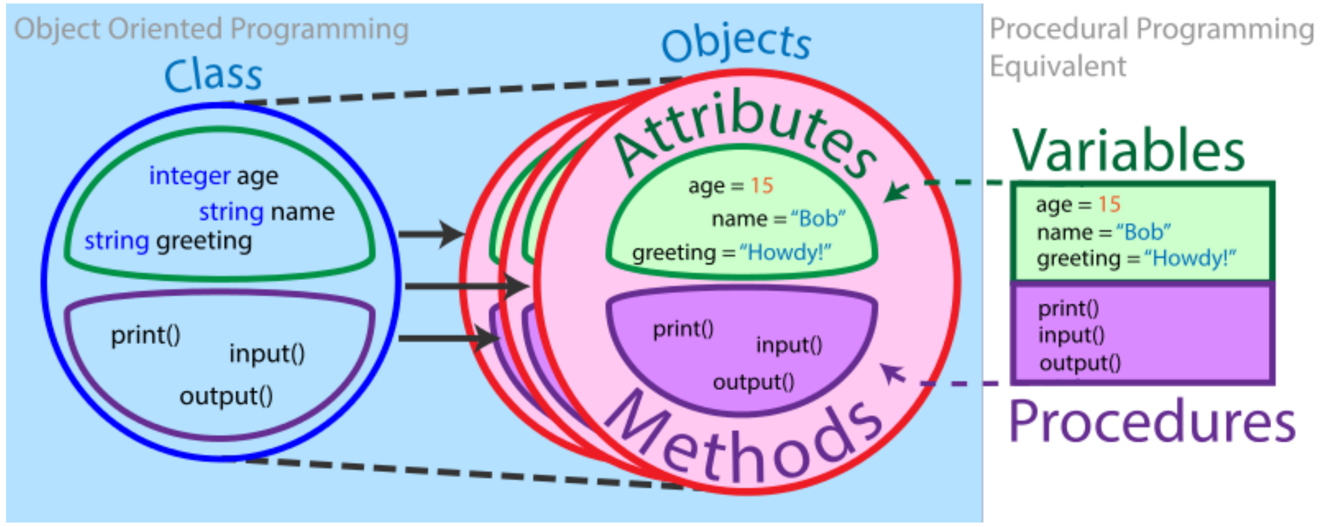
JavaScript OOP

Roi Yehoshua
2018

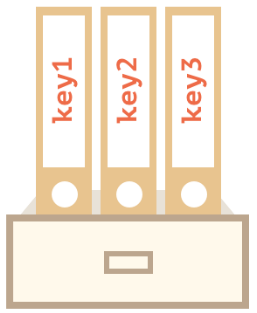
Object Oriented Programming

- **Object-oriented programming (OOP)** is a programming paradigm based on the concept of "objects", which represent the entities in the program
- Objects may contain data, in the form of fields, also known as attributes or properties
- Objects may contain code, in the form of functions, also known as methods
- Objects are instances of classes, which also determine their type



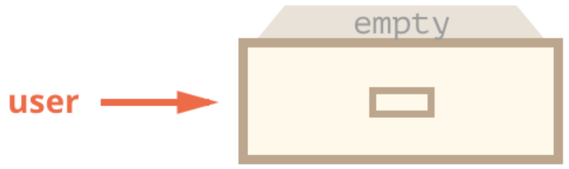
Objects in JavaScript

- An object is a collection of related data and/or functionality
- We can imagine an object as a cabinet with signed files
  - Every piece of data is stored in its file by the key
  - It's easy to find a file by its name or add/remove a file



- An empty object ("empty cabinet") can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
```



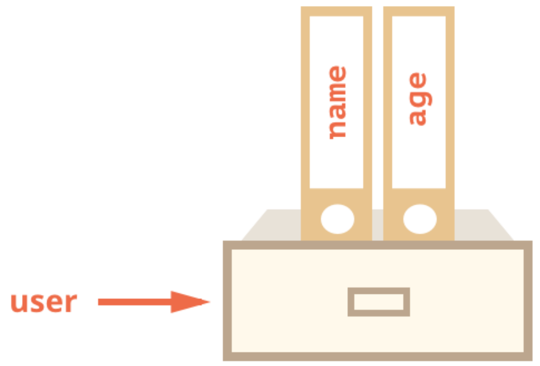
- Usually, the figure brackets {...} are used. That declaration is called an *object literal*.

Literals and Properties

- We can immediately put some properties into {...} as "key: value" pairs:

```
let user = {      // an object
    name: 'John',  // by key "name" store value 'John'
    age: 30        // by key "age" store value 30
};
```

- A property has a key (also known as "name" or "identifier") before the colon ":" and a value to the right of it
- In the user object, there are two properties:
  - The first property has the name "name" and the value 'John'
  - The second one has the name "age" and the value 30



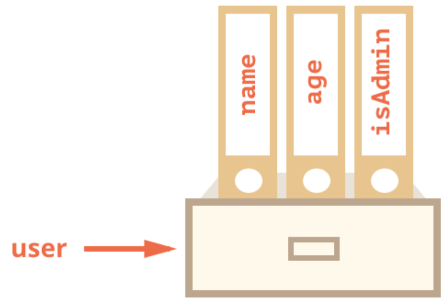
Literals and Properties

- Property values are accessible using the **dot notation**:

```
// Get fields of the object
alert(user.name); // John
alert(user.age); // 30
```

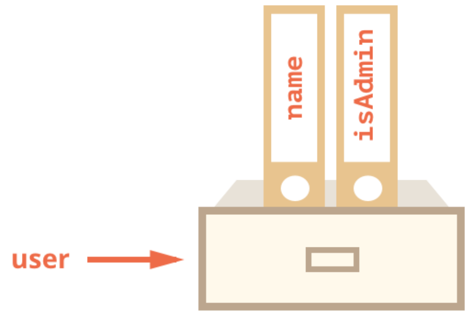
- You can add new properties to an object also using the dot notation
  - The property can be of any type

```
// Add properties to an object
user.isAdmin = true;
```



- To remove a property, we can use **delete** operator:

```
// Delete properties from an object
delete user.age;
```



Literals and Properties

- Square brackets also provide a way to obtain the property name as the result of any expression, which may depend on user input as in the following example:

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

let key = prompt("What do you want to know about the user?", "name");

// access by variable
alert(user[key]); // John (if enter "name")
```

- Square brackets are much more powerful than the dot notation, but they are also more cumbersome to write
- So usually when property names are known and simple, the dot is used. And if we need something more complex, then we switch to square brackets.

Exercise (1)

- Write the following code, one line for each action:
  - Create an empty object product
  - Add the property name with the value 'Laptop'
  - Add the property price with the value 1200
  - Change the value of the price to 1000
  - Show the product's name and price on the screen
  - Remove the property name from the object

Property Value Shorthand

- The use-case of making a property from a variable with the same name is so common, that there's a special *property value shorthand* to make it shorter:

```
let name = "John";
let age = 30;

let user = {
    name: name,
    age: age
};
alert(user.name); // John
```

```
let name = "John";
let age = 30;

let user = {
    name, // same as name: name
    age // same as age: age
};
alert(user.name); // John
```

- We can use both normal properties and shorthands in the same object:

```
let user = {
    name,  // same as name: name
    age: 30
};
```

Property Existence Check

- Accessing a non-existing property returns undefined
- Thus, you can test if a property exists by getting it and comparing it to undefined:

```
let user = {};

alert(user.noSuchProperty === undefined); // true means "no such property"
```

- There also exists a special operator in to check for the existence of a property

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

alert("age" in user); // true, user.age exists
alert("blabla" in user); // false, user.blabla doesn't exist
```

for..in Loop

- To iterate over all keys of an object, there exists a special form of the loop: for..in
- The syntax:

```
for (key in object) {
    // executes the body for each key among object properties
}
```

- For instance, let’s output all properties of user:

```
let user = {
    name: "John",
    age: 30,
    isAdmin: true
};

for (let key in user) {
    // keys
    alert(key);  // name, age, isAdmin
    // values for the keys
    alert(user[key]); // John, 30, true
}
```

Exercise (2)

- We have an object storing salaries of our team:

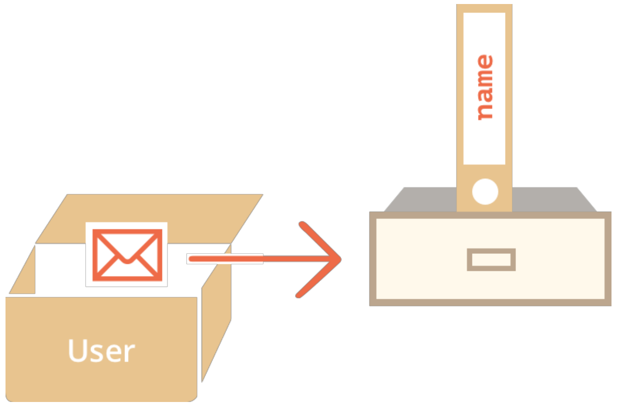
```
let salaries = {
    John: 100,
    Ann: 160,
    Peter: 130
}
```

- Write the code to sum all salaries and store in the variable sum
  - Should be 390 in the example above
- If salaries is empty, then the result must be 0

Object References

- When you define an object in JavaScript, the object’s variable stores not the object itself, but its address in memory, i.e., a reference to it

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



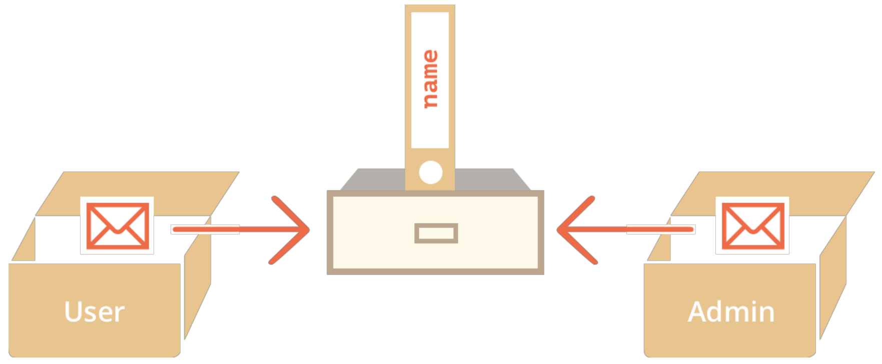
- The object is stored somewhere in memory, and the variable user has a “reference” to it
- If we imagine an object as a cabinet, then a variable is a key to it

Copying Objects by Reference

- When an object variable is copied (or passed as a function argument) – the reference is copied, not the object:

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

let admin = user; // copy the reference
```



- Now we have two variables, each one with the reference to the same object
- We can use any variable to access the cabinet and modify its contents:

```
admin.name = "David"; // changed by the "admin" reference
alert(user.name); // "David", changes are seen from the "user" reference
```

Comparing Objects by Reference

- The equality == and strict equality === operators for objects work exactly the same
- Two objects are equal only if they are the same object
- For instance, two variables reference the same object, they are equal:

```
let a = {};
let b = a; // copy the reference

alert(a == b); // true, both variables reference the same object
alert(a === b); // true
```

- And here two independent objects are not equal, even though both are empty:

```
let a = {};
let b = {}; // two independent objects

alert(a == b); // false
```

Const Object

- An object declared as const *can* be changed
- For instance:

```
const user = {
    name: "John"
};
user.age = 25; // No error, since the reference "user" doesn't change
alert(user.age); // 25
```

- The const would give an error if we try to set user to something else, for instance:

```
// Error (can't reassign user)
user = {
    name: "David"
};
```

Cloning Objects

- Copying by reference is good most of the time
- However, there are we cases where we want to duplicate an existing object, i.e., create an independent copy of it
- We can write the code that replicates the structure of an existing object by iterating over its properties and copying them on the primitive level (using a for..in loop)
- Also we can use the method Object.assign() for that. Its syntax is:

```
Object.assign(dest[, src1, src2, src3...])
```

  - Arguments dest, and src1, ..., srcN (can be as many as needed) are objects
  - It copies the properties of all objects src1, ..., srcN into dest
  - Then it returns dest

Cloning Objects

- For example, we can copy all properties of user into an empty object

```
let user = {
    name: "John",
    age: 30
};
let clone = Object.assign({}, user);

// now clone is a fully independent clone
clone.name = "David"; // changed the data in it
alert(user.name); // still John in the original object
```

- Note that if there are properties in user that are references to other objects, they will be copied by reference
- There is a standard algorithm for **deep cloning** that handles such cases
  - You can use a working implementation of it from the JavaScript library lodash
  - The method is called _.cloneDeep(obj).

Object Methods

- JavaScript methods are actions that can be performed on objects
- A method is a property containing a function definition:

```
let user = {
    name: "John",
    age: 30
};
user.sayHi = function () {
    alert("Hello!");
};

user.sayHi(); // Hello!
```

- Here we've used a Function Expression to create the function and assign it to the property user.sayHi of the object
- So, here we've got a method sayHi of the object user

Method Shorthand

- There exists a shorter syntax for methods in an object literal:

```
let user = {
    sayHi: function () {
        alert("Hello");
    }
};
```

- we can omit "function" and just write sayHi():

```
let user = {
    sayHi() { // same as "sayHi: function()"
        alert("Hello");
    }
};
```

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

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
```

Exercise (3)

- Create an object calculator with three methods:
  - read() prompts for two values and saves them as object properties
  - sum() returns the sum of saved values
  - mul() multiplies saved values and returns the result

```
let calculator = {
    // ... your code ...
};

calculator.read();
alert(calculator.sum());
alert(calculator.mul());
```

Exercise (4)

- Here the function makeUser() returns an object
- What is the result of accessing its ref? Why?

```javascript
function makeUser() {
    return {
        name: "John",
        ref: this
    };
};

let user = makeUser();

alert(user.ref.name); // What's the result?
```

Exercise (5)

- There's a ladder object that allows to go up and down:

```
let ladder = {
    step: 0,
    up() {
        this.step++;
    },
    down() {
        this.step--;
    },
    showStep: function () { // shows the current step
        alert(this.step);
    }
};
```

- Now, if we need to make several calls in sequence, can do it like this:

```
ladder.up();
ladder.up();
ladder.down();
ladder.showStep(); // 1
```

- Modify the code of up and down to make the calls chainable, like this:

```
ladder.up().up().down().showStep(); // 1
```

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)
```

Call and apply

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

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

- 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
```

Symbol Type

- Symbol is a primitive type for unique identifiers
- Object property keys may be either of string type, or of symbol type
  - Property keys of other types are coerced to strings
- A value of Symbol type can be created using Symbol():

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

- We can also give symbol a description, 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 they have the same description

```
let id1 = Symbol("id");
let id2 = Symbol("id");
alert(id1 == id2); // false
```

Hidden Properties

- Symbols allow us to create “hidden” properties of an object, that no other part of code can occasionally access or overwrite
- For instance, if we want to store an “identifier” for the object user, we can use a symbol as a key for it:

```
let user = { name: "John" };
let id = Symbol("id");

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

- The benefit of using Symbol("id") over a string "id" is that if another script wants to have its own "id" property inside user for its own purposes, then it can create its own Symbol("id"), without causing any conflicts:

```
// ...
let id = Symbol("id");
user[id] = "Their id value";
```

Symbol Type

- If we want to use a symbol in an object literal, we need square brackets:

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

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

  - That’s because we need the value from the variable id as the key, not the string “id”
- Symbols are skipped by for..in loops
- In contrast, Object.assign copies both string and symbol properties

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, e.g.:
  - Symbol.hasInstance
  - Symbol.iterator
  - Symbol.toPrimitive
  - …and so on
- For instance, Symbol.iterative allows us to make an object iterable
  - We'll see its use very soon

Constructor Functions

- The regular {...} syntax allows to create one object
- But often we need to create many similar objects, like multiple users or menu items
- That can be done using constructor functions and the "new" operator
- Constructors are regular functions, that follow two conventions:
  - They are named with capital letter first
  - They should be executed only with "new" operator

```
function User(name) {
    this.name = name;
    this.isAdmin = false;
}

let user = new User("Adam");
alert(user.name); // Adam
alert(user.isAdmin); // false

let user2 = new User("Ann");
alert(user.name); // Ann
```

The new operator

- When a function is executed as new User(...), it does the following steps:
  - A new empty object is created and assigned to **this**
  - The function body executes
    - Usually it modifies this, adds new properties to it
  - The value of this is returned
- In other words, new User(...) does something like:

```
function User(name) {
    // this = {};  (implicitly)

    // add properties to this
    this.name = name;
    this.isAdmin = false;

    // return this;  (implicitly)
}
```

Methods in Constructor

- In the constructor, we can add to **this** not only properties, but methods as well
- For instance, new User(name) below creates an object with the given name and the method sayHi:

```
function User(name) {
    this.name = name;

    this.sayHi = function () {
        alert("My name is: " + this.name);
    };
}

let john = new User("John");
john.sayHi(); // My name is: John
```

Exercise (6)

- Create a constructor function **Calculator** that creates objects with 3 methods:
  - read() asks for two values using prompt and remembers them in object properties
  - sum() returns the sum of these properties
  - mul() returns the multiplication product of these properties
- For instance:

```
let calculator = new Calculator();
calculator.read();

alert("Sum = " + calculator.sum());
alert("Mul = " + calculator.mul());
```

Exercise (7)

- Create a constructor function Accumulator(startingValue)
- Object that it creates should:
  - Store the “current value” in the property value. The starting value is set to the argument of the constructor startingValue.
  - The read() method should use prompt to read a new number and add it to value.
    - In other words, the value property is the sum of all user-entered values with the initial value startingValue.
- Here’s the demo of the code:

```
let accumulator = new Accumulator(1); // initial value 1
accumulator.read(); // adds the user-entered value
accumulator.read(); // adds the user-entered value
alert(accumulator.value); // shows the sum of these values
```

Exercise (8)

- Create a constructor function Calculator that creates “extendable” calculator objects
- First, implement the method calculate(str) that takes a string like "1 + 2" in the format “NUMBER operator NUMBER” (space-delimited) and returns the result. Should understand plus + and minus -.

```
let calc = new Calculator();
alert(calc.calculate("3 + 7")); // 10
```

- Then add the method addOperator(name, func) that teaches the calculator a new operation. It takes the operator name and the two-argument function func(a,b) that implements it. Usage example:

```
let powerCalc = new Calculator();
powerCalc.addMethod("*", (a, b) => a * b);
powerCalc.addMethod("/", (a, b) => a / b);
powerCalc.addMethod("**", (a, b) => a ** b);

let result = powerCalc.calculate("2 ** 3");
alert(result); // 8
```

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.
```

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}"); // 佫, 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
}
```

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
```

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 method **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}`);
}
```

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"
```

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

Exercise (9)

- Write a function checkSpam(str) that returns true if str contains ‘viagra’ or ‘XXX’, otherwise false
- The function must be case-insensitive:

```
alert(checkSpam('buy ViAgRA now')); // true
alert(checkSpam('free xxxxx')); // true
alert(checkSpam("innocent rabbit")); // false
```

Exercise (10)

- Create a function truncate(str, maxlength) that checks the length of the str and, if it exceeds maxlength – replaces the end of str with the ellipsis character "…", to make its length equal to maxlength
- The result of the function should be the truncated (if needed) string
- For instance:

```
alert(truncate("What I'd like to tell on this topic is:", 20)); // "What I'd like to te…"
alert(truncate("Hi everyone!", 20)); // "Hi everyone!"
```

Arrays

- Arrays are used to store lists of related information, e.g., the names of the students in a class, a shopping list, or the grades of your exams
- The values inside an array are called **elements**
- There are two syntaxes for creating an empty array:

```
let arr = new Array();
let arr = []; // more common
```

- We can supply initial elements in the brackets:

```
let fruits = ['Apple', 'Orange', 'Melon'];
```

- An array can store elements of any type

```
// mix of values
let arr = ['Apple', { name: 'John' }, true, function () { alert('hello'); }];
```

Accessing Array Elements

- Array elements are numbered, starting with zero
- We can get an element by its index number in square brackets:

```
let fruits = ['Apple', 'Orange', 'Melon'];

alert(fruits[0]); // Apple
alert(fruits[1]); // Orange
alert(fruits[2]); // Melon
```

- We can replace an element:

```
fruits[2] = 'Pear'; // now ['Apple', 'Orange', 'Pear']
```

- Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ['Apple', 'Orange', 'Pear', 'Lemon']
```

- You can use alert to show the whole array:

```
alert(fruits);
```

Array Length

- The **length** property of an array returns the the number of array elements:

```
let fruits = ['Apple', 'Orange', 'Melon'];
alert(fruits.length); // 3
```

  - The length property automatically updates when we modify the array
- The length is actually not the count of values stored in the array, but the greatest numeric index plus one:

```
let fruits = [];
fruits[123] = 'Apple‘;
alert(fruits.length); // 124
```

- The length property is writable
  - If we increase it manually, nothing happens. But if we decrease it, the array is truncated.

```
let arr = [1, 2, 3, 4, 5];
arr.length = 2; // truncate to 2 elements
alert(arr); // [1, 2]
```

Iterating an Array

- You can cycle through the array items using a for loop over the indexes:

```
let fruits = ['Apple', 'Orange', 'Melon'];

for (let i = 0; i < arr.length; i++) {
    alert(arr[i]); // Apple, Orange, Melon
}
```

- But for arrays there is another form of loop, **for..of**:

```
for (let fruit of fruits) {
    alert(fruit); // Apple, Orange, Melon
}
```

- The for..of doesn't give access to the number of the current element, just its value, but in most cases that's enough

Array as a Queue

- A **queue** is an ordered collection of elements which supports two operations:
  - push appends an element to the end
  - shift gets an element from the beginning, advancing the queue

- Arrays support both operations:

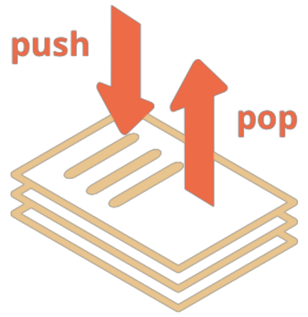
```
let fruits = ['Apple', 'Orange'];
fruits.push('Melon');
alert(fruits); // Apple, Orange, Melon

alert(fruits.shift()); // remove Apple and alert it
alert(fruits); // Orange, Melon
```

- The first element added to the queue will be the first one to be removed
  - This makes the queue a **FIFO** (First-In-First-Out) data structure

Array as a Stack

- There’s another use case for arrays – the data structure named **stack**
- It supports two operations:
  - push adds an element to the end
  - pop takes an element from the end



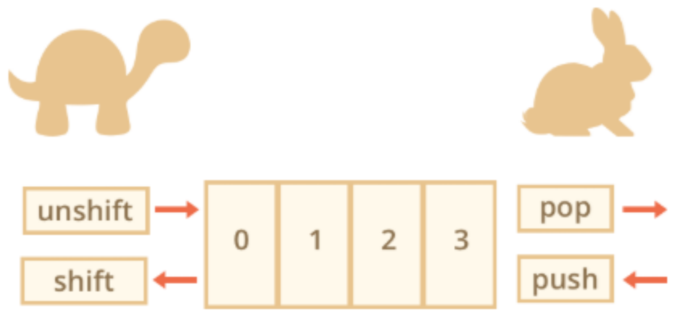
```
let fruits = ['Apple', 'Orange'];
fruits.push('Pear');
alert(fruits); // Apple, Orange, Pear

alert(fruits.pop()); // remove "Pear" and alert it
alert(fruits); // Apple, Orange
```

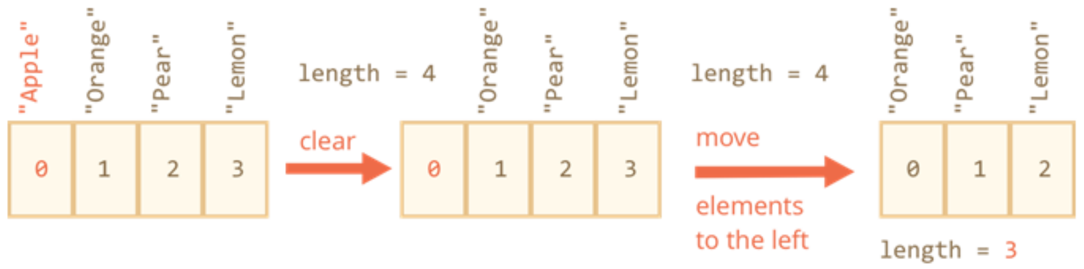
- A stack is usually illustrated as a pack of cards: new cards are added to the top or taken from the top
- For stacks, the latest pushed item is received first
  - This makes the stack **LIFO** (Last-In-First-Out) data structure

Performance

- Methods push/pop run fast, while shift/unshift are slow

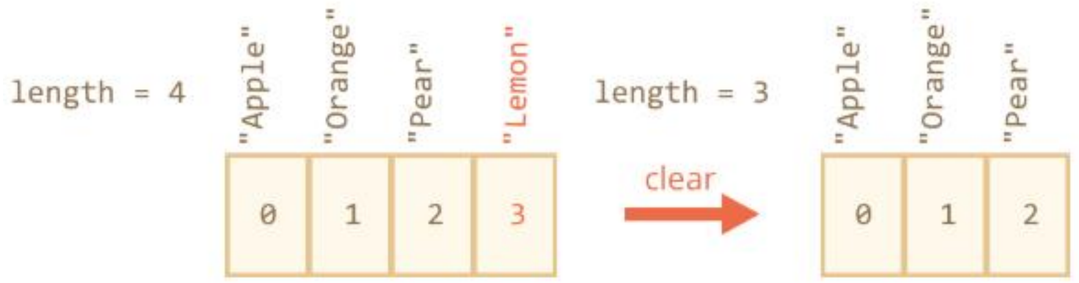


- For example, the shift operation must do 3 things:
  - Remove the element with the index 0
  - Move all elements to the left, renumber them from the index 1 to 0, from 2 to 1 and so on
  - Update the length property



Performance

- On the other hand, push/pop do not need to move anything, because other elements keep their indexes
- To extract an element from the end, the pop() method cleans the index and shortens length:



Multi-Dimensional Arrays

- Arrays can have items that are also arrays
- We can use it for multidimensional arrays, to store matrices:

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
alert(matrix[1][1]); // 5
```

Exercise (11)

- Let’s try 5 array operations:
  - Create an array styles with items “Jazz” and “Blues”
  - Append “Rock-n-Roll” to the end
  - Replace the value in the middle by “Classics”
    - Your code for finding the middle value should work for any arrays with odd length
  - Strip off the first value of the array and show it
  - Prepend Rap and Reggae to the array
- The array in the process:

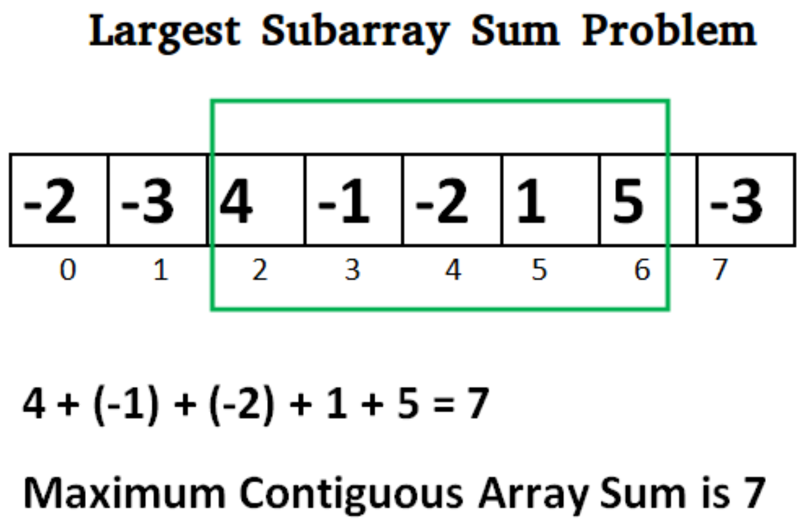
```
Jazz, Blues
Jazz, Bues, Rock-n-Roll
Jazz, Classics, Rock-n-Roll
Classics, Rock-n-Roll
Rap, Reggae, Classics, Rock-n-Roll
```

Exercise (12)

- Write a function sumInput() that:
  - Asks the user for values using prompt and stores the values in the array
  - Finishes asking when the user enters a non-numeric value, an empty string, or presses “Cancel”
  - Calculates and returns the sum of array items

Exercise (13)

- The input is an array of numbers, e.g. arr = [-2, -3, 4, -1, -2, 1, 5, -3]
- Your task is to find the contiguous subarray of arr with the maximal sum of numbers
- Write the function getMaxSubSu(arr) that will find and return that sum


Largest Subarray Sum Problem
-2 -3 4 -1 -2 1 5 -3
0 1 2 3 4 5 6 7
4 + (-1) + (-2) + 1 + 5 = 7
Maximum Contiguous Array Sum is 7