Л Е К Ц И Я №5

Web технологии

I. ВЪВЕДЕНИЕ

Създаването на Web услуги предполага използването на множество технологии. Една част от тях се използват с цел изграждане на потребителския интерфейс и генериране на заявки към мрежови или облачни услуги (**front-end** програмиране), а друга част се използват с цел обслужване на тези заявки и реализиране на бизнес логиката на услугата (**back-end** програмиране). Основните технологии, чрез които се реализира front-end частта от услугата, са HTML, CSS и JavaScript. Тези технологии функционират под управлението на Web браузър. Тяхното предназначение е както следва:

1. Hyper Text Markup Language (HTML). HTML е език чрез който се указва на браузъра какво да визуализира в изгледа на браузъра. Всеки HTML компонент се описва с Document Object Model (DOM) обект. Необходимият HTML код е записан в един или множество файлове с разширение html. Тези файлове съдържат код, който е съставен от етикети (tags). На настоящият етап се работи с версия 5 на HTML – HTML5.
2. Cascading Style Sheets (CSS). Основната цел на CSS кода е задаване на стилово форматиране на DOM обектите. Стилово форматиране може да се реализира и чрез HTML код, но това не трябва да се използва. За да се реализира по-лесно стиловото форматиране и за да може лесно да се модифицира, то трябва да се реализира само чрез CSS код, който е съсредоточен в един или множество файлове с разширение css. Свързването на CSS код с HTML код се реализира чрез един или няколко HTML етикети с име link. На настоящият етап се работи с версия 3 на CSS – CSS3.
3. JavaScript. Това е скрипт език, който по своята същност е хибриден – поддържа се обектно-ориентирано и функционално програмиране. Неговата основна цел е изпълнение на програмен код (логика) от ниво браузър. За да е възможно това е необходимо всеки браузър да има вграден интерпретатор на JavaScript код. Чрез JavaScript код имате достъп до свойствата и методите на DOM обектите и по този начин можете да ги манипулирате. Можете да генерирате различни формати заявки към Web или облачно-базирани услуги (XML, JSON и др.), както и да обработвате отговорите, получени от тези услуги. Необходимият код е добре да бъде съсредоточен в един или множество файлове с разширение js. Той се зарежда в паметта и свързва с необходимия HTML код чрез етикет с име script. На настоящият етап се работи с основно с версии 7 и 8 на JavaScript.

Нека да обобщим: Чрез HTML кода се задава какво трябва да се визуализира в изгледа на браузъра; чрез CSS се реализира стилово форматиране на това, което се вижда, а чрез JavaScript се интегрира логика, която се изпълнява от страна на клиента.

II. HYPER TEXT MARKUP LANGUAGE

Всеки HTML документ се състои от **етикети**. Етикетите имат имена и трябва да са в ъглови скоби < >. Освен име, всеки етикет може да има един или няколко атрибута. **Атрибутите** следват след името на етикета и имат име и стойност. Връзката на името и стойността на етикета е символ =. Стойността на атрибутите са в кавички “ ”. Съществуват два вида етикети в зависимост от това дали в тях може да има други етикети или не. Етикетите, които могат да съдържат други етикети, са етикети с тяло. Началото на тялото е етикет, който започва с името му, а краят – етикет, който започва със символ / след който следва името на етикета. Етикетите без тяло завършват със символ /. В кой етикет какви етикети може да има се задава от CSS стандарта, който се поддържа от W3C.

На Фиг.5.1 е показана структура на HTML5 документ.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Структура на HTML документ</title>
         <!-Задаване на необходимата кодова таблица за кирилица -->
        <meta charset="utf-8">
        <!—Зареждане на необходимия JavaScript код -->
        <script type="text/javascript" src="myCode.js"></script>
        <!—Зареждане на необходимия CSS код -->
         <link rel="stylesheet" type="text/css" href="myStyle.css" />
    </head>

    <body>
        Тяло на етикет body – съдържа HTML етикети, които задават какво се
         визуализира в изгледа на браузъра (потребителски интерфейс).
    </body>
```

Фиг. 5.1. Структура на HTML документ

Всеки HTML5 документ започва с етикет <!DOCTYPE>. Чрез него се задава, че документът съдържа HTML код. В тялото на етикет <html> се съдържат всички останали етикети. Структурата на документа се състои от две основни секции – *заглавен блок* и *тяло*. Заглавният блок се задава чрез началото и края на етикет <head>. В него може да има множество други етикети, по-важните от които са следните:

- <title> - име на страницата. Визуализира се в title bar на браузъра.
- <meta> - служи за предаване на мета информация към браузъра, например: Каква кодова таблица да се използва при визуализиране на текста от HTML страниците? Това се задава чрез атрибут charset. В конкретният пример е зададена кодова таблица UTF-8. Това ще ни позволи визуализиране на текст не само на латиница, но и на кирилица. Чрез meta етикети може да предавате информация не само към браузъра, но и към търсещите машини, които имат за задача да индексират вашите страници.
- <script> - служи за зарежда не на JavaScript код от файл, който се задава чрез атрибут src.
- <link> - служи за зареждане на CSS код от файл. Името на файла е стойността на атрибут href.

Етикет <body> формира тялото на HTML документа. В него се съдържат HTML етикети чрез които се реализира потребителския интерфейс. Компонентите, които могат да изградят един потребителски интерфейс, са разнообразни, например: текст, картинки, вградено видео, бутони, радио-бутони, check-боксове, плъзгачи и много други.

Структурирането и самото съдържание в тялото на документа е много важно за индексирането на вашия сайт от ***търсещите машини***. Текстовата информация от тялото на документа трябва да съответства семантично на описанието на сайта чрез мета данните от заглавния блок на документа (description, keywords). Задайте чрез мета таг с име robots, че сте съгласни вашата страница да се индексира. Създайте и файл **robots.txt** в root папката на Web проекта. Той ще бъде прочетен от търсещите машини. Целта е да им помогнете да индексират по-бързо съдържанието на сайта ви. Този файл съдържа информация за търсещите машини във формат свойство: стойност, например:

```
Allow: /
Sitemap: http://hostname.domain/sitemap.xml
```

В този случай се указва, че има файл, който съдържа карта на сайта - **sitemap.xml**. Този файл информира търсещата машина за всеки важен според вас документ:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <urlset
3        xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
4        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5        xsi:schemaLocation="http://www.sitemaps.org/schemas/sitemap/0.9
6              http://www.sitemaps.org/schemas/sitemap/0.9/sitemap.xsd">
7  <url>
8    <loc>http://www.kst.tugab.bg/</loc>
9    <lastmod>2020-09-04T11:58:44+00:00</lastmod>
10   <priority>1.00</priority>
11 </url>
12 <url>
13   <loc>http://www.kst.tugab.bg/redirect.php?name=news.php</loc>
14   <lastmod>2020-09-04T11:58:44+00:00</lastmod>
15   <changefreq>weekly</changefreq>
16   <priority>0.80</priority>
17 </url>
```

Фиг. 5.2. Примерно съдържание на файл sitemap.xml

Дори и да нямате файл sitemap.xml, всяка търсеща машина има възможности за създаване на тази карта автоматично. В този случай обаче не може да укажете важността на различните документи от сайта ви и колко често променяте съдържанието им. Последното е важно, за да се прецени кога търсещата машина да „посети" сайта ви отново. Търсещите машини могат да отчетат тази ваша информация, но могат и да я игнорират, ако преценят, че тя е малко вероятна. Търсещите машини „обичат" информацията в тялото да е добре структурирана. За целта е добре да използвате етикети header, footer, main и section (виж Фиг. 5.3) Този начин на структуриране на информацията ще ви позволи и създаване на **Single Page Apps** (SPA), без да е необходимо за това да използвате специална библиотека. При този тип Web приложения се работи само с една HTML страница, а съдържанието от тялото ѝ се променя динамично чрез JavaScript код. При конкретната структура на документа в даден момент от време може да показвате само една от всички секции, а останалите да са скрити.

```html
<html>
    <head>
        <title>Заглавие на страницата</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="description" content="Описание на страницата ..."/>
        <meta name="keywords" content="Ключови думи, описващи страницата"/>
        <meta name="robots" content="index, follow"/>
        <link rel="stylesheet" type="text/css" href="css/style.css"/>
        <script src="libs/jquery-3.1.1.min.js"></script>
        <script src="js/code.js"></script>
    </head>
    <body>
        <header>Информация за началото на блока с информация</header>
        <main>
            <section id="sec-1">
                Информация от Секция 1
            </section>
            <section id="sec-2">
                Информация от Секция 2
            </section>
            <section id="sec-3">
                Информация от Секция 3
            </section>
        </main>
        <footer>Информация за края на блока с информация</footer>
    </body>
</html>
```

Фиг. 5.3. Примерна структура на тялото на HTML документ

III. CASCADING STYLE SHEETS

Стиловото форматиране на DOM обектите може да се реализира чрез:

1) Специфични атрибути към различните HTML етикети;
2) Чрез етикет <style>, който се интегрира в HTML документа и
3) Чрез Cascading Style Sheet (CSS) код.

Единственият препоръчителен начин за стилово форматиране е чрез CSS код, записан в един или няколко файла. Всеки файл съдържа поредица от CSS правила. Всяко правило има синтаксис, показан на Фиг. 5.4.

```
селектор {
    свойство: стойност; // дефиниция 1
    свойство: стойност; // дефиниция 2
}
p {
    font-size: 14px;        // размер на шрифта 14 пиксела
    font-weight: bold;      // удебелени символи
    margin: 0px;            // Без отстъп спрямо останалите съседни обекти
    padding-top: 10px;      // Горен отстъп на текста от рамката на елемента
    text-align: center;     // Центриране на текста
    color: #00FF00;         // Задаване на цвят на текста (100% зелено)
}
```

Фиг. 5.4. Синтаксис на CSS

Всяко CSS правило има следните компоненти:

1. **Селектор** – указва за кого се отнася правилото. Селекторите най-често се свързват с един, няколко или всички DOM обекти. Например, ако желаем CSS правилото да се отнася за обектите от етикет <body>, то името на селектора трябва да е body. Когато правилото трябва да се отнася за група еднотипни (създадени с един и същ етикет) DOM обекти е необходимо да се укаже, че те са от един и същ клас. Това се реализира чрез атрибут class. Стойността на този атрибут е името на селектора, който трябва да се използва в CSS кода. За да се знае, че правилото се отнася за група елементи от даден клас, името на селектора трябва да започва със символ точка. Ако правилото трябва да се отнася само за един обект е необходимо задаване на неговия идентификатор чрез атрибут id. Стойността на идентификатора е името на селектора. В този случай това име в CSS кода трябва да се предхожда от символ диез (#). Ако един селектор трябва да се отнася за няколко типа етикети, те трябва да се разделят чрез символ запетая. Възможностите за задаване на селектор зависи от версията на CSS. При CSS3 се предоставя възможност за избор на над 20 вида синтаксис за селектора, по-често използваните от които са описани в Табл. 5.1.

Табл. 5.1. Основни типове селектори при CSS3

Синтаксис	Пример	Предназначение
*	*	Отнася се за всички DOM обекти
име	body	Отнася се за етикет body
.име	.p22	Отнася се за всички обекти от клас с име p22
#име	#p33	Отнася се за обект с идентификатор (id) с име p33
име, име	p, div	Отнася се за всички обекти, създадени чрез етикети p и div
име име	div p	Отнася се за всички обекти, създадени чрез етикет p, който е в тялото на етикет div.
име > име	div>p	Отнася се за всички обекти, създадени чрез етикет p, за който родител е div етикет.
име+име	div+p	Отнася се за всички обекти, създадени чрез етикет p, който следва веднага след div етикет.
[име=стойност]	[type=text]	Отнася се за обект с име на атрибут type и стойност на атрибута text.

2. **CSS декларации**. Всяко правило има тяло, което съдържа блок от декларации. Началото и краят на тялото се задава чрез фигурни скоби { }. Всяка декларация се състои от име на свойство, след което следва символ : . След този символ е стойността на свойството. Всяка декларация завършва със символ ; . Стойността за всяко свойство може да е низ, число, комбинация от число и символи или се връща от функция. Например, задаването на цвят може да се реализира по множество начини:

- Задаване на цвята чрез неговото име като низ: color: red;
- Задаване на цвета чрез шестнадесетичните (0-FF) стойности (8 бита) за червения (R), зеления (G) и синия (B) цвят: color: #FF0000; // 100% червено.
- Задаване на цвета чрез десетичните (0-255) стойности на RGB. За целта се използва функция rgb, например: color: rgb(255,0,0); // 100% червено.
- Задаване на цвета в RGB формат и неговата прозрачност. За целта се използва функция rgba, например:

```
color: rgba(255,0,0,0.7); // 100% червено и 70% прозрачност.
```

На Фиг. 5.5 е показано съдържанието на CSS файл, който съдържа различни по тип CSS правила.

```css
* {
    margin: 0;
    padding: 0;
}
body{
    font-family: 'Open Sans', sans-serif;
    color: #000050;
    height: 100%;
}
main > section {
    padding: 20px 5px;
    background: #cccccc;
}
#sec-1, #sec-2, #sec-3 {
    color: #000070;
    text-align: center;
    padding: 10px;
    transition: 0.5s ease-in;
}
header, footer {
    color: white;
    background: #2C6399;
    padding: 10px 10px;
    text-align: center;
}
```

Фиг. 5.5. Примерно съдържание на CSS файл

IV. JAVASCRIPT

За рождена дата на JavaScript се приема 1995 година, когато Netscape Corp. анонсира Mocha (по-късно LiveScript) като скрипт език за браузъра Netscape Navigator 2.0. По-късно (декември 1995) LiveScript се преименува на JavaScript. В основата на тази разработка е Брендън Айк, който е основател на проекта Mozilla, фондация Mozilla (създадена да поддържа проекта Mozilla) и Mozilla Corp., която разработва и разпространява браузъра Mozilla Firefox.

Програмният език JavaScript е интерпретаторен език от високо ниво. Както повечето скрипт езици и JavaScript поддържа ***динамично типизиране*** – типа на данните се определя в момента на инициализацията им (задава им се стойност). Той се базира на ECMAScript (ES), който през 1997 е стандартизиран (ECMA-262) като език за създаване на Web приложения от страна на клиента. В момента се работи основно с ES6+. JavaScript е хибриден език, прототипно-базиран, обектно ориентиран и поддържа функционално програмиране. Има вградени приложни програмни интерфейси (API) за работа с текст, масиви, списъци, регулярни изрази и др.

Програмният език JavaScript е създаден за разработване на Web приложения. Чрез него се пише основно *front-end* код – програмен код, който се интерпретира от Web браузър. Поради тази причина всеки браузър има вграден JavaScript интерпретатор. Това е основният начин за изпълнение на някаква логика под управлението на браузър. На съвременният етап JavaScript намира много по-широко приложение, например:

1. Създаване на *back-end* код – програмен код, който се изпълнява от страна на Web сървър. Чрез този код се реализира логиката на проекта, включително и достъпа до бази от данни. Типичен пример за такъв код е $Node.js$ -мулти-платформена среда (Microsoft Windows, Linux, OS X) за създаване на сървърни и мрежови приложения на JavaScript. В основата на $Node.js$ е V8 engine на Google. V8 компилира JavaScript кода директно до машинен код, който след оптимизация по бързодействие се изпълнява от машината на която е стартиран браузъра. Част от най-използваните сървъри за управление на $NoSQL$ бази данни като $MongoDB$ и $CouchDB$ използват https интерфейс за генериране на заявки към базата данни. Основна причина за използване на JavaScript като back-end код е неговата платформена независимост и възможности за писане на бърз асинхронен код.
2. Крос-платформи за създаване на *мобилни приложения* за различни мобилни операционни системи (Android, iOS, Windows 8, WebOS и др.) на базата на HTML5 и JavaScript. Съществува голямо разнообразие от подобни платформи, най-широко използваните от които са следните: $Sencha$, $PhoneGap$ ($Apache\ Cordova$), $Unity\ 3D$, $Titanium$, $Intel\ XDK$ и др.
3. Мобилни операционни системи, базирани на HTML5 и JavaScript. Ако трябва да се създават мобилни приложения на JavaScript за мобилни устройства е по-добре тази функционалност да е част от самата мобилна операционна система, отколкото да се използва някоя от крос-платформените програмни рамки (frameworks). Подобна операционна система е $Tizen$ на Samsung.
4. Създаване на *desktop* платформено-независими приложения. За целта, най-често се използват библиотеки $Electron.js$ и $NW.js$. Поддържаните операционни системи са Windows, OS X и Linux.
5. Създаване на програмния код за *вградени* ($embedded$) системи. Все повече микроконтролери и „компютри на чип" позволяват писането на програмния код да се реализира директно на JavaScript. Това се реализира чрез специализирани библиотеки или програмни рамки. Апаратните модули, които позволяват използване на JavaScript, са $Arduino$, $Raspberry\ Pi$, $Tessel$, $Samsung\ ARTIK$ и др.

Когато дадено приложение се разработва изцяло на JavaScript се казва, че се използва концепция „$Full-stack\ JavaScript$". Стекът от технологии включва:

- Потребителски интерфейс и логика от страна на клиента: HTML5, CSS3, JavaScript, JavaScript front-end библиотеки ($Angular$, $React$, Vue и др.).
- Бизнес логика на приложението: изцяло JavaScript програмни обекти.
- Слой данни: $MongoDB$, $MariaDB$, $Redis$ и др.
- Сървър: засега най-често $Node.js$.
- Програмни рамки за реализиране на $RESTful$ интерфейс със сървъра: $Express.js$, $Socket.io$ и $Total.js$.

Програмният код на JavaScript се интегрира директно в HTML страниците чрез етикет $script$ или чрез текстов файл с разширение js. Следователно, за въвеждането на изходния JavaScript код може да използвате кой да е *текстов* редактор. На практика се работи с on-line или off-line развойни среди (IDE). Основният недостатък на on-line развойните среди е невъзможността да ги използвате, ако нямате достъп до Интернет или мрежовата връзка в даден момент е много бавна. Алтернативата е да използвате off-line развойна среда, която трябва да инсталирате на вашия компютър. Най-често използваните развойни среди, които можете да използвате за тестване на HTML, CSS и JavaScript код, са: $NetBeans$, $Eclipse$ и $Microsoft\ Visual\ Studio$.

4.1. Ключови думи в JavaScript

Ключовите думи в един програмен език се използват за служебни цели и не могат да се използват с цел именуване на променливи, константи, обекти и класове. Трябва да се прави разлика между *ключови* и *резервирани* думи. Последните са ключови думи, които ще се използват при някоя от бъдещите версии на езика. Ключовите и резервни думи при ES (JavaScript) са описани в Табл. 5.2.

Табл. 5.2. Ключови и резервни думи в ES (JavaScript)

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const	continue
debugger	default	delete	do
double	else	enum	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

* Тези ключови думи могат да се използват само при ES6+.

4.2. Типизация на данните в JavaScript

В JavaScript всички данни се декларират чрез ключови думи $\boldsymbol{var}$ или $\boldsymbol{let}$ (ES6+), без да се задава техния тип явно. Вътрешно, след инициализация на данните, те се причисляват към един от следните типове с които JavaScript работи:

- *Числа*;
- *низове* (*текст*);
- *булеви данни*;
- *функции*;
- *обекти*.

Програмно можем да определим типа на данните чрез използване на ключова дума (оператор) $typeof$. Синтаксисът е „$typeof$ *данна*”. Този израз връща вътрешния тип на *данна*.

4.2.1. Низове

Типът на низовете е $string$. Те са текст на някакъв естествен език. При JavaScript низовете се дефинират чрез заграждане на текста в ***кавички*** или ***апостроф***:

```
var низ1 = "Технически Университет - Габрово";
var низ2 = 'ул. "Х. Димитър" 4';
var низ3 = "ул. \"Х. Димитър\" 4";
```

Когато в даден низ се налага да има кавички, например име на улица, той трябва да започва с апостроф (*низ*2) или за вътрешните кавички да се укаже, че това е кода на символа кавичка (\"), а не начало и край на низ (*низ*3). При JavaScript е възможно обединяване на низове в един низ чрез оператор +. Когато към един низ се добави число, JavaScript не връща грешка, а приема, че числото трябва да се преобразува до низ и след това да се добави до другия низ. В JavaScript има множество вградени методи за работа с низове. По-често използваните от тях са описани в Табл. 5.3.

Табл. 5.3. Основни методи, които се използват при работа с низове

Метод	Предназначение	Синтаксис
toUpperCase	Преобразува символите на низ от малки до главни.	низ.toUpperCase()
toLowerCase	Преобразува символите на низ от главни до малки.	низ.toLowerCase()
charAt	Връща символа от низ, който е на указана позиция (от началото на низ) в него.	низ.charAt(позиция)
charCodeAt	Връща кода (Unicode) на символа в низ, който е на указана позиция (от началото на низ) в него.	низ.charCodeAt(позиция)
indexOf	Търси *първият* съвпадащ низ в низ от *началото* към *края*. Връща индекса от който започва намерения низ и -1, ако низа не е намерен. Може да зададете позицията от която да започне търсенето като втори параметър.	низ.indexOf(„търсен низ”, позиция)
lastIndexOf	Търси *последният* съвпадащ низ в низ от *началото* към *края*. Има аналогичен синтаксис на indexOf.	низ.lastIndexOf(„търсен низ”, позиция)
search	Търси *първият* съвпадащ низ в низ от *началото* към *края*. Има аналогичен синтаксис на indexOf. Не може да се задава позицията от която започва търсенето, но могат да се използват *регулярни изрази*.	низ.search(„търсен низ”)
slice	Връща част от низа при зададени начален и краен индекс. Ако се използва само първият параметър – връща низ от указаната позиция до края на низа. Ако индексът е отрицателно число - адресирането е от края на низа.	низ.slice(начален_индекс, краен_индекс)
substring	Връща част от низа при зададени начален и краен индекс. Не поддържа отрицателни индекси.	низ.substring(начален_индекс, краен_индекс)
substr	Връща част от низа при зададени начален индекс и брой символи (дължина на низа).	низ.substr(начален_индекс, дължина)
replace	Заменя низ (*низ*1) от даден низ с друг низ (*низ*2). Изходния низ не се променя. Поддържа използването на регулярни изрази (първи параметър). По подразбиране се заменя само първият срещнат низ.	низ.replace(„низ1”, „низ2”)
concat	Слепване на низ към низ. Изпълнява функциите на оператор +. Може да слепва масиви от низове. Не променя изходния низ.	низ.concat(нов_низ);
split	Преобразува низ до масив от низове. Изисква задаване на разделител, чрез който определя кои да са елементите на масива. Разделителят може да е кой да е символ или низ.	низ.split(разделител);

4.2.2. Масиви

Масивите са данни от тип $object$. В JavaScript масивът е *обект*, който може да съдържа множество (един или повече) елементи. Елементите могат да бъдат от един и същи тип, (числа, низове, обекти) или от различни типове. Масивите се декларират подобно на променливите, но след оператор = следват стойностите на масива, заградени в квадратни скоби [ ], например:

```
var масив1 = [];                          // празен масив
var масив2 = [1, 2, 3, 4, 5, 6];          // масив от числа
var масив3 = ["а", "б", "в"];             // масив от низове
var масив4 = [1, "а", 2, "б", "в", 3];    // масив от числа и низове
var масив5 = [масив2, масив3, масив4];    // масив от обекти (масиви)
```

За да адресирате даден елемент от масив трябва да използвате следния синтаксис:

```
имеНаМасива[индекс_на_елемента]
```

Първият елемент от даден масив има индекс 0, а последният – броят на елементите – 1. Например $масив2[3]$ ще върне стойност 4. Можете да получите броя на елементите в един масив, без да ги броите, чрез свойство $length$, например:

```
var n = масив2.length;
```

Списък на по-често използваните методи при работа с масиви и тяхното предназначение е описано в Табл. 5.4.

Табл. 5.4. Основни методи, които се използват при работа с масиви

Метод	Предназначение	Синтаксис
push	*Добавя* нов елемент в *края* на масива	масив.push(елемент)
pop	*Изтрива* елемента в *края* на масива	масив.pop()
unshift	*Добавя* нов елемент в *началото* на масива	масив.unshift(елемент)
shift	*Изтрива* елемента в *началото* на масива	масив.shift()
concat	Слепва два масива (добавя масив в края на масив)	масив.concat(нов_масив)
toString	*Преобразува* масив в *низ*. Елементите на масива са разделени със запетая.	масив.toString()
join	*Преобразува* масив в *низ*. Чрез параметър към метода може да се зададе какъв да бъде *разделителя* между елементите на масива. Без параметри методът функционира като $toString$	масив.join() масив.join(" ") масив.join("\n")
splice	*Вмъкване*, *заместване* или *изтриване* на елемент(и). Могат да се предават до 3 параметъра: 1) позиция, за която се отнася действието; 2) колко елемента, след тази позиция да бъдат премахнати; 3) ако има вмъкване – стойността на новия елемент.	*Вмъкване*: масив.splice(3,0, елемент) *Заместване*: масив.splice(3,1, елемент) *Изтриване*: масив.splice(3,1)
slice	Извлича масив от масив. Елементите на новия масив зависят от параметрите, които се предават: 1) позиция на началният елемент; 2) позиция на крайния елемент+1.	var нов_масив = масив.slice(3,5)
indexOf	Проверка за наличие на елемент в масив. Ако елементът съществува, методът връща неговата позиция. В противен случай връща -1.	масив.indexOf(елемент)
reduce	Вика функция, която се предава на метода толкова пъти, колкото са елементите в масива	масив.reduce(функция)

4.2.3. Функции

Функциите са данни от тип $\boldsymbol{function}$. Въпреки, че типът на функциите е $function$, в JavaScript те са *обекти*, по-точно обекти от тип $function$. За всеки обект-функция се поддържат вътрешни свойства и методи. Например, свойство $arguments$ е масив, който съдържа предаваните на функцията параметри. След ключова дума $function$ следват името на функцията (*име*) и скоби (). Интерпретаторът разпознава функциите именно по тези скоби. Имената на функциите могат да съдържат букви, цифри и символи като _ и $. Програмният код от тялото на функцията трябва да бъде във фигурни скоби { }.

На кода от тялото на функцията могат да се предават или да не се предават параметри (аргументи). Ако се предават параметри, техните имена се описват в скобите, разделени със запетая един от друг. Няма ограничение за броя на предаваните параметри. Стойностите на параметрите са необходими за формиране на логиката на функцията. При определени случаи, на функцията трябва да е възможно предаването на различен брой параметри при различните й извиквания. При JavaScript това е възможно чрез използване на оператор … ($\boldsymbol{spread\ operator}$).

В JavaScript обектите (включително масивите) се предават по ***референция***. Това означава, че ако промените стойностите на елементите на масив или свойство на обект от тялото на функцията, *това ще се отрази на оригиналните обекти*. На Фиг. 5.6а е показан програмния код на функция, която променя елемент от обект-масив. Ако трябва оригиналният обект да не се променя, можете да използвате копие на обекта от тялото на функцията (обект-клонинг). В JavaScript няма специални методи за *клониране*, но за масивите можете да използвате метод $slice$ без да му предавате параметри. В този случай, този метод ще върне нов обект - копие на оригиналния масив. Метод $slice$ реализира така нареченото ***повърхностно клониране*** ($shallow\ cloning$). Ако се налага ***дълбоко клониране*** ($deep\ cloning$) трябва да създадете нов масив и в цикъл да копирате всеки елемент на оригиналния масив и да го запишете в съответния елемент на новия масив.

а)

```
1  var масив = [1, 4, 5];
2  function моятафункция(масив) {
3      масив[2] = 500;
4      console.log(масив);
5  }
6  моятафункция(масив);
7  console.log(масив);
```

б)

```
▶ (3) [1, 4, 500]      code.js:4
▶ (3) [1, 4, 500]      code.js:7
```

Фиг. 5.6. Пример за предаване на параметри по референция:
а) програмен код, б) резултат

На Фиг. 5.7а е показан програмния код, който решава задачата чрез повърхностно клониране на масива (ред 3).

а)

```
1  var масив = [1, 4, 5];
2  function моятафункция(масив) {
3      масив = масив.slice();
4      масив[2] = 500;
5      console.log(масив);
6  }
7  моятафункция(масив);
8  console.log(масив);
```

б)

```
▶ (3) [1, 4, 500]      code.js:5
▶ (3) [1, 4, 5]        code.js:8
```

Фиг. 5.7. Пример за използване на повърхностно клониране на масив:
а) програмен код, б) резултат

Кодът от тялото на функцията се изпълнява при всяко „викане" на функцията. При JavaScript е възможно множество начини за викане на функции, в зависимост от начина им на дефиниране:

- Викане на функцията **по име**. Това е „стандартен" начин за викане на функция.
- Викане на функция чрез **името на променлива**. При JavaScript, тъй като функциите са тип данни, е възможно за стойност на променлива да се зададе функция. Функциите, които се дефинират без име се наричат ***анонимни***.
- Автоматично извикване на функция, веднага след нейното дефиниране. Този тип функции се наричат ***самоизвикващи се***.

Какво могат да връщат функциите? Изпълнението на кода от функцията спира при достигане до неговия край - оператор $return$. Последният се използва да укаже какво връща функцията като резултат. Ако след $return$ няма израз, функцията не връща стойност. В този случай функциалността ѝ се състои само в изпълнение на кода от тялото на функцията. Една JavaScript функция може да върне като резултат: **число**, **стойността на променлива или константа**, **масив**, **обект** или ***функция***.

Функционално програмиране с JavaScript

Всеки програмен език използва функции. При *императивните* езици функциите са подпрограми (последователност от команди), които реализират някаква функционалност, която може да промени логиката на цялата програма. При *функционалното* програмиране функцията симулира действието на математична функция, която връща винаги един и същ резултат за еднакви по стойност аргументи (кодът ѝ не дава страничен ефект). Този тип програмиране позволява писането на капсулиран код, който не дава странични ефекти, които са характерни за императивните функции. Вече знаем, че на една функция може да се предава като параметър друга функция, както и една функция да връща като резултата друга функция. Тази функционалност не е присъща за всички програмни езици, а само за тези, които поддържат ***функции първа класа*** ($First-Class\ Functions$). Тези функции са в основата на *функционалното програмиране*. Едно от предимствата на тези функции е възможността за създаване на ***функции от по-висок ред*** ($Higher-Order\ Functions$ - HOF). Функциите от по-висок ред е прието да се наричат ***функционали***. Основното предимство на функционалите е възможността за фокусиране на цялата логика, която ни е необходима, на едно място. Функционалното програмиране се базира на третиране на функциите като данни.

Повечето функционални езици се базират на *ламбда* пресмятане. При ES6+ е възможно използването на ***ламбда*** (λ) функции, известни още като ***функции-стрелка***. Те са като

анонимните функции – нямат име, но разликата е в това, че *ламбда* функциите се третират като данни. Синтаксисът на тези функции (виж Фиг. 5.8) изисква да се използва *дебела стрелка* (=>) чрез която се замества ключова дума $function$.

```
() => {
    // тяло на функцията
}
```

```
(пар1[,пар2]) => {
    // тяло на функцията
}
```

Фиг. 5.8. Синтаксис на *ламбда* функциите в JavaScript

Самоизвикващи се функции

При ES6+ е възможно дефиниране на функции, които се викат веднага след края на декларирането им. Те се наричат $Immediately\ Invoked\ Function\ Expressions$ (IIFE), за по-кратко - ***самоизвикващи се функции***, въпреки, че това наименование не е достатъчно точно. Най-често използваният стил на дефиниране на IIFE е даден от Дъглас Крокфорд (Douglas Crockford):

```
;(function() {
    ...
})();
```

```
;(() => {
    ...
})();
```

```
;((пар1, ...,парN) => {
    ...
})(стойност1,...,стойностN);
```

Следва пример за самоизвикваща се функция, която показва съобщение чрез функция $alert$:

```
(() => {
    alert("Самоизвикваща се функция.");
})();
```

4.2.4. Обекти

Програмният език JavaScript е обектно ориентиран. Той работи с два вида обекти:

- Обекти-литерали.
- Обекти, създадени по шаблон от клас.

Обекти-литерали

В JavaScript се поддържа програмни ***обекти-литерали***. Те са асоциативни масиви, по-точно колекции от двойки "ключ-стойност". Всеки ключ може да се използва еднократно в колекцията и предоставя име за *свойство* на обект. Обектите са специфична комбинация от *свойства* и техните *стойности*. Свойствата могат да бъдат както *данни*, така и *функции*. Следователно, обектът е комбинация от специфични променливи (описват характеристиките на обекта) и функции (описват действията, които обектът може да реализира). Обектът, създаден чрез литерал, е прието да се нарича ***сингълтън*** ($singleton$) или накратко ***сек***. Секът е вид шаблон за дизайн, който се използва при обектно-ориентираното програмиране. Този шаблон се прилага обикновено при моделиране на обекти, които трябва да бъдат глобално достъпни за останалите обекти на приложението. Тази концепция е печеливша когато съществува само един обект или когато е ограничено представянето на определен брой обекти. Синтаксисът на дефиниране на обекти-литерали в JavaScript е следния:

```
var имеНаОбекта = {
    имеНаСвойство_1: стойност,
    имеНаСвойство_N: стойност
};
```

Нека да създадем обект, който описва даден човек чрез неговото име, години и града в който живее.

```
var личност = {
    име: ”Иван”,
    години: 20,
    град: ”Габрово”
};
```

Обектът *личност* е съставен от 3 свойства, всички от тип данни (едно число и два низа). Въпреки, че *личност* се декларира като променлива (*var*) тя е нещо по-сложно, защото самата тя е изградена от променливи. Достъпът до свойствата на един обект може да се реализира по два начина:

- Точкова нотация: *имеНаОбекта*.*имеНаСвойство*;
- Скоба-нотация: *имеНаОбекта*['*имеНаСвойство*'];

Тъй като функциите в JavaScript са данни, то някои от свойствата на един обект-литерал могат да бъдат функции:

```
var личност = {
    име: "Иван",
    възраст: 20,
    град: "Габрово",
    върниИнформация: function() {
        var информация = this.име+" е на "+
            this.възраст+" години.";
        return информация;
    }
};
```

Фиг. 5.9. Пример за обект-литерал

Обекти, получени чрез използване на програмен клас

JavaScript поддържа както функционално програмиране, така и обектно-ориентирано програмиране. Има два подхода за деклариране на класове: 1) Чрез използване на функции и 2) Чрез ключова дума *class*.

Ще разгледаме само втория подход, който е по-нов и е препоръчително да се използва.
За деклариране на клас се използва ключова дума *class*, която е налична при ES6+. Синтаксисът за деклариране на програмен клас е показан на Фиг. 5.10.

```
class ИмеНаКласа extends ИмеНаКласКойтоСеНаследява {
    constructor(параметри) {
        // Инициализация на
        // свойствата на класа:
        this.именаСвойство = стойност;
    }
    имеНаМетод() {
        // код от тялото на метода
    }
    // Други методи от класа
}
```

Фиг. 5.10. Синтаксис на програмен клас

Всеки клас може да има специален метод, наречен **конструктор**. Веднага след създаване на обект от класа се вика неговия конструктор. Името на конструктора е $constructor$ (ред 2). Може да декларирате ***само един конструктор***, за разлика от обектно-ориентираните езици при които могат да се декларират множество конструктори с име, което съвпада с името на класа. Програмният език JavaScript позволява ***наследяване*** на класове. То се реализира както при Java – използва се ключова дума $extends$. От тялото на конструктора можете да изпълните метод $\boldsymbol{super}$ чрез който да извикате конструктора на родителския клас. Метод $super$ трябва да е *първият* метод, който се изпълнява от тялото на конструктора.

4.3. Обработка на изключения

Обработката на изключения при JavaScript е подобно по функционалност на това при обектно-ориентирани езици като Java и C#. За целта се използват ключови думи try, $catch$ и $finally$. В тялото на клауза try трябва да е програмния код, който може да генерира грешка(и). В тялото на клауза $catch$ трябва да е програмния код, който обработка евентуална грешка(и). Ако трябва да се изпълни програмен код, независимо дали е имало или не грешка, той трябва да е в тялото на клауза $finally$. Синтаксисът е следния:

```
try {
    // Код, който може да генерира грешки.
}
catch(error) {
    // Код, който обработва възникналите грешки. Информация за грешката
    // се получава от обект error.
}
finally {
    // код, който се изпълнява независимо дали има или не грешки.
}
```

VI. ВЪПРОСИ И ЗАДАЧИ ЗА ИЗПЪЛНЕНИЕ

1. Каква е структурата на един Web проект? Какви са основните HTML етикети, които се използват от тялото на заглавния блок и тялото на един HTML документ?
2. Какво е предназначението на CSS? Какъв е синтаксисът на CSS правилата. Като използвате сайта W3Schools проверете кои са CSS селекторите, които се въвеждат при CSS версия 3.
3. Напишете CSS правило, което задава жълт цвят на фона на всички div контейнери на съдържание, които са в тялото на етикет section.
4. Създайте Web проект чрез развойна среда NetBeans. За целта изберете:

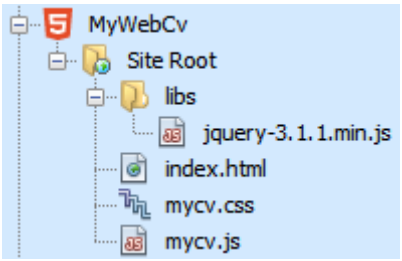
File -> NewProject … -> HTML5 Application

Развойната среда ще създаде „празен” Web проект, който съдържа в основната папка SiteRoot файл index.html. Това е документът, който се стартира при „сляпа” заявка към бъдещия сайт. Заявката е „сляпа”, когато не се задава име на ресурс до който желаем достъп, например http://localhost. Стартирайте приложението. В този момент ще се зареди Web сървър (Tomcat или Glassfish) чрез който ще можете да тествате приложението. Ако промените съдържанието на някой от файловете от проекта си,

средата автоматично ще пренареди новото съдържание (не е необходимо отново да стартирате приложението). Съдържанието на файл index.html е следното:

```
<html>
    <head>
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <div>TODO write content</div>
    </body>
</html>
```

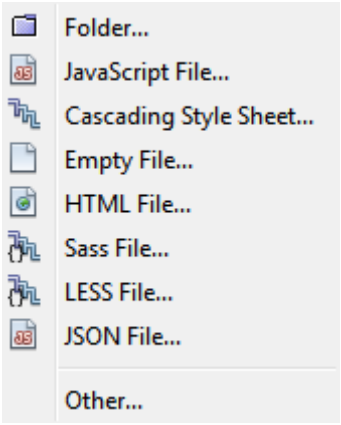
5. Създайте Web приложение, което ще бъде ваше лично CV. Структурата на проекта да бъде следната:



Освен файл index.html, създайте и следните файлове и папки:

- mycv.css – файл, който ще съдържа вашия CSS код.
- mycv.js – файл, който ще съдържа вашия JavaScript код.
- libs – папка в която ще запишете всички JavaScript библиотеки, необходими на вашия проект. В конкретния пример това е библиотека jQuery.

Когато трябва да включите към вашия проект нова папка или файл позиционирайте мишката над SiteRoot, натиснете десен бутон на мишката и изберете New… Изберете какво искате да вмъкнете в SiteRoot:



По аналогичен начин може да създавате ново съдържание или вмъквате готово съдържание чрез Paste (Ctrl+V) за всяка папка от проекта ви. На този етап редактирайте съдържанието на index.html и mycv.css. Минималната информация, която приложението трябва да визуализира, е ваша снимка и кратка текстова информация за вас.

6. Напишете JavaScript кода, който показва какъв е вътрешния тип на:

- Число, var a = 100;
- Масив, var array = [1,2,3];

- Обект-литерал, var o = {};
- Булеви данни, var flag = true;
- Функция, var f = function() {};

Опишете получената информация в таблица.

7. Разпечатайте на конзолата имената на всички дни от седмицата, които са записани в масив. Името на всеки ден да започва на нов ред. Целта е да реализирате заданието като използвате различни итерационни конструкции за реализиране на цикли.

Следват примерни варианти, които реализират условието на задачата. Ще използваме масив с цел запазване на имената на дните от седмицата:

```
var дниОтСедмицата =
["Понеделник", "Вторник", "Сряда", "Четвъртък",
"Петък", "Събота", "Неделя"];
```

Нека *индекс* е променлива, която указва номера на деня от седмицата, името на който желаем да извлечем от масива. Например, ако *индекс* $= 3$ ще се адресира „четвъртък". *Ако индекс* > 6 то *дниОтСедмицата*[*индекс*] ще върне $undefined$ защото се опитваме да адресираме извън границите на масива.

1. Реализация чрез конструкция for:

```
for (var индекс=0, n=дниОтСедмицата.length; индекс<n; индекс++) {
    console.log(дниОтСедмицата[индекс]);
}
```

2. Реализация чрез конструкция for без отчитане на размера на масива:

```
var индекс = 0;
for (;дниОтСедмицата[индекс] != undefined;) {
    console.log(дниОтСедмицата[индекс]);
    индекс++;
}
```

3. Реализация чрез конструкция $while$:

```
var индекс = 0;
const N = дниОтСедмицата.length;
while (индекс < N) {
    console.log(дниОтСедмицата[индекс]);
    индекс++;
}
```

4. Реализация чрез конструкция $while$, без отчитане на размера на масива:

```
var индекс = 0;
while (дниОтСедмицата[индекс] != undefined) {
    console.log(дниОтСедмицата[индекс]);
    индекс++;
}
```

5. Реализация чрез конструкция $for - in$:

```
for (елемент in дниОтСедмицата)
{
    console.log(дниОтСедмицата[елемент]);
}
```

6. Реализация чрез функция $forEach$:

```
дниОтСедмицата.forEach(function(ден) {
    console.log(ден);
});
```

7. Реализация чрез използване на метод $reduce$:

```
1  function покажи(temp,ден) {
2      return temp+"\n"+ден;
3  }
4  var резултат = дниОтСедмицата.reduce(покажи);
5  console.log(резултат);
```

Метод $reduce$ вика функция *покажи* толкова пъти, колкото са елементите в масива. Всеки път, когато се вика функция покажи й се предават два аргумента: 1) $temp$ – предходно върнатия резултат от метода. При първо викане на покажи, стойността на $temp$ е 0 (ако в масива има числа) или празен низ (ако в масива има низове); 2) ден – стойност на текущия елемент от масива. След обработка на всички елементи от масива $temp$ ще съдържа резултата – имената на всички дни от седмицата (всяко име на нов ред). В случая е безсмислено да задаваме име на функцията (*покажи*) – важно е не нейното име, а логиката която тя реализира. Затова е по-добре да използваме анонимна функция.

8. Реализация чрез използване на метод $reduce$ и анонимна функция:

```
var резултат = дниОтСедмицата.reduce(function(temp,ден) {
    return temp+"\n"+ден;
});
console.log(резултат);
```

9. Възможно най-кратката реализация на задачата е тази при която се използва метод $join$:

```
console.log(дниОтСедмицата.join("\n"));
```

8. Напишете функция, която връща сумата на стойностите на всички параметри, които и се предават. За да проверите дали предавания параметър е число използвайте оператор typeof и Клаузи try-catch.

Нека функцията да се нарича **сума**. Тъй като не е ясно колко параметъра се предават ще използваме spread оператор (…параметри). При spread оператор параметрите вътрешно се конвертират до масив. Затова броят им се получава чрез length (ред 3). Използваме for цикъл, за да проверим и сумираме стойността на всеки параметър (редове 4-17). След извличане на стойността на параметър (ред 5) се проверява дали той е число (ред 7). Ако е число се реализира операция сумиране (ред 8). В противен случай се генерира изключение, чийто текстово описание е „не е число” (ред 11). При грешка се активира кода от клауза catch (ред 15). Викането на функцията е показано на ред 20. Два от параметрите са невалидни – низ и булев тип данни.

```
function сума(...параметри) {
    let резултат = 0;
    let n = параметри.length;
    for(let i=0; i<n; i++) {
        let елемент = параметри[i];
        try {
            if(typeof елемент == "number") {
                резултат += елемент;
            }
            else {
                throw "не е число!";
            }
        }
        catch(грешка) {
            alert(елемент+": "+грешка);
        }
    }
    return резултат;
}
alert(сума(56,true,"проба",2.5));
```

9. Реализирайте Задача 8 като реализирате цикъла не чрез for, а чрез използване на метод forEach и ламбда функция.
10. Създайте обект-литерал който описва вас като студент. Обектът да съдържа следните свойства:
    - Име и фамилия на студента.
    - Година на обучение.
    - Специалност.
    - Семестър (летен или зимен).
    - Функция, която връща информация за студента като низ.
11. Реализирайте Задача 10 като използвате програмен клас, за да опишете студент.