return *argument* (no conversion).
String	See grammar and conversion algorithm below.
Symbol	Throw a **TypeError** exception.
Object	Apply the following steps: 1. Let *primValue* be ? ToPrimitive(*argument*, hint Number). 2. Return ? ToNumber(*primValue*).

To Primitive

- Internal method of JS
- Accepts *hint* as parameter
- It can be either *number* or *string*

7.1.1 ToPrimitive (*input* [, *PreferredType*])

The abstract operation ToPrimitive takes an *input* argument and an opti
Object type. If an object is capable of converting to more than one primi
following algorithm:

1. Assert: *input* is an ECMAScript language value.
2. If Type(*input*) is Object, then
    a. If *PreferredType* is not present, let *hint* be **"default"**.
    b. Else if *PreferredType* is hint String, let *hint* be **"string"**.
    c. Else *PreferredType* is hint Number, let *hint* be **"number"**.
    d. Let *exoticToPrim* be ? GetMethod(*input*, @@toPrimitive).
    e. If *exoticToPrim* is not **undefined**, then
        i. Let *result* be ? Call(*exoticToPrim*, *input*, « *hint* »).
        ii. If Type(*result*) is not Object, return *result*.
        iii. Throw a **TypeError** exception.
    f. If *hint* is **"default"**, set *hint* to **"number"**.
    g. Return ? OrdinaryToPrimitive(*input*, *hint*).
3. Return *input*.

toPrimitive

(*string*)

- call .toString()
- call .valueOf()

(*number*)

- call .valueOf()
- call .toString()

toPrimitive (string)

OBJECTS

{}	'[object Object]'
{toString() {return 'hello'}}	'hello'
[1,2,'hello']	'1,2,hello'
[]	''
[null, undefined]	','

toPrimitive (number)

{}	NaN
{valueOf() {return 1}}	1
['1','2']	NaN
[1,2]	NaN
['1']	1
[1]	1
['']	0
['0']	0
[null]	0
[undefined]	0

toBoolean

Falsy

- false
- NaN
- null
- undefined
- 0
- -0
- 0n
- ”

Truthy

- Everything else

7.1.2 ToBoolean (*argument*)

The abstract operation ToBoolean converts *argument* to a value of type Boolean according to Table 9:

Table 9: ToBoolean Conversions

Argument Type	Result
Undefined	Return **false**.
Null	Return **false**.
Boolean	Return *argument*.
Number	If *argument* is **+0**, **-0**, or **NaN**, return **false**; otherwise return **true**.
String	If *argument* is the empty String (its length is zero), return **false**; otherwise return **true**.
Symbol	Return **true**.
Object	Return **true**.

Quiz

coercion.js

```
1  console.log(20 * 0);
2  console.log(20 * -0);
3  console.log(20 * '0');
4  console.log(20 * '-0');
5  console.log(20 * `${0}`);
6  console.log(20 * `${-0}`);
7
8  console.log(10 - null);
9  console.log(10 - undefined);
10 console.log(10 - [null]);
11 console.log(10 - [undefined]);
12
13 console.log(1 < 2 < 3);
14 console.log(3 > 2 > 1);
15
16 console.log(!!`${undefined}`);
17 console.log(!!`${null}`);
18
```

== (loose equality)

7.2.14 Abstract Equality Comparison

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is the same as Type(*y*), then
    a. Return the result of performing Strict Equality Comparison *x* === *y*.
2. If *x* is **null** and *y* is **undefined**, return **true**.
3. If *x* is **undefined** and *y* is **null**, return **true**.
4. If Type(*x*) is Number and Type(*y*) is String, return the result of the comparison *x* == ! ToNumber(*y*).
5. If Type(*x*) is String and Type(*y*) is Number, return the result of the comparison ! ToNumber(*x*) == *y*.
6. If Type(*x*) is Boolean, return the result of the comparison ! ToNumber(*x*) == *y*.
7. If Type(*y*) is Boolean, return the result of the comparison *x* == ! ToNumber(*y*).
8. If Type(*x*) is either String, Number, or Symbol and Type(*y*) is Object, return the result of the comparison *x* == ToPrimitive(*y*).
9. If Type(*x*) is Object and Type(*y*) is either String, Number, or Symbol, return the result of the comparison ToPrimitive(*x*) == *y*.
10. Return **false**.

=== (strict equality)

7.2.15 Strict Equality Comparison

The comparison *x* === *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is different from Type(*y*), return **false**.
2. If Type(*x*) is Number, then
    a. If *x* is **NaN**, return **false**.
    b. If *y* is **NaN**, return **false**.
    c. If *x* is the same Number value as *y*, return **true**.
    d. If *x* is **+0** and *y* is **-0**, return **true**.
    e. If *x* is **-0** and *y* is **+0**, return **true**.
    f. Return **false**.
3. Return SameValueNonNumber(*x*, *y*).

Iterators

Definitions

- Iteration - process of accessing data one by one from a collection
- Iterator - an interface that allows this process
- Iterable - a collection of data that can be iterated

iterator.js

```
1  const array = [1, 2, 3];
2
3  for (let i = 0; i < array.length; i++) {
4      console.log(array[i])
5  }
6
7  // 1
8  // 2
9  // 3
```

iterator.js

```
function getIterator(array) {
    let i = 0;

    function next() {
        const returnObject = { value: array[i], done: i >= array.length };
        i++;
        return returnObject
    }

    return { next }
}

const testData = [1,2,3];

const iterator = getIterator(testData);

console.log(iterator.next()) // { value: 1, done: false }
console.log(iterator.next()) // { value: 2, done: false }
console.log(iterator.next()) // { value: 3, done: false }
console.log(iterator.next()) // { value: undefined, done: true }

```

iterator.js

```
const iterbaleObject = {
    0: 1,
    1: 2,
    2: 3,
    length: 3,
    [Symbol.iterator]: function () {
        let i = 0;

        const next = () => {
            const returnObject = { value: this[i], done: i >= this.length };
            i++;
            return returnObject;
        }

        return { next };
    }
}

for (const test of iterbaleObject) {
    console.log(test);
}
```

Generator

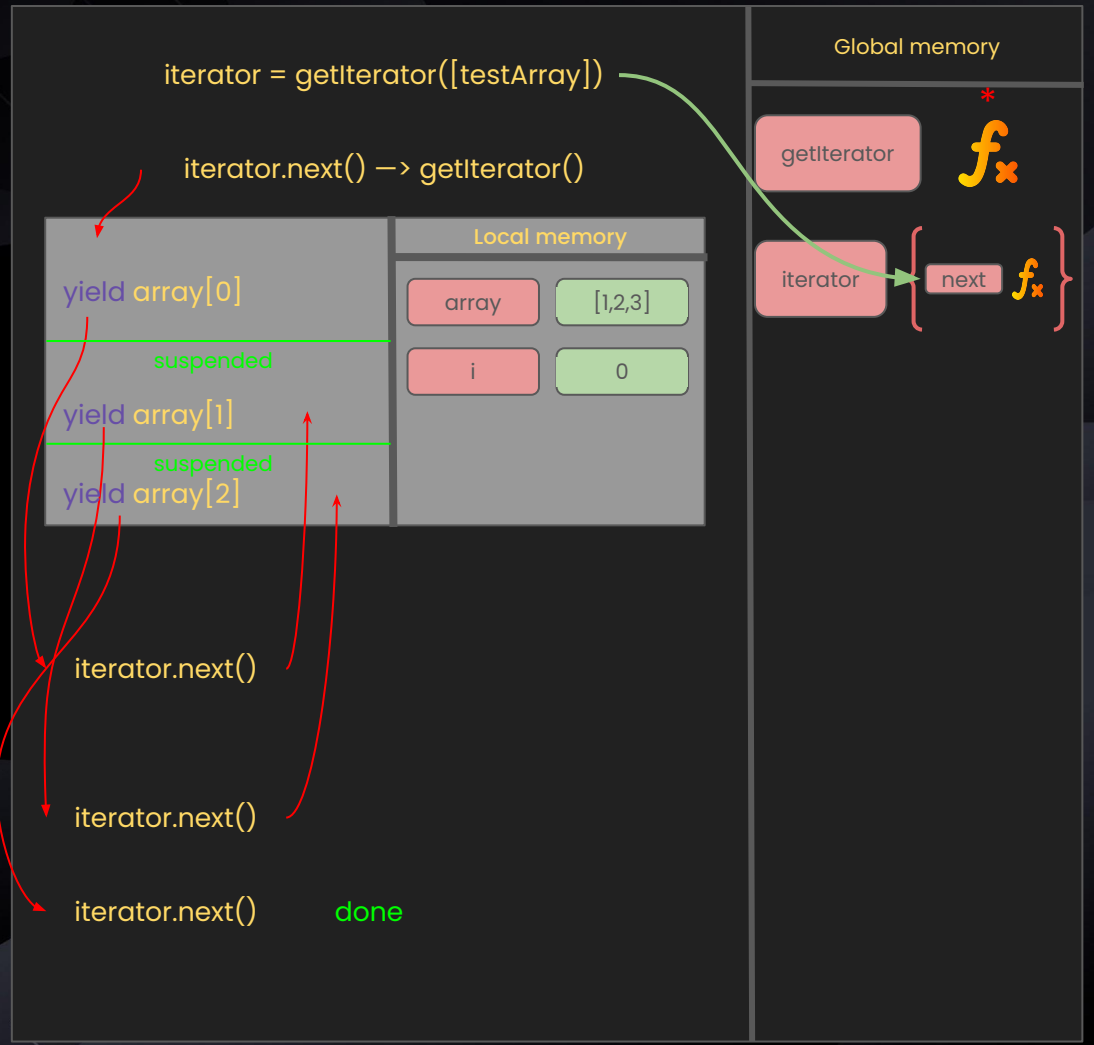
- A special type of function that creates iterator
- But it has some super powers!

generator.js

```
function* getIterator(array) {
    for (let i = 0; i < array.length; i++) {
        yield array[i];
    }

    return;
}

const testArray = [1, 2, 3];
const iterator = getIterator(testArray);

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true }

```

generator.js

```
1  function* getIterator(array) {
2      for (let i = 0; i < array.length; i++) {
3          yield array[i];
4      }
5
6      return;
7  }
8
9  const testArray = [1, 2, 3];
10 const iterator = getIterator(testArray);
11
12 console.log(iterator.next()); // { value: 1, done: false }
13 console.log(iterator.next()); // { value: 2, done: false }
14 console.log(iterator.next()); // { value: 3, done: false }
15 console.log(iterator.next()); // { value: undefined, done: true }
16
```



generator.js

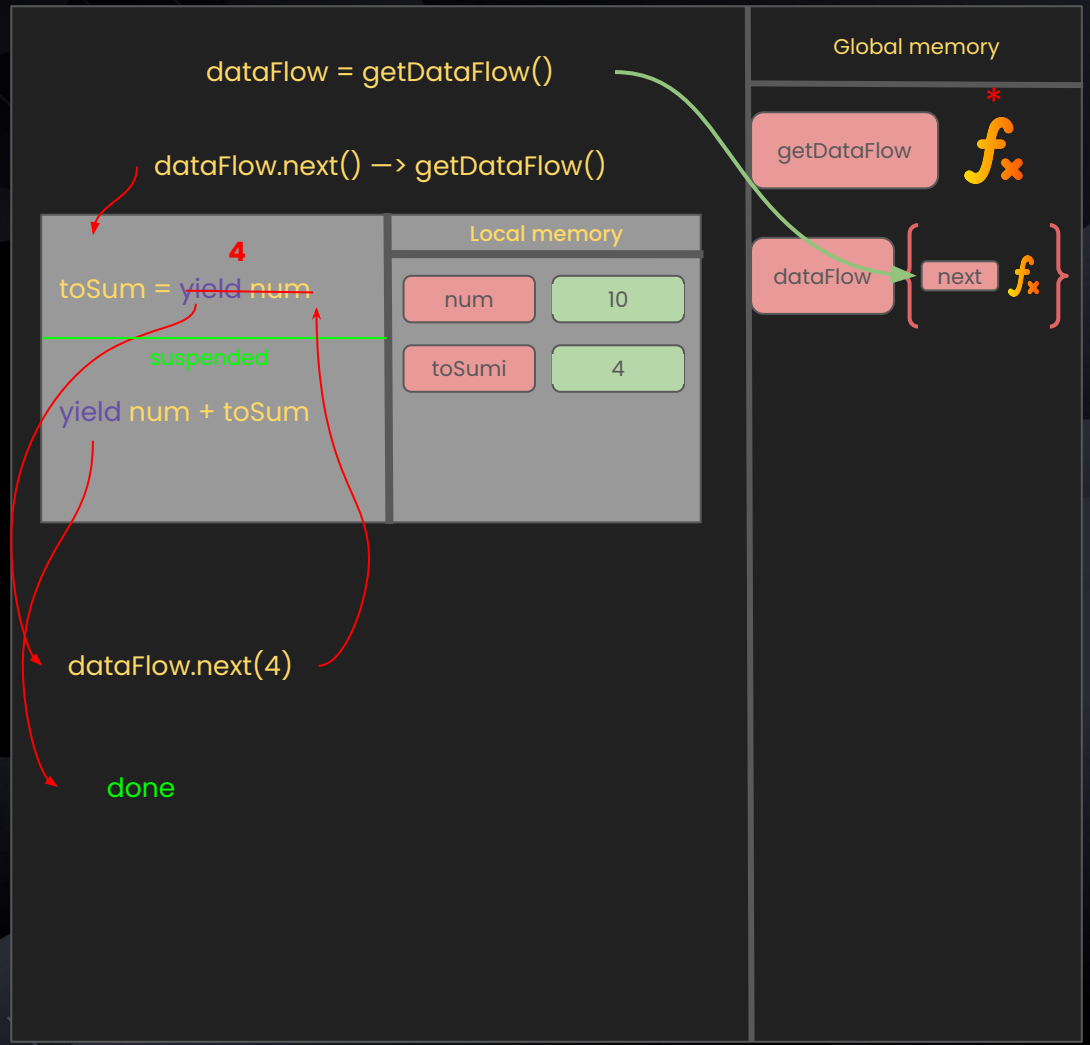
```
const iterbaleObject = {
    0: 1,
    1: 2,
    2: 3,
    length: 3,
    [Symbol.iterator]: function* () {
        for (let i = 0; i < this.length; i++) {
            yield this[i];
        }

        return;

    },
};

for (const test of iterbaleObject) {
    console.log(test);
}

```

generator.js

```
function* getDataFlow() {
    const num = 10;
    const toSum = yield num;
    yield num + toSum;
}

const dataFlow = getDataFlow();

console.log(dataFlow.next().value); // 10
console.log(dataFlow.next(4).value); // 14
```

generator.js

```
1  function* getDataFlow() {
2      const num = 10;
3      const toSum = yield num;
4      yield num + toSum;
5  }
6
7  const dataFlow = getDataFlow();
8
9  console.log(dataFlow.next().value); // 10
10 console.log(dataFlow.next(4).value); // 14
11
```



Immutable Objects

- Every field on an object has 3 internal property descriptors that determine actions that can be performed on the field

writable

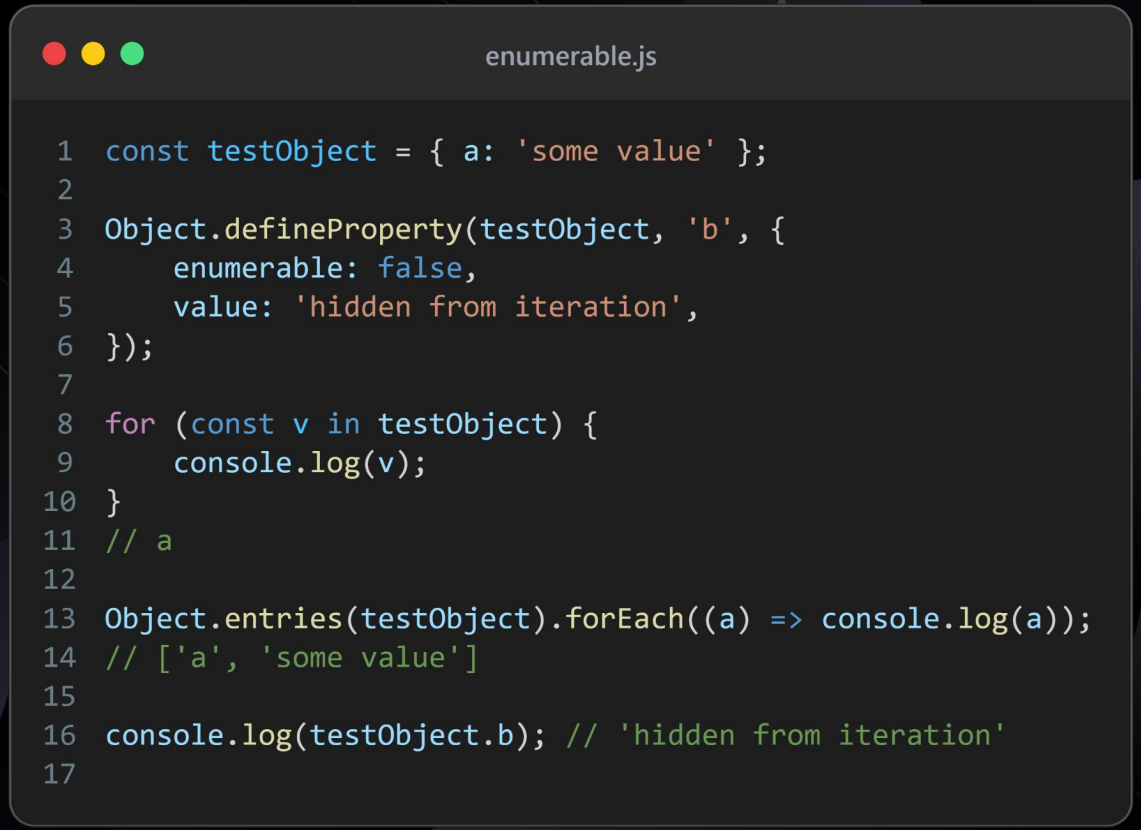
- boolean - determines if the value of a property can be changed

writable.js

```
const testObject = {};

Object.defineProperty(testObject, 'myProp', {
    writable: false,
    value: 'hello',
});

testObject.myProp = 'bye';

console.log(testObject.myProp); // 'hello'

Object.defineProperty(testObject, 'myOtherProp', {
    writable: true,
    value: 'hi',
});

testObject.myOtherProp = 'this can be changed';
console.log(testObject.myOtherProp); // 'this can be changed'

```

enumerable

- boolean - determines if the property shows up in iteration


enumerable.js
1 const testObject = { a: 'some value' };
2
3 Object.defineProperty(testObject, 'b', {
4 enumerable: false,
5 value: 'hidden from iteration',
6 });
7
8 for (const v in testObject) {
9 console.log(v);
10 }
11 // a
12
13 Object.entries(testObject).forEach((a) => console.log(a));
14 // ['a', 'some value']
15
16 console.log(testObject.b); // 'hidden from iteration'
17

configurable

- boolean - if false, property cannot be deleted, other descriptors can’t be changed expect for writable from true to false

configurable.js

```
const testObject = {};

Object.defineProperty(testObject, 'a', {
    configurable: false,
    value: 'cannot be deleted',
});

delete testObject.a;

console.log(testObject.a); // 'cannot be deleted'

Object.defineProperty(testObject, 'a', {
    enumerable: true,
}); // TypeError: Cannot redefine property: a

```

Object freeze

- Object cannot be extended
- All existing properties will have: writable: false, configurable: false
- That means: no new properties, can’t change or delete existing ones
- but if prop is an array or object, then it can still be changed (would require separate freeze)

freeze.js

```
const testObject = {
    a: 'hello',
};

Object.freeze(testObject);

testObject.a = 'bye';
console.log(testObject.a); // 'hello'

testObject.b = 'foo';
console.log(testObject.b); // undefined

delete testObject.a;
console.log(testObject.a); // 'hello'

```

Object seal

- Object cannot be extended
- All existing properties will have: configurable: false
- That means: no new properties, can’t delete existing ones, but still can change existing ones

seal.js

```
1  const testObject = {
2      a: 'hello',
3  };
4
5  Object.seal(testObject);
6
7  testObject.a = 'bye';
8  console.log(testObject.a); // 'bye'
9
10 testObject.b = 'foo';
11 console.log(testObject.b); // undefined
12
13 delete testObject.a;
14 console.log(testObject.a); // 'bye'
15
```


THE
END