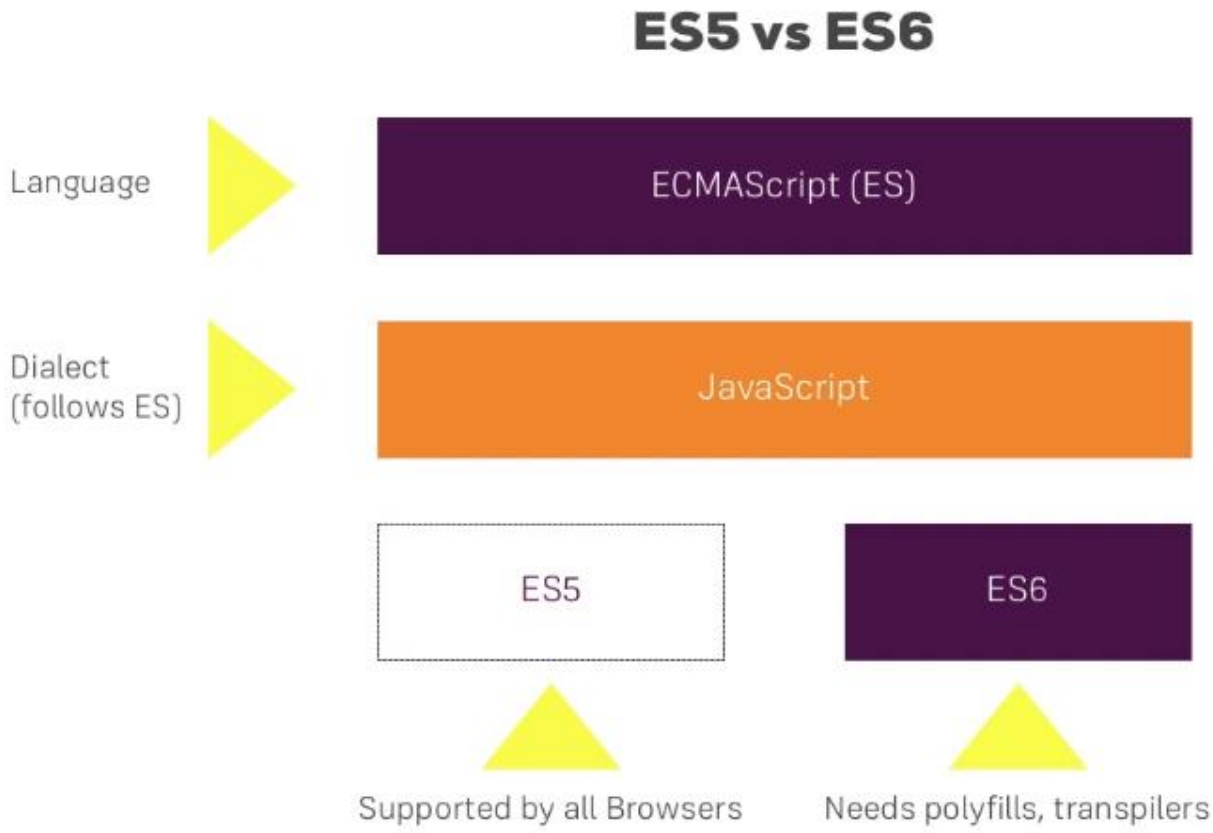
Section 1: Introduction:

3. JavaScript Languages - ES6 vs ES5:



-
- ES needs polyfills or transpilers to convert it to ES6 because still not all browsers supports all features of ES6.

5.ES6 Compatibility with Browsers:

- https://kangax.github.io/compat-table/es6/

6. Using ES6 Today:

- In order to work with ES6 you basically need three things.
- The first thing is a package or a tool which compiles ES6 code to ES5.

  Compiler Examples
  Babel: https://babeljs.io/
  Traceur: https://github.com/google/traceur-compiler
-

- **The second thing is a Module Loader:**
- Working with modules, so you have a lot of JavaScript files you want to load it dynamically. You need a package to do that because this behavior also isn't supported natively by browsers yet. So, you need the module loader.

  Module Loader / Packaging Examples
  SystemJS: https://github.com/systemjs/systemjs
  Webpack: https://github.com/webpack/webpack
-
- And lastly third thing you need a little server serving your application because even though it is static since you're working with modules those modules need to be served dynamically and therefore a little lightweight server is needed for it.

  Servers
  Live-server: https://www.npmjs.com/package/live-server
  Lite-server: https://github.com/johnpapa/lite-server
  Webpack-dev-server: https://webpack.github.io/docs/webpack-dev-server.html
-

Section 2: Syntax Changes & Additions:

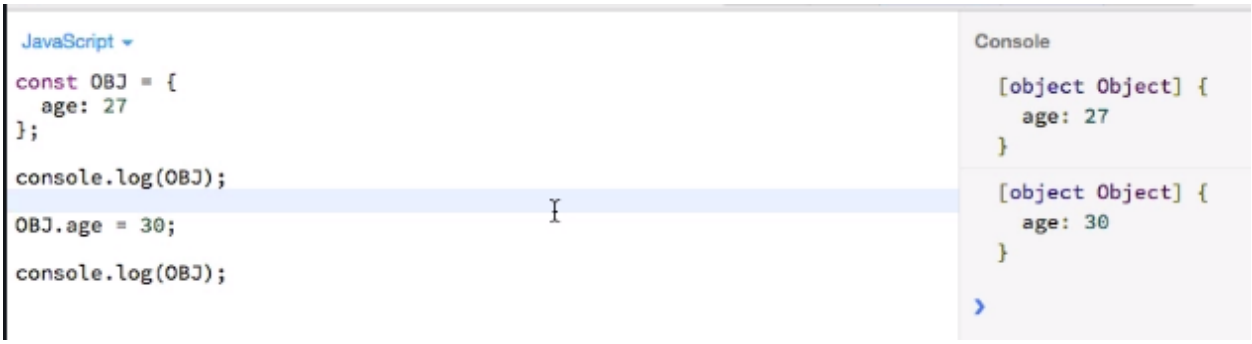
7. Let & Block Scope:

- 

- 

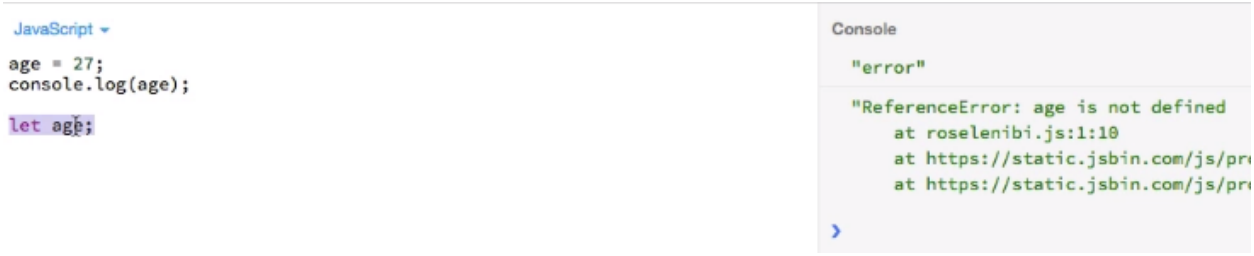
8. Constants with "const":

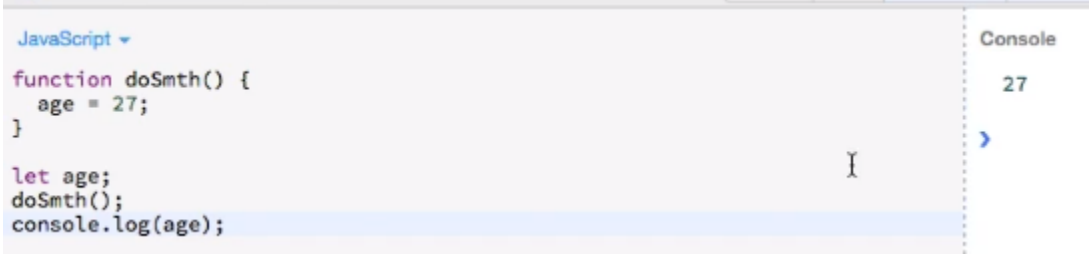
- 

- 

9. Hoisting in ES6:

- 

-
- So, the hoisting of the variables with let or const doesn't work the way it works with var. We have to declare our variable before actually initializing it.
- 

- 

10. (Fat) Arrow Functions:

- JavaScript

```
function fn() {
  console.log('Hello!');
}

fn();
```

Console: "Hello!"

- JavaScript

```
var fn = () => {
  console.log('Hello!');
}

fn();
```

Console: "Hello!"

- JavaScript

```
var fn = () => console.log('Hello!');

fn();
```

Console: "Hello!"

- JavaScript

```
var fn = () => 'Hello';

console.log(fn());
```

Console: "Hello"

- JavaScript

```
var fn = () => {
  let a = 2;
  let b = 3;
  return a + b;
};

console.log(fn());
```

Console: 5

- JavaScript

```
var fn = (a, b) => a + b;

console.log(fn(3, 8));
```

Console: 11

- JavaScript

```
var fn = a => a + 5;

console.log(fn(3));
```

Console: 8

- JavaScript

```
setTimeout(() => console.log('Hello'), 1000);
```

Console: "Hello"

11. (Fat) Arrow Functions and the "this" Keyword:

- **Arrow function has lexical scoping.**

```
function fn() {
  console.log(this);
}

fn();

var fn2 = () => console.log(this);

fn2();
```

- Both this will be referring to global scope(window object).

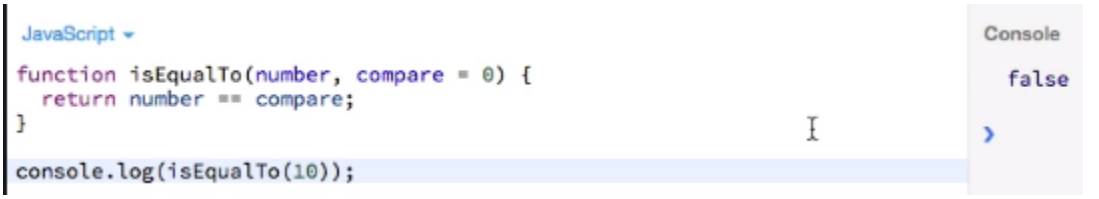
- **Normal Function call on button click event**: In this case 'this' actually refers to what call this function. Here it called on button click, that's why we got HTMLButtonElemet object.

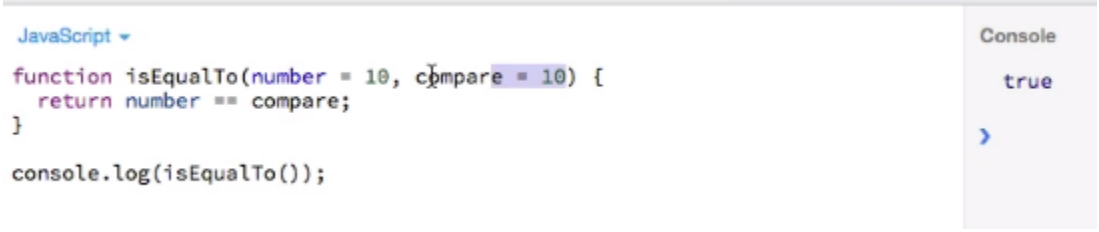
- **Arrow function call on button click:** here 'this' refers to window object.

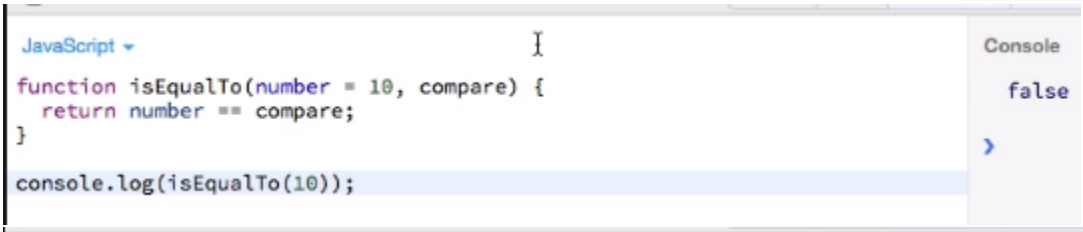
- You won't need bind or apply or call all those work arounds you use with ES5 to get this to the right context, but it will just keep the context in which arrow function is defined.

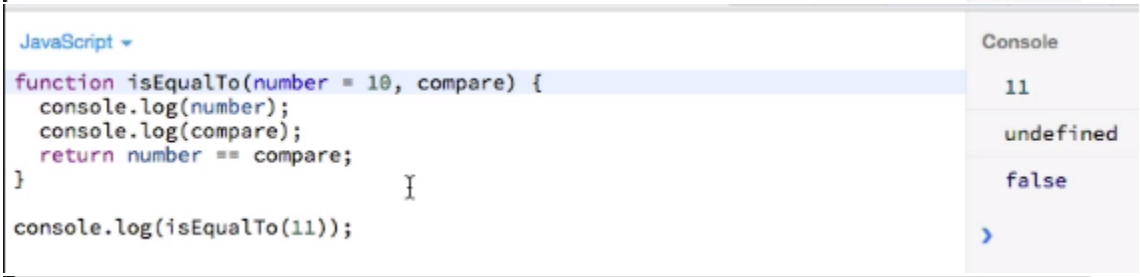
- **Do arrow functions have ‘arguments’? will function and prototype objects will be create for the arrow functions.**

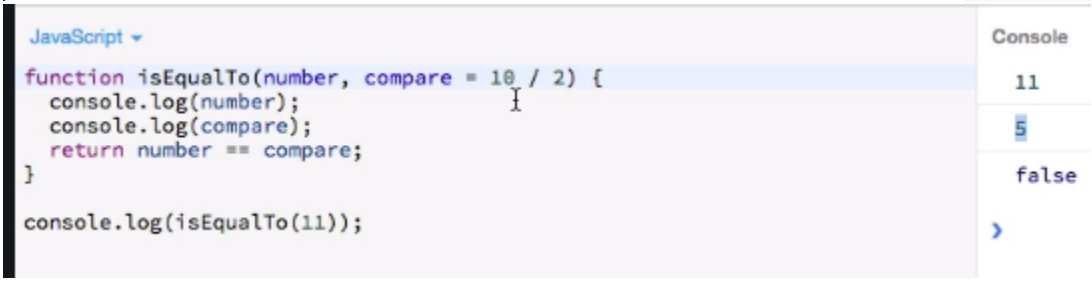
12. Functions and Default Parameters:

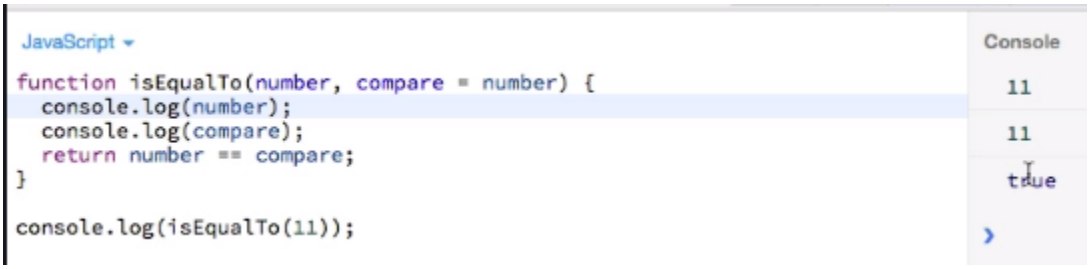
- 

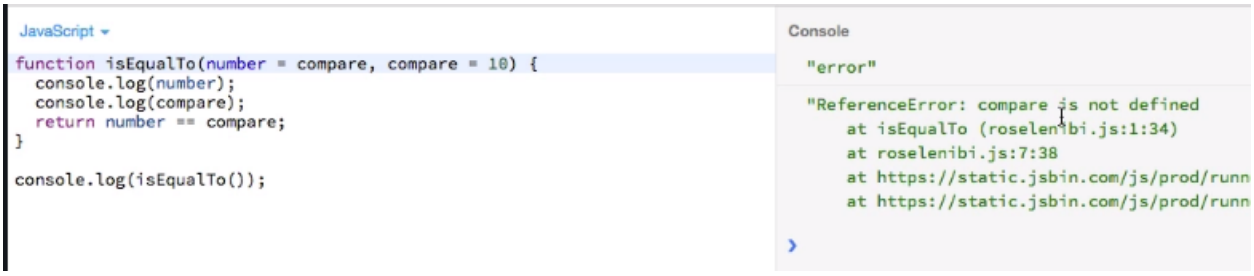
- 

- 

- 

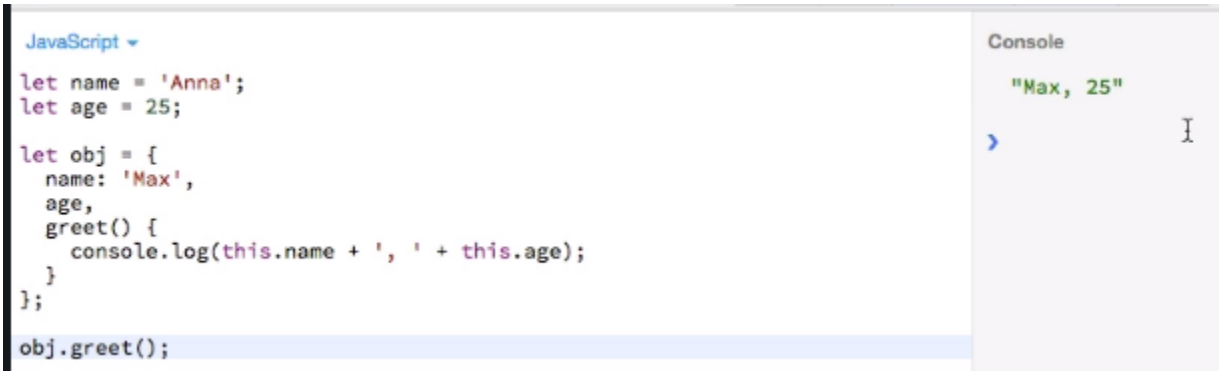
- 

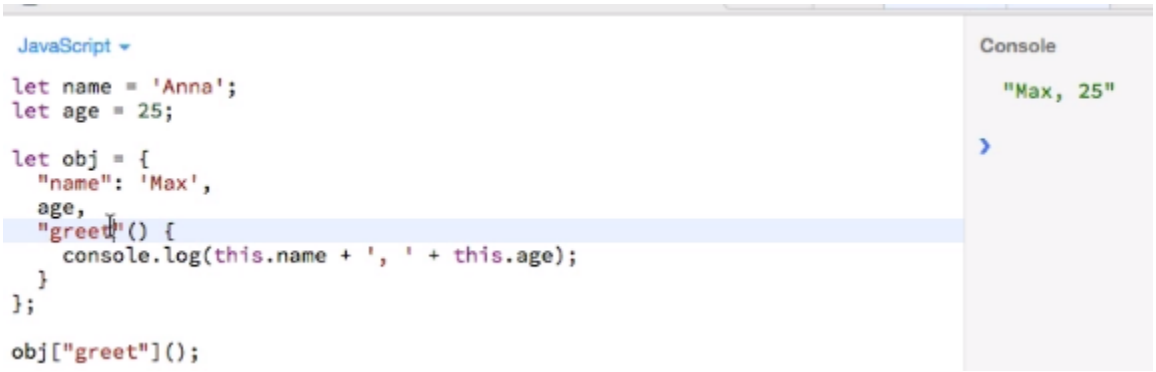
- 

- 

13. Object Literal Extensions:

- 

- Well actually the object will take the values from the surrounding variables.
- So, if we're not specifying values here to initialize this object, it will automatically look if it has variables declared before the declaration of this object, which match the property names declared before.
- 

- 

- **Dynamically add property to object literal:**

```
let name = 'Anna';
let age = 25;

let ageField = "age";

let obj = {
  "name": 'Max',
  [ageField]: 28,
  "greet me"() {
    console.log(this.name + ', ' + this.age);
  }
};
console.log(obj);
obj["greet me"]();
```

```
[object Object] {
  age: 28,
  greet me: greet me() {
    window.runnerWindow.proxyCon
  },
  name: "Max"
}
"Max, 28"
```

14. The Rest Operator:

```
function sumUp(...toAdd) {
  console.log(toAdd);
  let result = 0;
  for (let i = 0; i < toAdd.length; i++) {
    result += toAdd[i];
  }
  return result;
}

console.log(sumUp(100, 10, 20));
```

```
[100, 10, 20]
130
```

```
function sumUp(...toAdd) {
  console.log(toAdd);
  let result = 0;
  for (let i = 0; i < toAdd.length; i++) {
    result += toAdd[i];
  }
  return result;
}

console.log(sumUp(100, 10, "20"));
```

```
[100, 10, "20"]
"11020"
```

15. The Spread Operator:

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

console.log(Math.max(numbers));
```

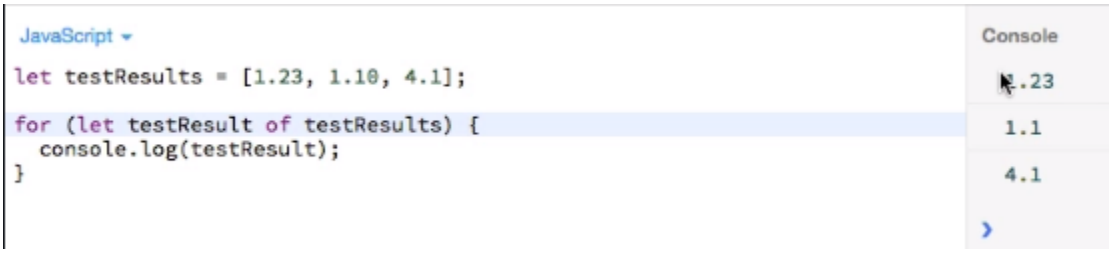
```
NaN
```

- Here max method expecting list of arguments not an array.

- 

- Here spread operator spreads an array into list of arguments.
- Spread operator also used for cloning the object and array. It is the alternate of Object.assign().

16. For of loop:

- 

- This is a shorter syntax and it's perfect for looping through arrays and that is what it's there for you to quickly grab the individual elements off an array and do something with them.

17. Template Literals:

Before ES6, we had to deal with these ugly string concatenations:

```
var name = 'Mosh';
var message = 'Hi ' + name + ',';
```

Now, with template literals (previously called template strings), we can define a string with placeholders and get rid of all those concatenations:

- ```
  var name = 'Mosh';
  var message = `Hi ${name},`;
  ```
  ```

Another benefit of using template literals is that they can expand multiple lines. They are particularly useful when composing email messages:

```
var message = `
Hi ${name},

Thank you for joining my mailing list.

Happy coding,
Mosh
`;
```

-

18. Destructuring – Arrays:

We can also destructure arrays but we use square brackets ([]) instead of curly braces ({}). Let's say we have an array and we want to extra the first and second item and store them in two different variables:

```
// ES5
const values = ['John', 'Smith'];
const first = values[0];
const last = values[1];
// ugly!
```

With destructuring, we can re-write the above code like this:

```
// ES6
const values = ['John', 'Smith'];
const [first, last] = values;
```

-

- JavaScript

```
let numbers = [1,2,3];

let [a, b] = numbers;

console.log(a);
console.log(b);
console.log(numbers);
```

Console

```
1
2
[1, 2, 3]
```

- **With extra element:**

JavaScript

```
let numbers = [1,2,3];

let [a, b, c, d] = numbers;


console.log(d);
```

Console

```
undefined
```

- **With rest parameter:**

JavaScript

```
let numbers = [1,2,3];

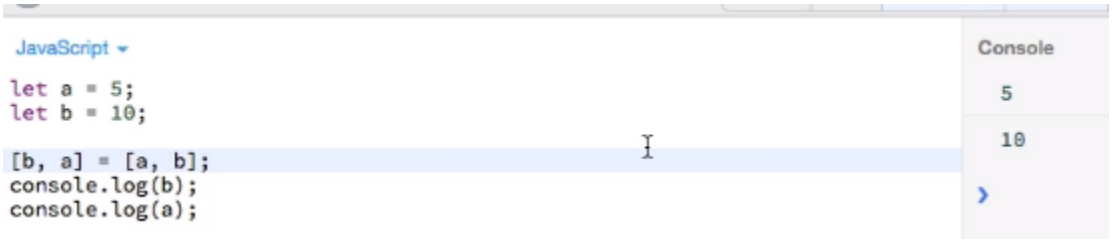
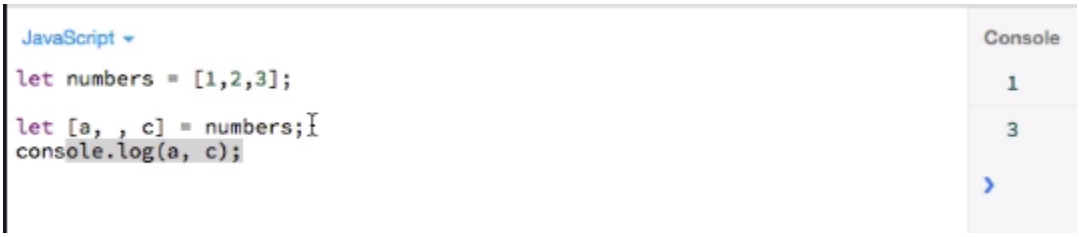
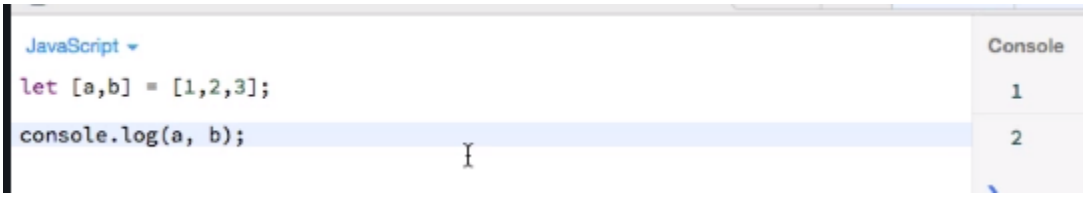
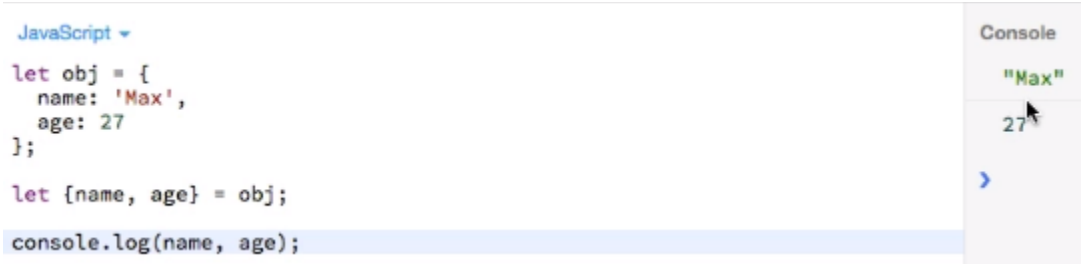
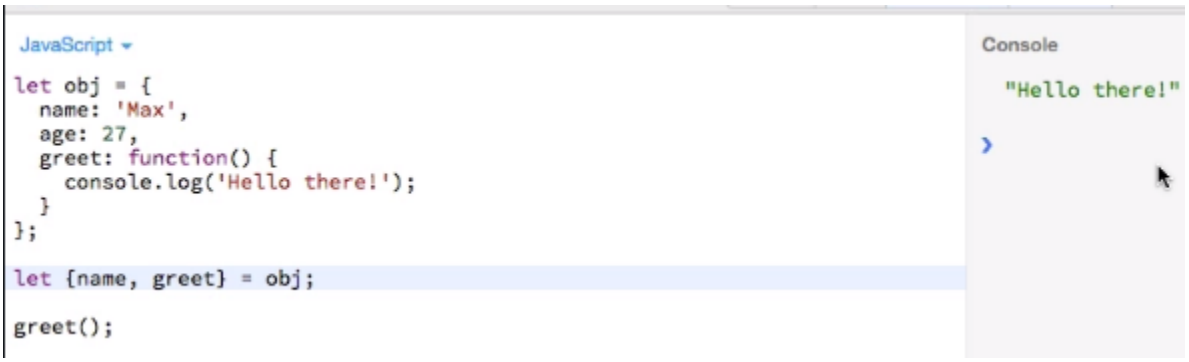
let [a, ...b] = numbers;


console.log(b);
```

Console

```
[2, 3]
```

- **Using default parameters in Destructuring:**

JavaScript

```
let numbers = [1,2,"3"];

let [a, b, c, d = 'Default'] = numbers;


console.log(d);
```

Console

```
"Default"
```

JavaScript

```
let numbers = [1,2,"3"];

let [a = 'Default', b, c, d = 'Default'] = numbers;


console.log(a);
```

Console

```
1
```

- **Swapping with Destructuring:**



- **Destructuring specific values not all values:**



- **Immediately while assigning Destructuring the array:**



19. Destructuring – Objects:



- we could do this in the array where we have a clear order with an object, we cannot do this. In objects we're Destructuring them by a name not by position like in the array.
- 

  Destructuring is an expression that allows us to extract properties from an object, or items from an array. Let's say we have an address object like this:

  ```
  const address = {
     street: '123 Flinders st',
     city: 'Melbourne',
     state: 'Victoria'
  };
  ```

  Now, somewhere else we need to access these properties and store their values in a bunch of variables:

  ```
  const street = address.street;
  const city = address.city;
  const state = address.state;
  ```

  We have this repetitive code over and over: "address." repeated 3 times. Object destructuring gives us a short and clean syntax to extract the value of multiple properties in an object:

- ```
  const { street, city, state } = address;
  ```
  ```

Note that we don't have to list all the properties in the address object. Maybe we're interested only in the **street** property:

```
const { street } = address;
```

- **Object Destructuring:**

  Object destructuring is particularly useful when you're dealing with nested objects:

```
const person = {
   name: 'Mosh',
   address: {
      billing: {
         street: '123 Flinders st',
         city: 'Melbourne',
         state: 'Victoria'
      }
   }
};
```

  Without destructuring, we would have to write this ugly and repetitive code:

```
const street = person.address.billing.street;
const city = person.address.billing.city;
const state = person.address.billing.state;
// So annoying!
```

  Now, we can achieve the same result using a single line of code:

```
const { street, city, state } = person.address.billing;
```

Section 3: Modules & Classes:

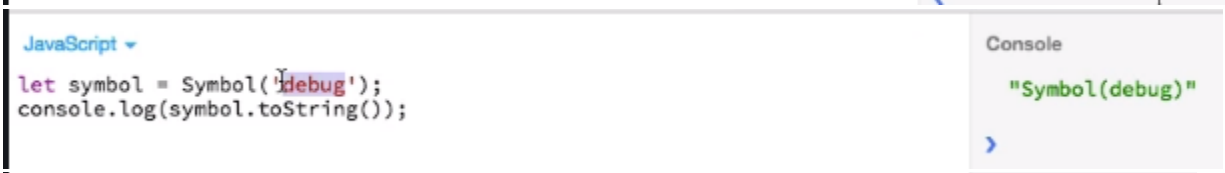
Section 4: Symbols:

40. Introduction:

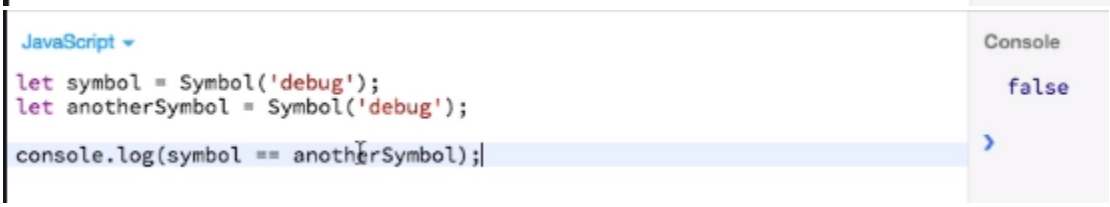
- So, symbols basically are a new primitive type like number, strings or Booleans. And the main thing is they provide a unique identifier.
- Also, symbols are not iterable.

41. Symbols Basics:

- It is kind of constructor thing without the new word.
- 

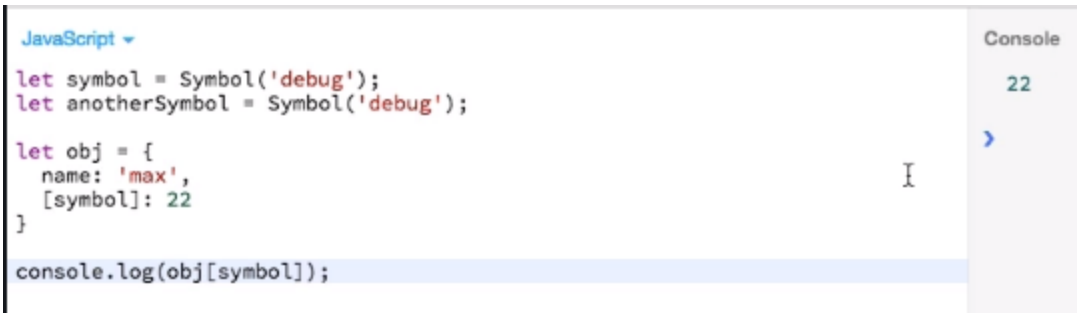
- 

- 

- 

- These symbols do not match because again behind the scenes they stand for unique IDs so are different.
- Now I was already explaining that symbols are great in conjunction with objects.
- To add symbol to our object we need to enclose it into square brackets. Without square brackets it is like adding a property to object not symbol.
- 

- We only see name Max. The symbol is actually not printed out. You also won't see if I use a for loop, but it is still there if I access it explicitly with square brackets notation on the object.

- 

- This is just for meta-programming part. You could store some meta information about this object like created at this time, Add the time stamp if important for your application. And then you could access it if you needed.
- So, this is why symbols are really important and useful tool to add some metaprogramming power to your objects or to your app as a whole.
- Every symbol value returned from Symbol() is unique. A symbol value may be used as an identifier for object properties.

  If you really want to create a `Symbol` wrapper object, you can use the `Object()` function:

```
let sym = Symbol('foo')
typeof sym      // "symbol"
let symObj = Object(sym)
typeof symObj   // "object"
```

- From JavaScript.Info:

Symbol type

By specification, object property keys may be either of string type, or of symbol type. Not numbers, not booleans, only strings or symbols, these two types.

Till now we've been using only strings. Now let's see the benefits that symbols can give us.

Symbols

A "symbol" represents a unique identifier.

A value of this type can be created using `Symbol()`:

```
// id is a new symbol
let id = Symbol();
```

Upon creation, we can give symbol a description (also called a symbol name), mostly useful for debugging purposes:

```
// id is a symbol with the description "id"
let id = Symbol("id");
```

- Symbols are guaranteed to be unique. Even if we create many symbols with the same description, they are different values. The description is just a label that doesn't affect anything.

  For instance, here are two symbols with the same description – they are not equal:

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

-

Symbols don't auto-convert to a string

Most values in JavaScript support implicit conversion to a string. For instance, we can `alert` almost any value, and it will work. Symbols are special. They don't auto-convert.

For instance, this `alert` will show an error:

```
let id = Symbol("id");
alert(id); // TypeError: Cannot convert a Symbol value to a string
```

That's a "language guard" against messing up, because strings and symbols are fundamentally different and should not accidentally convert one into another.

If we really want to show a symbol, we need to explicitly call `.toString()` on it, like here:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), now it works
```

Or get `symbol.description` property to show the description only:

```
let id = Symbol("id");
alert(id.description); // id
```

- ## "Hidden" properties

  Symbols allow us to create "hidden" properties of an object, that no other part of code can accidentally access or overwrite.

  For instance, if we're working with `user` objects, that belong to a third-party code. We'd like to add identifiers to them.

  Let's use a symbol key for it:

  ```
  let user = { // belongs to another code
    name: "John"
  };

  let id = Symbol("id");

  user[id] = 1;

  alert( user[id] ); // we can access the data using the symbol as the key
  ```

...But if we used a string `"id"` instead of a symbol for the same purpose, then there *would* be a conflict:

```
let user = { name: "John" };

// Our script uses "id" property
user.id = "Our id value";

// ...Another script also wants "id" for its purposes...

user.id = "Their id value"
// Boom! overwritten by another script!
```

What's the benefit of using `Symbol("id")` over a string `"id"`?

As `user` objects belongs to another code, and that code also works with them, we shouldn't just add any fields to it. That's unsafe. But a symbol cannot be accessed accidentally, the third-party code probably won't even see it, so it's probably all right to do.

Also, imagine that another script wants to have its own identifier inside `user`, for its own purposes. That may be another JavaScript library, so that the scripts are completely unaware of each other.

Then that script can create its own `Symbol("id")`, like this:

```
// ...
let id = Symbol("id");

user[id] = "Their id value";
```

There will be no conflict between our and their identifiers, because symbols are always different, even if they have the same name.

Symbols in a literal

If we want to use a symbol in an object literal `{...}`, we need square brackets around it.

Like this:

```
let id = Symbol("id");

let user = {
  name: "John",
  [id]: 123 // not "id: 123"
};
```

Symbols are skipped by for...in

Symbolic properties do not participate in `for..in` loop.

For instance:

```
let id = Symbol("id");
let user = {
  name: "John",
  age: 30,
  [id]: 123
};

for (let key in user) alert(key); // name, age (no symbols)

// the direct access by the symbol works
alert( "Direct: " + user[id] );
```

- `Object.keys(user)` also ignores them. That's a part of the general "hiding symbolic properties" principle. If another script or a library loops over our object, it won't unexpectedly access a symbolic property.
  In contrast, Object.assign copies both string and symbol properties:

```
let id = Symbol("id");
let user = {
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123
```

- ## Global symbols

As we've seen, usually all symbols are different, even if they have the same name. But sometimes we want same-named symbols to be same entities. For instance, different parts of our application want to access symbol `"id"` meaning exactly the same property.

To achieve that, there exists a *global symbol registry*. We can create symbols in it and access them later, and it guarantees that repeated accesses by the same name return exactly the same symbol.

In order to read (create if absent) a symbol from the registry, use `Symbol.for(key)`.

That call checks the global registry, and if there's a symbol described as `key`, then returns it, otherwise creates a new symbol `Symbol(key)` and stores it in the registry by the given `key`.

For instance:

```
// read from the global registry
let id = Symbol.for("id"); // if the symbol did not exist, it is created

// read it again (maybe from another part of the code)
let idAgain = Symbol.for("id");

// the same symbol
alert( id === idAgain ); // true
```

-

Symbol.keyFor

For global symbols, not only `Symbol.for(key)` returns a symbol by name, but there's a reverse call: `Symbol.keyFor(sym)`, that does the reverse: returns a name by a global symbol.

For instance:

```
// get symbol by name
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// get name by symbol
alert( Symbol.keyFor(sym) ); // name
alert( Symbol.keyFor(sym2) ); // id
```

The `Symbol.keyFor` internally uses the global symbol registry to look up the key for the symbol. So it doesn't work for non-global symbols. If the symbol is not global, it won't be able to find it and returns `undefined`.

That said, any symbols have `description` property.

For instance:

```
let globalSymbol = Symbol.for("name");
let localSymbol = Symbol("name");

alert( Symbol.keyFor(globalSymbol) ); // name, global symbol
alert( Symbol.keyFor(localSymbol) ); // undefined, not global

alert( localSymbol.description ); // name
```

- ## System symbols

There exist many "system" symbols that JavaScript uses internally, and we can use them to fine-tune various aspects of our objects.

They are listed in the specification in the Well-known symbols table:

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`
- `Symbol.iterator`
- `Symbol.toPrimitive`
- ...and so on.

For instance, `Symbol.toPrimitive` allows us to describe object to primitive conversion. We'll see its use very soon.

- Other symbols will also become familiar when we study the corresponding language features.

Summary

`Symbol` is a primitive type for unique identifiers.

Symbols are created with `Symbol()` call with an optional description (name).

Symbols are always different values, even if they have the same name. If we want same-named symbols to be equal, then we should use the global registry: `Symbol.for(key)` returns (creates if needed) a global symbol with `key` as the name. Multiple calls of `Symbol.for` with the same `key` return exactly the same symbol.

Symbols have two main use cases:

1. "Hidden" object properties. If we want to add a property into an object that "belongs" to another script or a library, we can create a symbol and use it as a property key. A symbolic property does not appear in `for..in`, so it won't be accidentally processed together with other properties. Also it won't be accessed directly, because another script does not have our symbol. So the property will be protected from accidental use or overwrite.

   So we can "covertly" hide something into objects that we need, but others should not see, using symbolic properties.

2. There are many system symbols used by JavaScript which are accessible as `Symbol.*`. We can use them to alter some built-in behaviors. For instance, later in the tutorial we'll use `Symbol.iterator` for iterables, `Symbol.toPrimitive` to setup object-to-primitive conversion and so on.

Technically, symbols are not 100% hidden. There is a built-in method Object.getOwnPropertySymbols(obj) that allows us to get all symbols. Also there is a method named Reflect.ownKeys(obj) that returns *all* keys of an object including symbolic ones. So they are not really hidden. But most libraries, built-in functions and syntax constructs don't use these methods.

-

42. Shared Symbols:

43. Advantages of (unique) IDs / Symbols:

44. Well-Known Symbols:

Section 5: Iterators & Generators:

46. Introduction:

- Iterators are basically all objects that know how a to access values in a collection, one at a time. So, for example an array is such an iterator.
- That means you can loop through it and has a collection of objects and it knows how to output them one after another, means we can loop through.
- Now you can also create your own objects with your own iterator logic.
- A generator is a function which yields different values and especially powerful combinations of course if you use generators and iterators together.

47. Iterator Basics:

- So, an array of course is iterable and an object is also iterable if it has a certain well-known symbol built in.

```
let array = [1, 2, 3];

console.log(typeof array[Symbol.iterator]);
```

```
"function"
```

- So, we have an iterator which is a function that is used when we the actual loop through an array.
- Let's call this iterator function from above image:

```
let array = [1, 2, 3];

let it = array[Symbol.iterator]();

console.log(it);
```

```
[object Array Iterator]
```

- It's an object and you can see array iterator here. And actually, it's an object that only has one method, the next method that I can call this method and let's see what we print:

```
let array = [1, 2, 3];

let it = array[Symbol.iterator]();

console.log(it.next());
```

```
[object Object] {
  done: false,
  value: 1
}
```

- You see we get a new object which has a 'done' property which is set to false and a value of 1. The first value in our array.
- So, let's try again by calling it two times:

```
let array = [1, 2, 3];

let it = array[Symbol.iterator]();

console.log(it.next());
console.log(it.next());
```

```
[object Object] {
  done: false,
  value: 1
}
[object Object] {
  done: false,
  value: 2
}
```

- So, you see by calling it multiple times we kind of seem to be stepping through the values of this array.

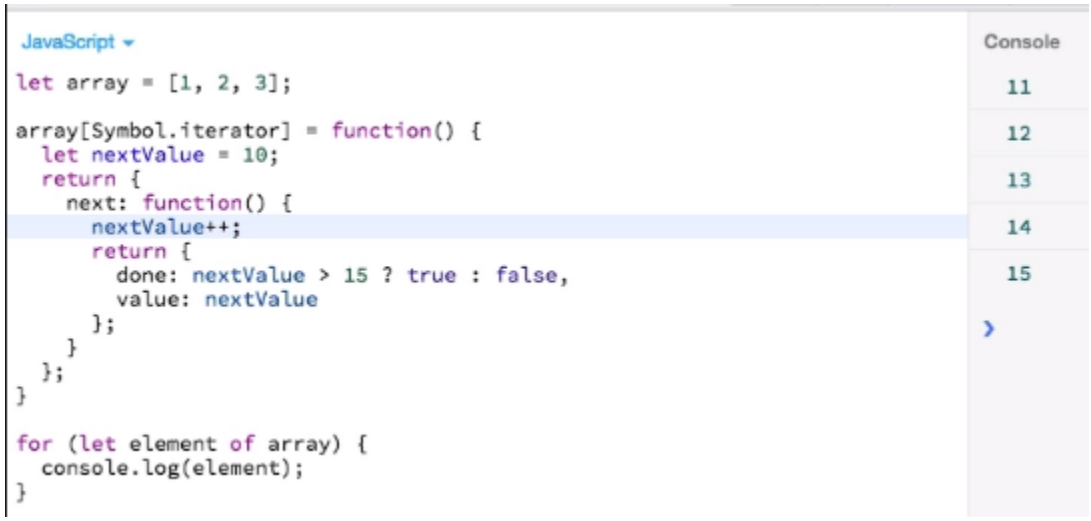
- 

- So ‘done’ will be only true once it really has exhausted all the values in the collection and therefore, we print three times an object with done false and the individual values.
- And it is only true once we reached the first undefined value here.

48. Iterators in Action:

- We can make any object iterable all we have to do is implement this symbol here. and then we are able to loop through, which will allow us to loop through our own objects.
- **Modified iterator function: looping through next method**
- 

- **Modified iterator function: looping through for of loop**



-

49. Creating a Custom, Iteratable Object:

-
- We just made our personal object Iteratable and it's up to you of course to define what we want to iterate.
- 50. Generators Basics:
- A generator looks like a normal function. The first important thing to make it a generator is to add a star an asterisk. Now you may add this directly in front of the function name. You may add it in between with white spaces around or directly after function.
- Important thing to really make it a generator and to do something is to add the yield keyword.

- **Let's see what actually happens if I run this function:**

```
function *select() {
  yield 'House';
  yield 'Garage';
}

select();
```

```
Console
>
```

- Nothing happened. And here comes the connection to iterators that by running a generator we actually get back an iterator we get back an object for which we can loop.
- **So, I'll assign a to 'it' as function call and call it:**

```
function *select() {
  yield 'House';
  yield 'Garage';
}

let it = select();
console.log(it.next());
console.log(it.next());
console.log(it.next());
```

```
Console
[object Object] {
  done: false,
  value: "House"
}
[object Object] {
  done: false,
  value: "Garage"
}
[object Object] {
  done: true,
  value: undefined
}
>
```

- It allows us to create a logic state and a function to yield different values and that we can use an iterator that will loop through those values.

51. Generators in Action:

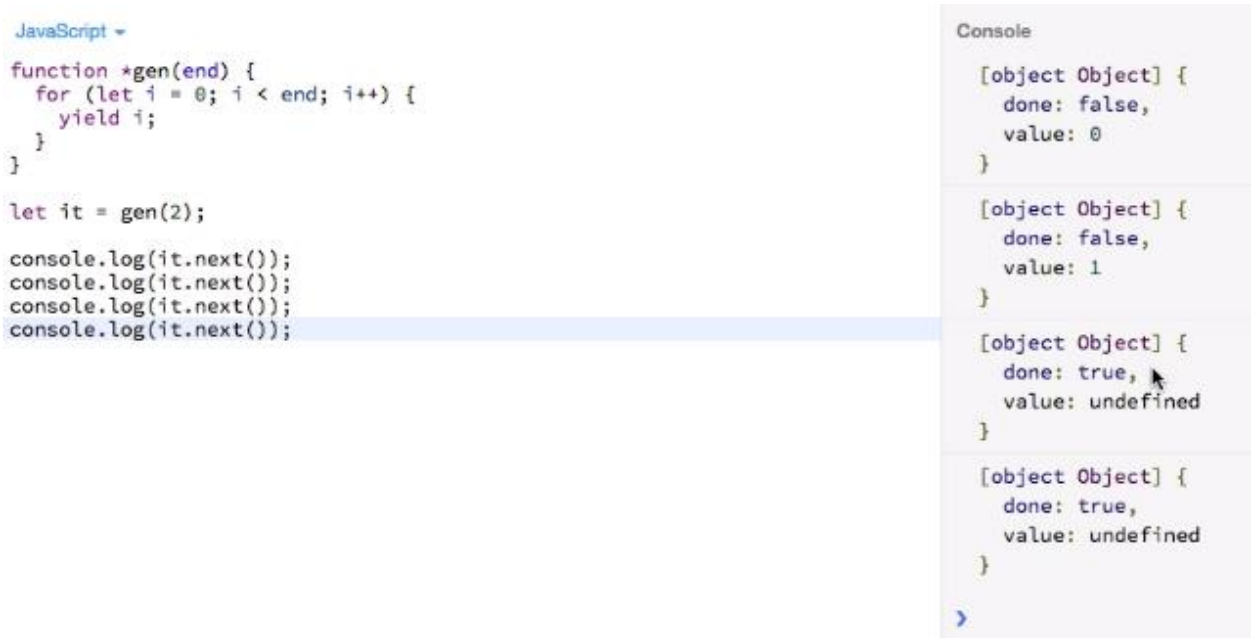
- https://javascript.info/generators

```
let obj = {
  [Symbol.iterator]: gen
}

function *gen() {
  yield 1;
  yield 2;
}

for (let element of obj) {
  console.log(element);
}
```

```
Console
1
2
>
```

- Now we're using a generator and iterator will return us to loop through our object. It's much easier for us to manage state.

- And also imagine the possibilities it offers us. Because with our generator we might also do some let's say asynchronous task and yield those results step by step like fetching something from a server or anything like that.
- And with our iterator we could then use them and take advantage of that step by step approach instead of having a function which runs from start to finish and then gives result.
- **Passing arguments in generator:**

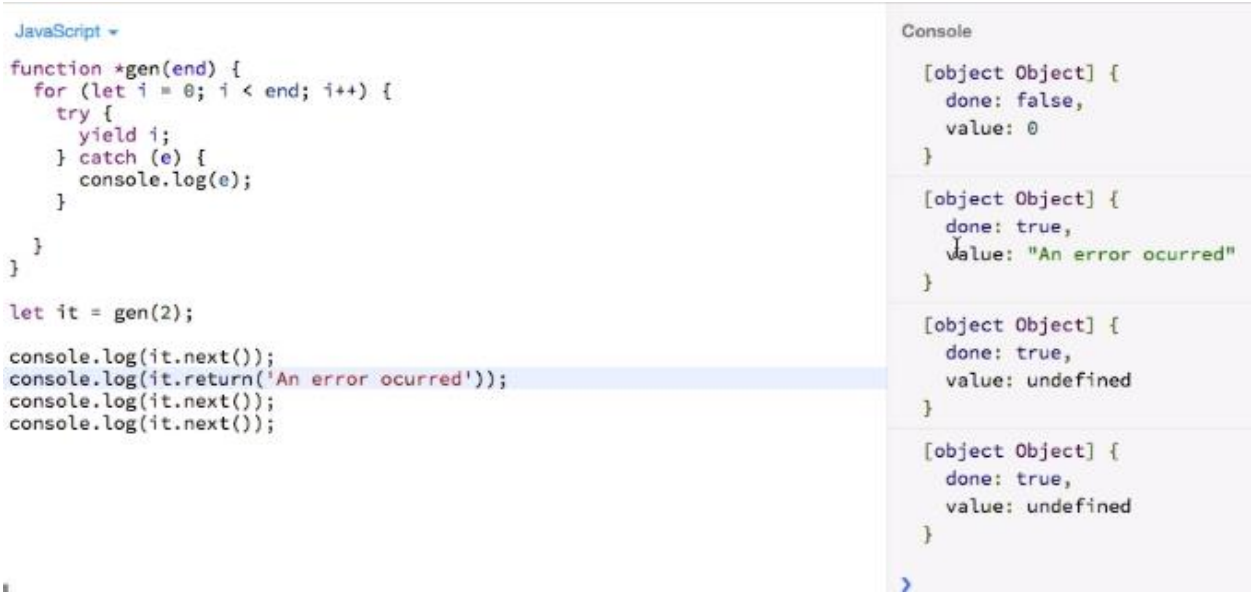


52. Controlling Iterators with throw and return:

- Throwing an error from iterator:

- **Handling error in iterator:**

- **Return keyword in iterator:**



Section 6: Promises:

- They are a useful object a helper to work with asynchronous tasks

Section 7: Extensions of Built-in Objects:

63. The Object:

- **Object.assign(): as merging objects.**

```
JavaScript

var obj1 = {
  a: 1
}

var obj2 = {
  b: 2
}

var obj = Object.assign(obj1, obj2);

console.log(obj);
```

```
Console

  [object Object] {
    a: 1,
    b: 2
  }

>
```

- Well we get a new object which is now the combination of the two objects passed to assign.
-
- **An interesting question is what will happen if we merge two objects which for example have different constructors and they are different prototypes so quickly set up such an example:**

```
JavaScript

class Obj1 {
  constructor() {
    this.a = 1;
  }
}

class Obj2 {
  constructor() {
    this.b = 2;
  }
}

var obj1 = new Obj1();
var obj2 = new Obj2();

var obj = Object.assign(obj1, obj2);

console.log(obj);
```

```
Console

  [object Object] {
    a: 1,
    b: 2
  }

>
```

- But of course, the interesting question is what the prototype of that object is:
- Because the prototype of obj1 will be Obj1.prototype and obj2 will have Obj2.prototype. Also, object one is an instance of Obj1 and obj2 is an instance of Obj2.

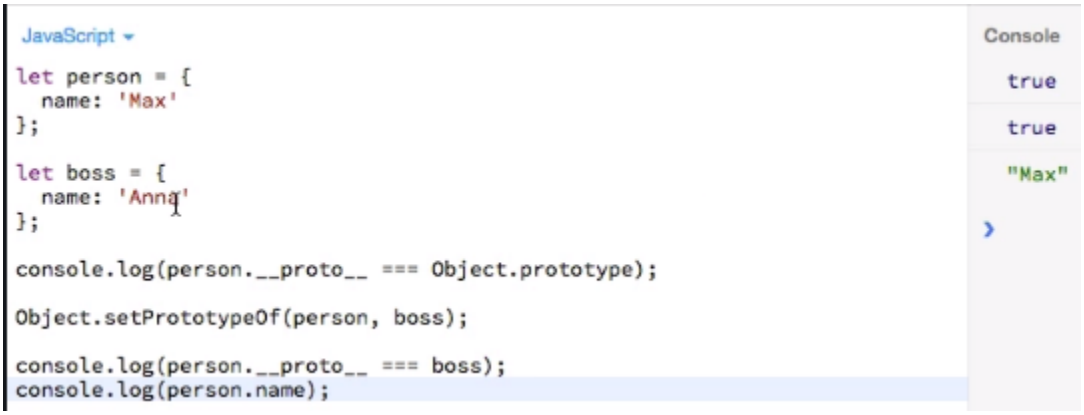
```
JavaScript ▾
class Obj1{
  constructor(){
    this.a=1;
  }
}

class Obj2{
  constructor(){
    this.b=2;
  }
}

var obj1 = new Obj1();
var obj2 = new Obj2();

var obj = Object.assign(obj1,obj2);

console.log(obj instanceof Obj1);
console.log(obj instanceof Obj2);

console.log(obj.__proto__ === Obj1.prototype);
console.log(obj.__proto__ === Obj2.prototype);
console.log(obj.__proto__ === Object.prototype);
```

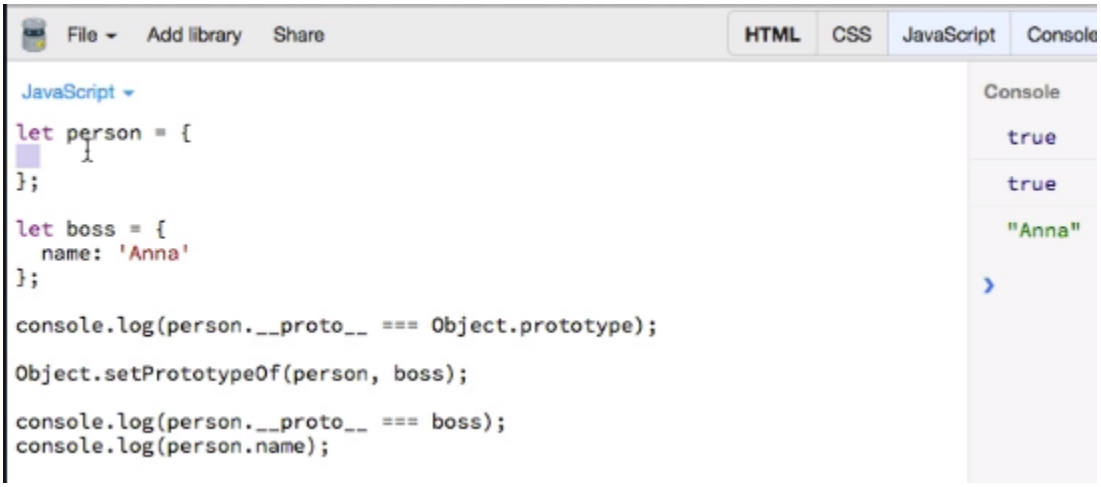
```
Console
  true
  false
  true
  false
  false
›
```

- So, this assign method merge given objects into the target object means into obj1 in our case, without cloning it. I few change obj then it will impact obj1 also.
- **Merging in empty object:**

```
JavaScript ▾
class Obj1{
  constructor(){
    this.a=1;
  }
}

class Obj2{
  constructor(){
    this.b=2;
  }
}

var obj1 = new Obj1();
var obj2 = new Obj2();

var obj = Object.assign({} , obj1, obj2);

console.log(obj.__proto__ === Obj1.prototype);
console.log(obj.__proto__ === Object.prototype);
```

```
Console
  false
  true
›
```

- **setPrototypeOf() method:**
- With Object.create() method we would set the prototype at a time we create an object. Now with this new method we can change it after the object is created.
- 

- 

64. The Math Object:

- **sign() method:**
- 

- 

-
```
let number = 0;
console.log(Math.sign(number));
```
```
Console
0
```
-
```
let number = 'a string';
console.log(Math.sign(number));
```
```
Console
NaN
```

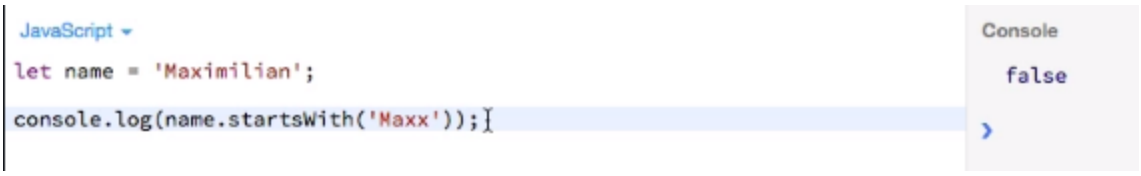
- **trunc() method:**

```
let number = 3.78;
console.log(Math.trunc(number));
```
```
Console
0
3
```
-
```
let number = 0.78;
console.log(Math.trunc(number));
```
```
Console
0
```

- **Diff between Math.floor() and Math.trunc():**
-
```
let number = -3.78;
console.log(Math.floor(number));
```
```
Console
-4
```
-
```
let number = -3.78;
console.log(Math.trunc(number));
```
```
Console
-3
```

65. Strings

- **startsWith(): this is case sensitive.**
-
```
let name = 'Maximilian';
console.log(name.startsWith('Max'));
```
```
Console
true
```

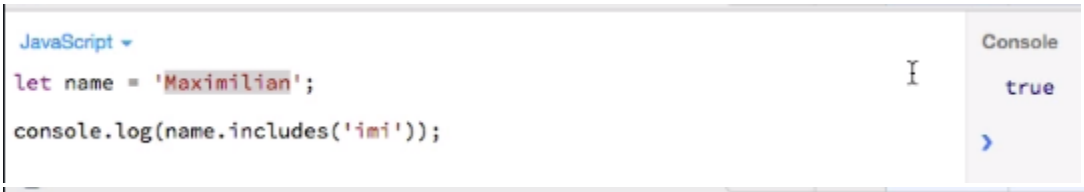
- 

- 

- **endsWith(): this is case sensitive.**
- 

- 

- **Includes():**
- 

- 

- 

-

66. The Number Object:

- **isNaN():**

```
let number = NaN;
console.log(isNaN(number))
```

```
Console
true
>
```

-

```
let number = NaN;
console.log(Number.isNaN(number))
```

```
Console
true
>
```

- **isFinite():**

```
let number = 100000000;
console.log(Number.isFinite(number))
```

```
Console
true
>
```

-

```
let number = Infinity;
console.log(Number.isFinite(number))
```

```
Console
false
>
```

- **isInteger():**

```
let number = Infinity;
console.log(Number.isInteger(number))
```

```
Console
false
>
```

-

```
let number = 10;
console.log(Number.isInteger(number))
```

```
Console
true
>
```

-

```
let number = 10.1;
console.log(Number.isInteger(number))
```

```
Console
false
>
```

- **67. Arrays (1/2)**
-

```
JavaScript
let array = Array(5);
console.log(array);
```

```
Console
[undefined, undefined, undefined, undefined, undefined]
>
```

- **Array.of(): Creating a new array by passing a list of arguments.**
-

```
JavaScript
let array = Array.of(5, 10, 11);
console.log(array);
```

```
Console
[5, 10, 11]
>
```

- **Array.from(): kind of applying a map function on the passed array in the method and it return new array.**
-

```
JavaScript
let array = [10, 12, 16];
console.log(array);

let newArray = Array.from(array, val => val * 2);

console.log(newArray);
console.log(array);
```

```
Console
[10, 12, 16]
[20, 24, 32]
[10, 12, 16]
>
```

- **fill():**
-

```
JavaScript
let array = [10, 12, 16];

array.fill(100);

console.log(array);
```

```
Console
[100, 100, 100]
>
```

-

```
JavaScript
let array = [10, 12, 16];

array.fill(100, 1, 2);

console.log(array);
```

```
Console
[10, 100, 16]
>
```

-

```
JavaScript
let array = [10, 12, 16];

array.fill(100, 0, 2);

console.log(array);
```

```
Console
[100, 100, 16]
>
```

- **Find(): Returns first element matching with the passed arrow function.**

```
let array = [10, 12, 16];

console.log(array.find(val => val >= 12));
```

```
12
```

```
var inventory = [
    {name: 'apples', quantity: 2},
    {name: 'bananas', quantity: 0},
    {name: 'cherries', quantity: 5}
];

function findCherries(fruit) {
    return fruit.name === 'cherries';
}

console.log(inventory.find(findCherries));
```

```
[object Object] {
  name: "cherries",
  quantity: 5
}
```

68. Arrays (2/2):

- **copyWithin():**

```
arr.copyWithin(target[, start[, end]])
```

```
let array = [1, 2, 3];

console.log(array.copyWithin(1, 2));

console.log(array);
```

```
[1, 3, 3]
[1, 3, 3]
```

```
let array = [1, 2, 3];

console.log(array.copyWithin(1, 0));
```

```
[1, 1, 2]
```

```
let array = [1, 2, 3];

console.log(array.copyWithin(1, 0, 2));
```

```
[1, 1, 2]
```

The `copyWithin()` method shallow copies part of an array to another location in the same array and returns it without modifying its length.

JavaScript Demo: Array.copyWithin()

```
1  const array1 = ['a', 'b', 'c', 'd', 'e'];
2
3  // copy to index 0 the element at index 3
4  console.log(array1.copyWithin(0, 3, 4));
5  // expected output: Array ["d", "b", "c", "d", "e"]
6
7  // copy to index 1 all elements from index 3 to the end
8  console.log(array1.copyWithin(1, 3));
9  // expected output: Array ["d", "d", "e", "d", "e"]
10
```

Run ›

Reset

```
> Array ["d", "b", "c", "d", "e"]
> Array ["d", "d", "e", "d", "e"]
```

-
- array.entries():

-

-

Section 8: Maps & Sets

72. Maps - Creation & Adding Items:

- JavaScript

```
let cardAce = {
  name: 'Ace of Spades'
};

let cardKing = {
  name: 'King of Clubs'
};

let deck = new Map();
deck.set('as', cardAce);
deck.set('kc', cardKing);

console.log(deck);
```

Console

```
[object Map] { ... }
```

- JavaScript

```
let cardAce = {
  name: 'Ace of Spades'
};

let cardKing = {
  name: 'King of Clubs'
};

let deck = new Map();
deck.set('as', cardAce);
deck.set('kc', cardKing);

console.log(deck.size);

deck.set('as', cardAce);

console.log(deck.size);
```

Console

```
2
2
```

- JavaScript

```
let cardAce = {
  name: 'Ace of Spades'
};

let cardKing = {
  name: 'King of Clubs'
};

let deck = new Map();
deck.set('as', cardAce);
deck.set('kc', cardKing);

console.log(deck.get('as'));
```

Console

```
[object Object] {
  name: "Ace of Spades"
}
```

- 

- **Clear(): will clear whole map.**
- 

74. Maps - Looping through Maps:

- 

- JavaScript

```
let cardAce = {
  name: 'Ace of Spades'
};

let cardKing = {
  name: 'King of Clubs'
};

let deck = new Map();
deck.set('as', cardAce);
deck.set('kc', cardKing);

for (key of deck.keys()) {
  console.log(key);
}
```

Console

```
"as"
"kc"
```

- JavaScript

```
let cardAce = {
  name: 'Ace of Spades'
};

let cardKing = {
  name: 'King of Clubs'
};

let deck = new Map();
deck.set('as', cardAce);
deck.set('kc', cardKing);

for (value of deck.values()) {
  console.log(value);
}
```

Console

```
[object Object] {
  name: "Ace of Spades"
}
[object Object] {
  name: "King of Clubs"
}
```

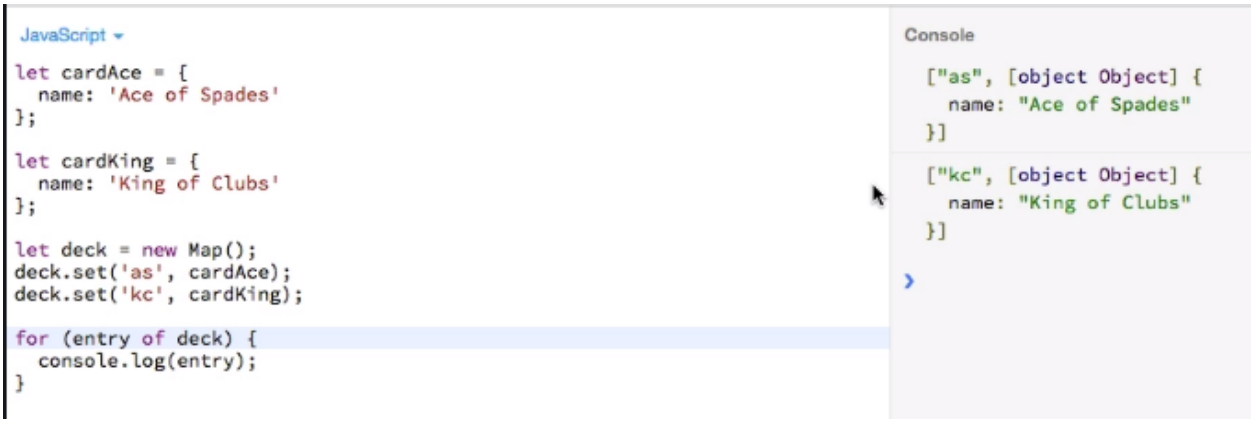
- JavaScript

```
let cardAce = {
  name: 'Ace of Spades'
};

let cardKing = {
  name: 'King of Clubs'
};

let deck = new Map();
deck.set('as', cardAce);
deck.set('kc', cardKing);

for (entry of deck.entries()) {
  console.log(entry);
}
```

Console

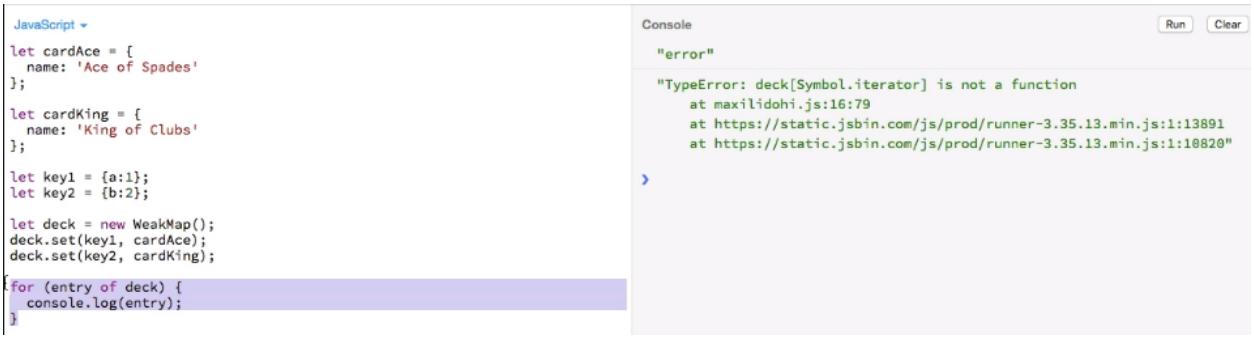
```
["as", [object Object] {
  name: "Ace of Spades"
}]
["kc", [object Object] {
  name: "King of Clubs"
}]
```

- **No need to use entries method it is kind of obsolete.**

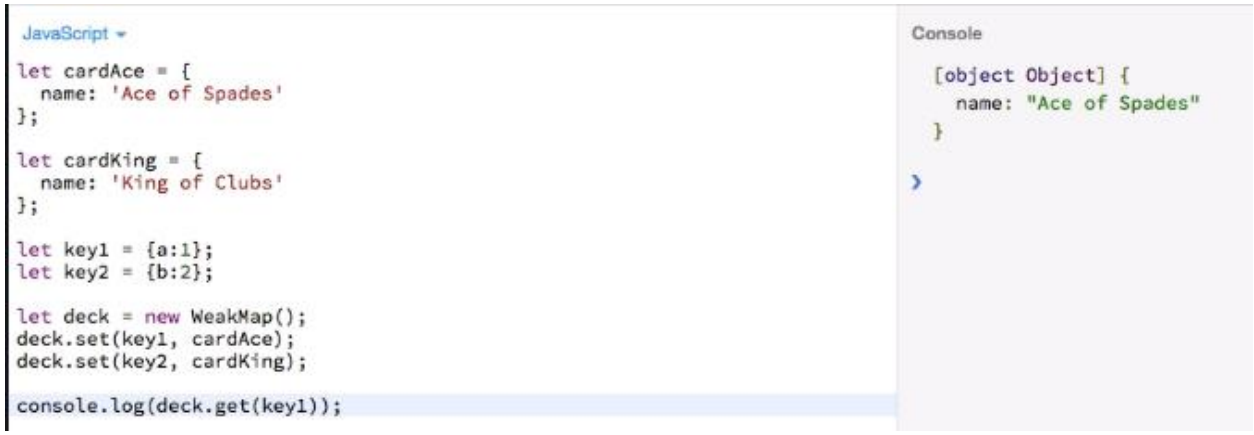
- 

76. The WeakMap:

- 

- Now the reason for this error is that in a weak map your key only be a JavaScript object.
- It is called weak because it holds weak references to the entries in the map.
- This in turn means that Ada is able to get rid of those entries if they're not used anymore for your code so they can be garbage collected for dead behavior
- 

- **We can’t to this because it does not have iterator.**

- 

77. Sets - Creation and Adding Items:

- 

- 

78. Sets - Managing Items:

- 

- JavaScript

```
let set = new Set([1, 1, 1]);

set.add(2);

set.clear();

for (element of set) {
  console.log(element);
}
```

Console

>

- JavaScript

```
let set = new Set([1, 1, 1]);

set.add(2);

console.log(set.has(1));

for (element of set) {
  console.log(element);
}
```

Console

```
true
1
2
>
```

79. Sets - Looping through Sets:

- JavaScript

```
let set = new Set([1, 1, 1]);

set.add(2);

for (element of set.entries()) {
  console.log(element);
}
```

Console

```
[1, 1]
[2, 2]
>
```

- JavaScript

```
let set = new Set([1, 1, 1]);

set.add(2);

for (element of set.keys()) {
  console.log(element);
}
```

Console

```
1
2
>
```

- JavaScript

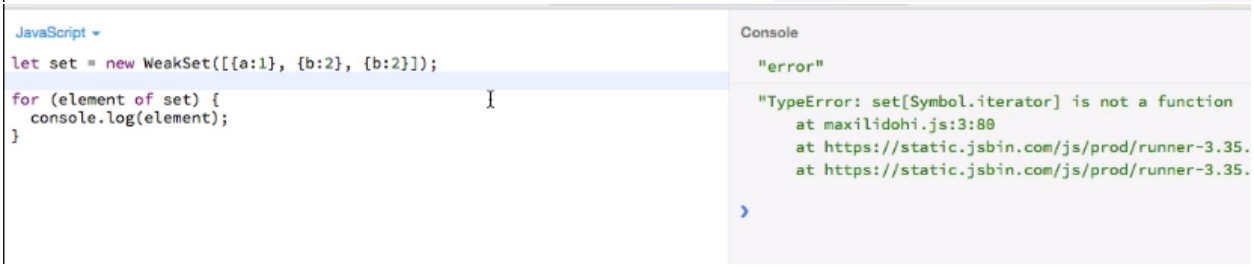
```
let set = new Set([1, 1, 1]);

set.add(2);

for (element of set.values()) {
  console.log(element);
}
```

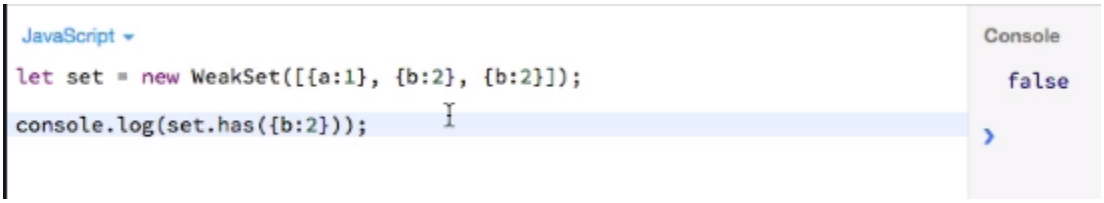
Console

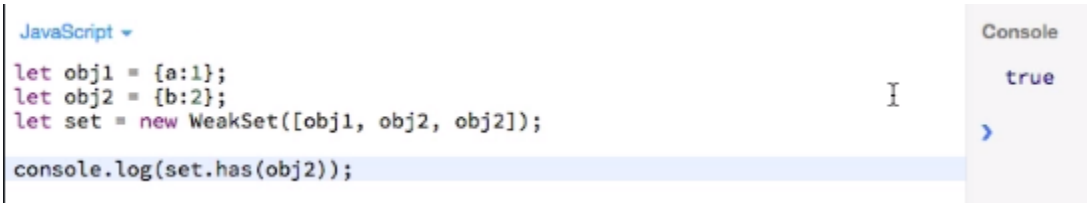
```
1
2
>
```

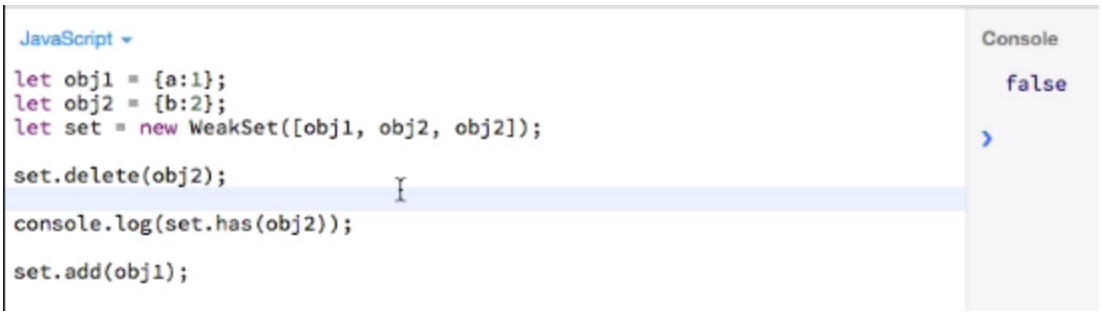
81. The WeakSet:

- Here also values have to be an object for the same reason as with the map only for objects is JavaScript able to well garbage collect them and to determine if they still in use.
- 

- 

- We still get an error of course because like the weak map the weak set is not iterable.
- 

- So, it is like a new object created in the memory. It is not the same that we have in our weak set.
- 

- 
JavaScript
let obj1 = {a:1};
let obj2 = {b:2};
let set = new WeakSet([obj1, obj2, obj2]);
set.delete(obj2);
console.log(set.has(obj2));
set.add(obj1);
Console
false