Regular expressions. Memory management

softserve

Agenda

- Regular expression
  - Regular expressions flags
  - Regular expressions methods
  - RegExp in String methods
  - Metacharacters
  - Quantifiers
- Memory management

softserve

Regular expressions

softserve

Main concept

- **Regular expressions (reg exp)** - a formal language for finding and manipulating substrings in text
- Regular expressions date back to the 1950s when they were formalized by Stephen Kleene as a conceptual search pattern for string processing algorithms
- JavaScript is one of the programming languages in which regular expression support is **built directly into the language**
- The result of working with a regular expression can be:

- verification of the presence of the desired sample in the given text;
- definition of a substring of text that is matched to the pattern;
- definition of groups of characters corresponding to individual parts of the sample.

Regular expressions in JavaScript

A regular expression is a sequence of characters that forms a **search pattern**.

When you search for data in a text, you can use this search pattern to describe what you are searching for.

JavaScript has a **RegExp object** defined to work with regular expressions.

There are two syntax options for creating a regular expression:

1) creating a **new RegExp object** using the constructor:

```
let regexp1 = new RegExp('pattern')
```

2) use of **regular expression literals**

```
let regexp2 = /pattern/
```

softserve

Regular expressions flags

Regular expressions may (or may not) have flags that affect the search:

```
let regexp = new RegExp("pattern", "flags");
let regexp = /pattern/flags;
```

i - with this flag, the search is not case sensitive: there is no difference between x and X

g - with this flag, the search searches for all matches, without it, only the first

m - multiline mode

s - turns on the “dotall” mode, at which point . can match linefeed character \n

u - includes full Unicode support

y - search mode for a specific position in the text

RegExp methods. exec()

The **exec()** method searches string for text that **matches regexp**. If it finds a match, it returns an **array of results**; otherwise, it returns **null**.

```
RegExp.exec( string );
```

```
let str = "Data transfer started!";
let regExp = /Data/;
let result = regExp.exec(str);
console.log(result);    // Data

str = "Session complete";
result = regExp.exec(str);
console.log(result);    // null
```

softserve

RegExp methods. test()

The **test()** method searches string for text that matches regexp. If it finds a match, it **returns true**; otherwise, it **returns false**.

```
RegExp.test( string );
```

```
let str = "Data transfer started!";
let regExp = /Data/;
let result = regExp.test(str);
console.log(result);   // true

str = "Session complete";
result = regExp.test(str);
console.log(result);   // false
```

softserve

Sets and ranges [ ]

A few characters in square brackets [ ] mean "search for any character given."

Sets

[xyz] means **any** of the 3 characters: ‘x ',‘ y' or ‘z '.

```
let regExp = /[sk]/;
let str1 = "test message";
let str2 = "another data";
console.log(regExp.test(str1)); // true
console.log(regExp.test(str2)); // false
```

Ranges

The **range** of characters can be set through a **dash**.

```
let regExp1 = /[a-d]/;
let str1 = "test message";
let str2 = "some text";
console.log(regExp1.test(str1)); // true
console.log(regExp1.test(str2)); // false
```

```
let regExp2 = /[6-9]/; // 6,7,8,9
let pin1 = "2145";
let pin2 = "5248";
console.log(regExp2.test(pin1)); // false
console.log(regExp2.test(pin2)); // true
```

softserve

Mix ranges [ ]. Excluding ranges

Groups of characters can be combined with each other to build more complex regular expressions:

```
/[A-E0-3]/.test('DC7748')    // true
/[s-z0-9]/.test('@code1')    // true
```

Excluding ranges are indicated by the caret symbol **^** at the beginning of the range and correspond to **any character except** for those **specified**.

[^ bem] - any character except ‘b', 'e‘ or ‘m'

[^ 0-3] - any character except the number 0,1,2,3

[^ @%] - any character except ‘@’ or ‘% '

softserve

RegExp in String methods. search(), split()

Some String object methods may use regular expressions as a parameter. General syntax:

```
string.method_name(RegExp);
```

The **search()** method finds the **index of the first match** match in a string (otherwise - returns -1):

```
let str = "some text here!";
let re = /tex/;
let result = str.search(re);
console.log(result); // 5
```

! Important limitation: str.search can only return the position of the first match.

The **split()** method can use regular expressions to **split lines**. For example, we divide the application by the symbol “;”:

```
let re = /;/;
let nameList = "Igor;Ostap;Viktor;Iryna";
console.log(nameList.split(re)); // ["Igor", "Ostap", "Viktor", "Iryna"]
```

softserve

RegExp in String methods. match(), replace()

To find all matches in a string, the **match()** method is used. If there are no matches, *null* is returned:

```
let str = "John came home and did homework";
let re = /home/g;
let result = str.match(re);
console.log(result); // ["home", "home"]
```

The replace() method allows you to replace all matches of a regular expression with a specific string. We can use it without regular expressions, but when the **first replace argument is a string, it replaces only the first match**.

```
"My home is a good home!".replace('home', 'car')    // "My car is a good home!"
"My home is a good home!".replace(/home/, 'car')   // "My car is a good home!"
"My home is a good home!".replace(/home/g, 'car')  // "My car is a good car!"
```

! The regular expression must include the "g" flag, otherwise the methods will work with the first match.

softserve

Metacharacters

Metacharacters are characters with a special meaning:

- **\d** matches any digit, equivalent to [0-9]
- **\D** matches any character that’s not a digit, equivalent to [^0-9]
- **\w** matches any alphanumeric character (plus underscore), equivalent to [A-Za-z_0-9]
- **\W** matches any non-alphanumeric character, anything except [^A-Za-z_0-9]
- **\s** matches any whitespace character: spaces, tabs, newlines
- **\S** matches any character that’s not a whitespace
- **\0** matches null
- **\n** matches a newline character
- **.** matches any character that is not a newline char (e.g. \n)

Quantifiers

To specify the **number of repetitions** in regular expressions, you need to add a **quantifier**.

- **+** – matches one or more repetitions of the previous character.
- ***** – matches any number of repetitions of the previous character or its absence.
- **?** – makes the character optional.
- **{n}** – corresponds to the nth number of repetitions of the previous character
- **{n,}** – matches n or more repetitions of the previous character
- **{n,m}** – matches n to m repetitions of the previous character
- **^** – matches the beginning of a line
- **$** – matches end of line

Examples of using metacharacters and quantifiers

+ - match one or more (>=1) items. Same as {1,}

```
let str = "+38(093)-458-22-76";
console.log( str.match(/\d+/g) );  // ["38", "093", "458", "22", "76"]
```

***** - means zero or more. Same as {0,}.

```
console.log( "100 10 1".match(/\d0*/g) ); // ["100", "10", "1"]
```

? – 0 or 1, same as {0,1}

```
console.log( "100 10 1".match(/10?/g) );  // ["10", "10", "1“]
```

{n,m}

```
console.log("My vacation is not 2, but 22 days".match(/\d{2,3}/g) ); // ["22"]
```

^ $

```
/^\d{4}\w{1,3}$/.test('1000PRO') // true
```

Escaping a special characters

To search for special characters [\ ^ $. | ? * + (), we need to add in front of them \ ("**escape** them").

Suppose we want to find a literal point. Not "any character", but a dot.

To use a special character as normal, add a backslash to it : \\.

```
console.log( "Paragraph 2.3".match(/\d\.\d/) );  // 2.3
console.log( "Paragraph 223".match(/\d\.\d/) );  // null
```

softserve

Groups, logical OR

Using parentheses, you can create **groups of characters**: **(...)**

If you apply a quantifier to a group, it will **act on the entire group**, and not just on one previous character:

```
let re1 = /(ha)+/gi;
let re2 = /ha+/gi;
let str = "Hahaaaha";
console.log(str.match(re1)); // ["Haha", "ha"]
console.log(str.match(re2)); // ["Ha", "haaa", "ha"]
```

To indicate an **OR condition**, use the symbol **|** :

```
let re = /html|css|java(script)?/gi;
let str = "Java first appeared, then HTML, then JavaScript";
console.log( str.match(re) ); // ["Java", "HTML", "JavaScript"]
```

softserve

Multiline mode

m: **enables multiline mode**. It only affects the behavior of ^ and $.

In multi-line mode, they mean not only the beginning/end of the text, but also the beginning/end of **each line** in the text.

```
let str = `1) Lviv
2) Rivne
3) Ternopil`;
console.log( str.match(/^\d/gm) ); // ["1", "2", "3"]
 Without m:
let str = `1) Lviv
2) Rivne
3) Ternopil`;
console.log( str.match(/^\d/g) ); // ["1"]
```

softserve

Form validation with RegExp

Simple validation can be implemented by using the appropriate attributes in the form elements. In particular, you can apply the **type** attribute

```
<form>
   <input type="number" name="age" min="1" max="125" placeholder="Enter your age">
</form>
```

However, applying such validation may be insufficient, in which case you can validate using regular expressions using the **pattern** attribute. The pattern attribute works with the following input types: text, search, url, tel, email, and password:

```
<form>
   <input type="text" name="street" pattern="[A-Za-z]+\s?\d+" minlength="4" maxlength="25">
</form>
```

softserve

Memory management

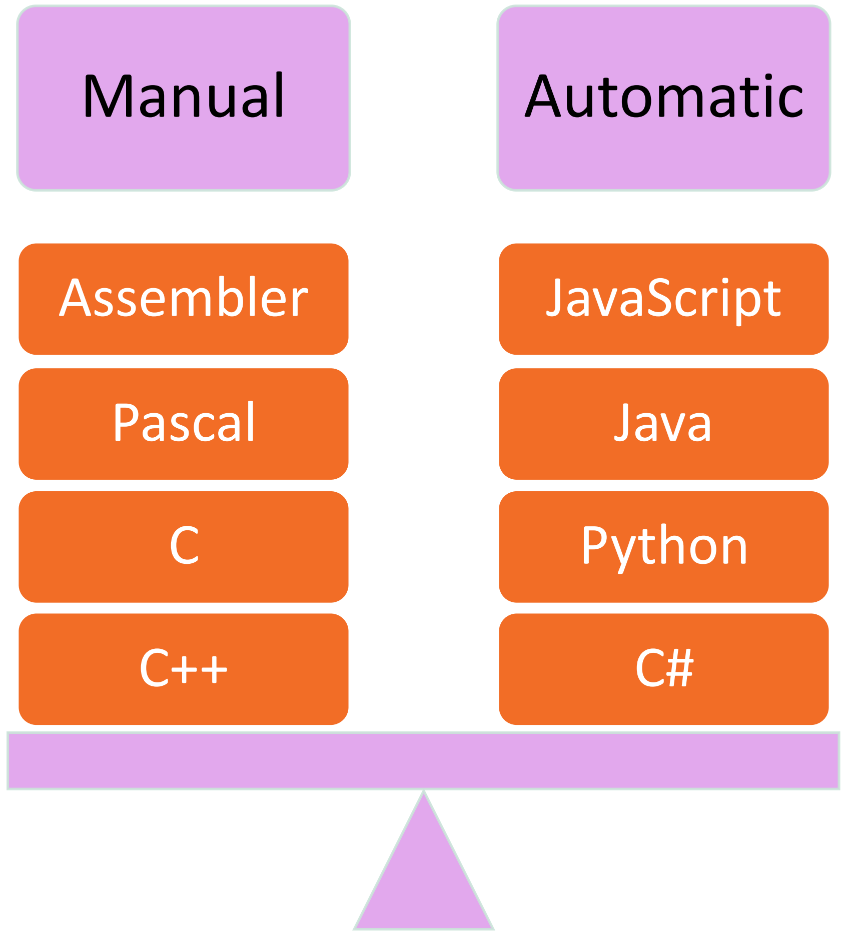
softserve

Main concepts

Languages, like C, have low-level memory management primitives such as *malloc()* and *free()*. These primitives are used by the developer to explicitly allocate and free memory from and to the operating system.

At the same time, JavaScript allocates memory when things (objects, functions, etc.) are created and "**automatically**" frees it up when they are not used anymore, a process called **garbage collection**.

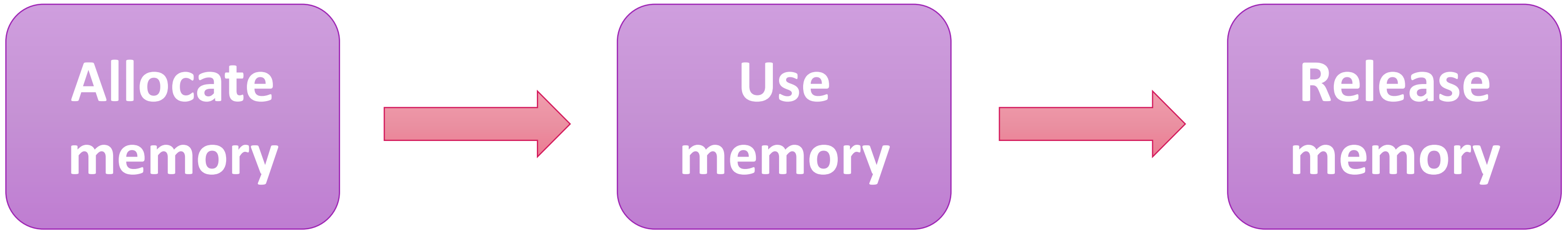
This seemingly "automatical" nature of freeing up resources is a source of confusion and gives JavaScript (and other high-level-language) developers the false impression they can choose not to care about memory management. **This is a mistake.**



softserve

Memory life cycle

No matter what programming language you’re using, memory life cycle is pretty much always the same:



1. **Allocate memory** — memory is allocated by the operating system which allows your program to use it. In low-level languages (e.g. C) this is an explicit operation that you as a developer should handle. In high-level languages, however, this is taken care of for you.
2. **Use memory** — this is the time when your program actually makes use of the previously allocated memory. **Read** and **write** operations are taking place as you’re using the allocated variables in your code.
3. **Release memory** — now is the time to release the entire memory that you don’t need so that it can become free and available again. As with the **Allocate memory** operation, this one is explicit in low-level languages.

1) Memory allocation

In order not to bother the developer with memory allocations, JavaScript automatically allocates memory when the initial values are declared.

```
var n = 123; // allocates memory for a number
var s = 'azerty'; // allocates memory for a string

var o = {
  a: 1,
  b: null
}; // allocates memory for an object and contained values

// (like object) allocates memory for the array and contained values
var a = [1, null, 'abra'];

function f(a) {
  return a + 2;
} // allocates a function (which is a callable object)

// function expressions also allocate an object
someElement.addEventListener('click', function() {
someElement.style.backgroundColor = 'blue'; }, false);
```

softserve

1) Memory allocation

Memory allocation through function calls

Some function calls allocate memory to an object.

```
var d = new Date(); // allocates a Date object

var e = document.createElement('div'); // allocates a DOM element
```

Some methods allocate memory for new values or objects:

```
var s = 'azerty';
var s2 = s.substr(0, 3); // s2 is a new string
// Since strings are immutable values,
// JavaScript may decide to not allocate memory,
// but just store the [0, 3] range.

var a = ['ouais ouais', 'nan nan'];
var a2 = ['generation', 'nan nan'];
var a3 = a.concat(a2);
// new array with 4 elements being
// the concatenation of a and a2 elements.
```

softserve

2) Using memory

Using the allocated memory in JavaScript basically, means reading and writing in it.

This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

softserve

3) Release memory

Most of the memory management issues come at this stage.

The hardest task here is to figure out when the allocated memory is not needed any longer. It often requires the developer to determine where in the program such piece of memory is not needed anymore and free it.

High-level languages embed a piece of software called **garbage collector** which job is to track memory allocation and use in order to find when a piece of allocated memory is not needed any longer in which case, it will automatically free it.

Unfortunately, this process is an approximation since the general problem of knowing whether some piece of memory is needed is undecidable (can’t be solved by an algorithm).

3) Release memory. Garbage collector

Most garbage collectors work by collecting memory which can no longer be accessed, e.g. all variables pointing to it went out of scope. That’s, however, an under-approximation of the set of memory spaces that can be collected, because at any point a memory location may still have a variable pointing to it in scope, yet it will never be accessed again.

References

The main concept that algorithms for garbage collectors rely on is the **concept of references**. In the context of memory management, an object refers to another object if accessed (explicitly or implicitly). For example, a JavaScript object has a reference to its prototype (implicit link) and to its property values (explicit link).

In this context, the concept of "object" is broader than conventional JavaScript objects, and also contains the scope of functions (or global scope).

softserve

Reference-counting garbage collection

This is the most primitive garbage collection algorithm. This algorithm narrows the problem of determining whether an object is still needed to determine whether other objects still have references to that object. An object is called "garbage", or it can be collected if it has zero references.

Reference-counting garbage collection

```
var x = {
  a: {
    b: 2       // 2 objects are created. One is referenced by the other as one of its properties.
  }             // The other is referenced by virtue of being assigned to the 'x' variable.
};              // Obviously, none can be garbage-collected.

var y = x;      // The 'y' variable is the second thing that has a reference to the object.
x = 1;          // Now, the object that was originally in 'x' has a unique reference
                //   embodied by the 'y' variable.

var z = y.a;    // Reference to 'a' property of the object.
                //   This object now has 2 references: one as a property,
                //   the other as the 'z' variable.

y = 'mozilla';  // The object that was originally in 'x' has now zero
                //   references to it. It can be garbage-collected.
                //   However its 'a' property is still referenced by
                //   the 'z' variable, so it cannot be freed.

z = null;       // The 'a' property of the object originally in x
                //   has zero references to it. It can be garbage collected.
```

softserve

Memory Leaks in JavaScript

In essence, memory leaks can be defined as memory that is not required by an application anymore that for some reason is not returned to the operating system or the pool of free memory.

Types of Common JavaScript Leaks

1: Accidental global variables

2: Forgotten timers or callbacks

3: Out of DOM references

4: Closures

Memory Leaks in JavaScript. Finding

1. Check initial memory state.
2. Execute the suspicious action that should not increase memory.
3. Check memory again.
4. If memory is significantly higher, There is a **leak**.
5. Use Profiles and Timeline to track what's happening:

- Amount of DOM nodes;
- Retained memory

Useful links

https://www.debuggex.com/cheatsheet/regex/javascript

https://regexr.com/

https://flaviocopes.com/javascript-regular-expressions

https://learn.javascript.ru/regular-expressions

https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/

softserve

THANKS

softserve