JavaScript 7

Roi Yehoshua 2018

[DAN.IT] EDUCATION

What we learnt last time?

- Basics of Object Oriented Programming
- JavaScript objects
- Object methods
- Object cloning

[Our targets for today]

- ***this*** keyword
- Call
- Apply
- Working with strings

[DAN.IT EDUCATION]

The **this** Keyword

→ It is common that an object method needs to access the information stored in the object to do its job

→ For example, the code inside user.sayHi() may need the name of the user

→ To access the object, a method can use the **this** keyword

→ The value of **this** is the object “before the dot”, i.e., the object that was used to call the method

```
let user  = {
    name: "John",  age: 30,

    sayHi() {
        alert(this.name); // this == user
    }
};
user.sayHi(); // John
```

Unbounded **this**

→ In JavaScript **this** is “free”, its value is evaluated at call-time and does not depend on where the method was declared, but rather on what’s the object “before the dot”

→ For example, there is no syntax error in a code like this:

```
Function saySomething(){
      alert(this);
}

saySomething(); // undefined (in strict mode)
```

→ In this case **this** is undefined in strict mode

  → If we try to access this.name, there will be an error

→ In non-strict mode (if one forgets use strict) the value of **this** in such case will be the *global object* (window in a browser)

  → This is a historical behavior that "use strict" fixes

[DAN.IT] EDUCATION

this in Arrow Functions

→ Arrow functions are special: they don't have their "own" **`this`**

→ If we reference **this** from such a function, it's taken from the outer "normal" function

→ For instance, here arrow() uses **this** from the outer user.sayHi() method:

```
let user = {
    firstName: "Roi",  sayHi() {
        let func = () => alert(this.firstName);
        func();
    }
};

user.sayHi(); // Roi
```

Call and apply

→ There's a special built-in function method **func.call()** that allows to call a function explicitly setting **this**

→ The syntax is: `func.call(context, arg1, arg2, ...)`

→ It runs func providing the first argument as this, and the next as the arguments

→ As an example, in the code below we call sayHi in the context of different objects

```
function sayHi() {  alert(this.name);
}

let user = { name: "John" };  let admin = { name:
"Admin" };

// use call to pass different objects as "this"
sayHi.call(user); // this = John  sayHi.call(admin);
// this = Admin
```

Call and apply

→ And here we use call to call say with the given context and phrase:

```
function say(time, phrase) {
    alert(`[${time}] ${this.name}: ${phrase}`);
}

let user = { name: "John" };
say.call(user, '10:00', 'Hello'); // [10:00] John: Hello (this=user)
```

→ There is another built-in method **func.apply()** that works almost the same as func.call(), but takes an array-like object instead of a list of arguments:

```
function say(time, phrase) {
    alert(`[${time}] ${this.name}: ${phrase}`);
}

let user = { name: "John" };
let messageData = ['10:00', 'Hello']; // become time and phrase

// user becomes this, messageData is passed as a list of arguments (time, phrase)
say.apply(user, messageData); // [10:00] John: Hello (this=user)
```

[DAN.IT] EDUCATION

Call and apply

→ There is another built-in method func.apply() that works almost the same as func.call()

```
func.apply(context, args)
```

→ The syntax is:

→ The only syntax difference between call and apply is that call expects a list of arguments, while apply takes an array-like object with them

```
function say(phrase) {
    alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// user becomes this, and "Hello" becomes the first argument
say.call(user, "Hello"); // John: Hello
```

[DAN.IT] EDUCATION

Strings

→ In JavaScript, the textual data is stored as strings

  → There is no separate type for a single character

→ The internal format for strings is always UTF-16, it is not tied to the page encoding

→ Strings can be enclosed within either single quotes, double quotes or backticks:

```
let single = 'single-quoted';    let
double   =   "double-quoted";     let
backticks = `backticks`;
```

→ Single and double quotes are essentially the same

→ Backticks, however, allow us to embed any expression into the string, including function calls:

```
function sum(a, b) {  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

[DAN.IT] EDUCATION

Strings

→ Another advantage of using backticks is that they allow a string to span multiple lines:

```
let guestList = `Guests:
        * John
        * Peter
        * Mary
`;
alert(guestList); // a list of guests, multiple lines
```

Special Characters

→ You can create multiline strings with single quotes by using a so-called “newline character”, written as **\n**, which denotes a line break:

```
let guestList = "Guests:\n * John\n * Peter\n * Mary";
alert(guestList); // a multiline list of guests
```

→ There are other, less common “special” characters as well

→ All special characters start with a backslash character \, also called an “escape character”

Character	Description
\b	Backspace
\r	Carriage return
\t	Tab
\uNNNN	A unicode symbol with the hex code NNNN, for instance \u00A9 – is a unicode for the copyright symbol ©. It must be exactly 4 hex digits.
\u{NNNNNNNN}	Some rare characters are encoded with two unicode symbols, taking up to 4 bytes

Special Characters

→ Example with unicode:

```
alert("\u00A9"); // ©
alert("\u{20331}"); // 佱 , a rare chinese hieroglyph (long unicode)
alert("\u{1F60D}"); // 😍 , a smiling face symbol (another long unicode)
```

→ But what if we need to show an actual backslash \ within the string?

→ That's possible, but we need to double it like \\:

```
alert(`The backslash: \\`); // The backslash: \
```

String Length

→ The length property has the string length:

```
alert('My\n'.length); // 3
```

→ Note that \n is a single “special” character, so the length is indeed 3

→ Please note that str.length is a numeric property, not a function

→ There is no need to add brackets after it

Accessing Characters

→ To get a character at position pos, use square brackets [pos] or call str.charAt(pos)

→ charAt() exists mostly for historical reasons

→ The first character starts from the zero position:

```
let str = 'Hello';

// the first character  alert(str[0]);
// H  alert(str.charAt(0)); // H

// the last character
alert(str[str.length - 1]); // o
```

→ We can also iterate over characters using for..of:

```
for (let char of 'Hello') {
    alert(char); // H,e,l,l,o
}
```

[DAN.IT] EDUCATION

String are Immutable

→ Strings can't be changed in JavaScript. It is impossible to change a character.

→ Let's try it to show that it doesn't work:

```
let str = 'Hi';

str[0] = 'h';    // doesn't work
alert(str[0]);   // H
```

→ The usual workaround is to create a whole new string and assign it to str instead of the old one:

```
str = 'h' +  str[1];   // replace the string
alert(str);            // hi
```

[DAN.IT] EDUCATION

Changing the Case

→ Methods **toLowerCase()** and **toUpperCase()** change the case:

```
alert('Interface'.toUpperCase());  // INTERFACE
alert('Interface'.toLowerCase());  // interface
```

→ Or, if we want a single character lowercased:

```
alert('Interface'[0].toLowerCase()); // 'i'
```

Searching for substrings

→ There are multiple ways to look for a substring within a string

→ **str.indexOf**(substr, pos) looks for the substr in str, starting from the given position pos, and returns the position where the match was found or -1 if nothing can be found

```
let str = 'Widget with id';

alert(str.indexOf('Widget')); // 0, because 'Widget' is found at the beginning
alert(str.indexOf('widget')); // -1, not found, the search is case-sensitive
alert(str.indexOf("id")); // 1, "id" is found at the position 1 (..idget with id)
alert(str.indexOf("id", 2)) // starting the search from position 2
```

→ There is also a similar мфethod **str.lastIndexOf**(pos) that searches from the end of a string to its beginning

```
alert(str.lastIndexOf("id")); // 12
```

Searching for substrings

→ If we're interested in all occurrences, we can run indexOf in a loop

→ Every new call is made with the position after the previous match

```
let str = 'As sly as a fox, as strong as an ox';  let target = 'as'; // let's look for it
let pos = 0;  while (true) {
    let foundPos = str.indexOf(target, pos);
    if (foundPos == -1) break;

    alert(`Found at ${foundPos}`);
    pos = foundPos + 1; // continue the search from the next position
}
```

→ The same algorithm can be layed out shorter:

```
let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
    alert(`Found at ${pos}`);
}
```

[DAN.IT EDUCATION]

Searching for substrings

→ **str.includes**(substr, pos) returns whether str contains substr within

→ It's useful if we need to test for the match, but don't need its position

→ The optional second argument of str.includes is the position to start searching from

```
alert("Midget".includes("id")); // true
alert("Midget".includes("id", 3)); // false, from position 3 there is no "id"
```

→ The methods **str.startsWith**() and **str.endsWith**() do exactly what they say:

```
alert("Widget".startsWith("Wid")); // true, "Widget" starts with "Wid"
alert("Widget".endsWith("get"));   // true, "Widget" ends with "get"
```

[DAN.IT] EDUCATION

Getting a substring

→ There are 3 methods in JavaScript to get a substring:

Method	Selects…	Negatives
slice(start, end)	from start to end (not including end)	allows negatives
substring(start, end)	between start and end allows start to be greater than end	negative values mean 0
substr(start, length)	from start get length characters	allows negative start

→ Negative values for start/end mean that the position is counted from the string end

→ Examples for slice():

```
let str = "stringify";

alert(str.slice(0, 5)); // 'strin', the substring from 0 to 5 (not including 5)  alert(str.slice(0, 1)); // 's', from 0 to 1, but not including 1, so only character at 0  alert(str.slice(2)); // ringify, from the 2nd position till the end

alert(str.slice(-4, -1)); // gif, start at the 4th position from the right, end at the 1st from the right
```

Getting a substring

→ Examples for substring():

```
let str = "stringify";

// these are same for substring  alert(str.substring(2,
6)); // "ring"  alert(str.substring(6, 2)); // "ring"

// ...but not for slice:
alert(str.slice(2, 6)); // "ring" (the same)
alert(str.slice(6, 2)); // "" (an empty string)
```

→ Examples for substr():

```
let str = "stringify";
alert(str.substr(2, 4)); // ring, from the 2nd position get 4 characters
alert(str.substr(-4, 2)); // gi, from the 4th position get 2 characters
```

→ Although all three methods can do the same job, slice() is more commonly used

[DAN.IT] EDUCATION

Control questions

1. How does **this** keyword work?
2. When do we use **call** and **apply**?
3. How can we add special character on page?
4. How can we find a substring inside a string?

Materials

Core materials:
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this
https://learn.javascript.ru/object-methods
https://learn.javascript.ru/call-apply
http://learn.javascript.ru/es-string

Additional materials:
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/call
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/apply
https://habr.com/company/ruvds/blog/350536/

Video materials:
https://youtu.be/213r4EOHfF0

[DAN.IT] EDUCATION