НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Модуль для ізоляції контекстів JavaScript для віртуальних машин що працюють у браузері

Виконав : студент __4__ курсу, групи ІП-94
(шифр групи)
_____Перевалов Максим Олексійович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____ст. викладач Шемсединов Т.Г._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) доцент, к.т.н Волокита А.М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Перевалова Максима Олексійовича

1. Тема проєкту *Модуль для ізоляції контекстів JavaScript для віртуальних машин що працюють у браузері*
керівник проєкту Шемсединов Тимур Гафарович ст. викладач.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 31 травня 2023 року №2102-с

2. Термін здачі студентом закінченого проєкту *17 червня 2023 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Огляд предметної області.

Розділ 2. Аналіз технологій створення контекстів.

Розділ 3. Розробка програмного продукту.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) Діаграма модулів (структурна схема), Діаграма компонентів (функціональна схема), Послідовність дій визначення властивості (Принципова схема).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А.М.		

7. Дата видачі завдання «1» лютого 2023 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.02.2023-15.02.2023*	
2.	*Вивчення та аналіз завдання*	*15.02.2023-30.02.2023*	
3.	*Розробка архітектури та загальної структури системи*	*30.02.2023-15.03.2023*	
4.	*Розробка структур окремих підсистем*	*15.03.2023-15.04.2023*	
5.	*Програмна реалізація системи*	*15.04.2023-15.05.2023*	
6.	*Оформлення пояснювальної записки*	*15.05.2023-25.05.2023*	
7.	*Захист програмного продукту*	*30.05.2023*	
8.	*Передзахист*	*08.06.2023*	
9.	*Захист*	*24.06.2023*	

Студент-дипломник _____ Максим ПЕРЕВАЛОВ
(підпис)

Керівник проєкту _____ Тимур ШЕМСЕДИНОВ
(підпис)

АНОТАЦІЯ

У даній роботі було детально розглянуто різні підходи для створення ізольованих контекстів JavaScript в браузері та основні принципи їх роботи. Були проаналізовані їхні слабкі та сильні сторони. На основі аналізу було вибрано кращий механізм створення ізольованого контексту, на основі якого був побудований відповідний модуль. В результаті роботи було розроблено модуль для створення ізольованих контекстів, який вирішує задачу по відокремленню глобального JavaScript контексту. Програмний продукт був розроблений на мові JavaScript.

Ключові слова: глобальний контекст, JavaScript, веб-переглядач

ANNOTATION

In this project for a Bachelor's Degree, various approaches for creating isolated JavaScript contexts in the browser and the basic principles of their operation were considered in detail. Their weaknesses and strengths were analyzed. Based on the analysis, the best mechanism for creating an isolated context was selected, based on which the corresponding module was built. As a result of the work, a module for creating isolated contexts was developed, which solves the problem of separating the global JavaScript context. The software product was developed in the JavaScript programming language.

Keywords: global context, JavaScript, web browser

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Модуль для ізоляції	3		
			контекстів JavaScript для			
			JavaScript для віртуальних			
			машин що працюють у			
			браузері			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Модуль для ізоляції	65		
			контекстів JavaScript для			
			JavaScript для віртуальних			
			машин що працюють у			
			браузері			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Діаграма модулів	1		
			(Структурна схема)			
	А4	***ІАЛЦ.4672008.005 Д2***	Діаграма компонентів	1		
			(Функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Послідовність дій	1		
			визначення властивості			
			(Принципова схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Модуль для ізоляції	3		
			контекстів JavaScript для			
			JavaScript для віртуальних			
			машин що працюють у			
			браузері			
			Текст програмного коду			

Зм	*Лист*	*№ докум.*	*Підп*	*Дата*		
Розроб		Перевалов М.О.				
Перев		Шемсединов Т.Г.				

ІАЛЦ.467200.001 ОА

Модуль для ізоляції контекстів JavaScript для віртуальних машин що працюють у браузері
Опис альбому

Літ.	Аркуш	Аркушів
	1	1

КПІ ім. Ігоря Сікорського
ФІОТ
ІП-94

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Модуль для ізоляції контекстів JavaScript для віртуальних машин що працюють у браузері*»

Київ – 2023

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив		Перевалов М. О.			*Модуль для ізоляції контекстів JavaScript для віртуальних мащинщо праюють у браузері* Технічне завдання	Літ.	Аркуш 1 / Аркушів 3
Перевірив		Шемсединов Т.Г.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94	
Н. Контр.		Волокита А. М.					
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи для створення ізольованих JavaScript контекстів, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є проектування навчальних модулів, механізмів безпеки в веб-перглядачах, модулів які використовують маніпуляції з глобальним контекстом

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка модуля для створення ізольованих JavaScript контекстів що працюють у браузері

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий і інтуїтивно-зрозумілий інтерфейс системи.
- Автоматичне управління ресурсами витраченими на створення ізольованих контекстів
- Надати можливість користувачам передавати властивості ізольованого контексту
- Надати можливість користувачам виконувати свій програмний код в ізольованих контекстах

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- WebStorm 2022 IDE версії або вище.

5.3. Вимоги до апаратної частини

- ЦП Intel Pentium 4 або пізнішої версії із підтримкою SSE3.
- ROM не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.02.2023-15.02.2023
Вивчення та аналіз завдання	15.02.2023-30.02.2023
Розробка архітектури та загальної структури системи	30.02.2023-15.03.2023
Розробка структур окремих частин системи	15.03.2023-15.04.2023
Програмна реалізація системи	15.04.2023-15.05.2023
Виправлення помилок	15.05.2023-20.05.2023
Оформлення пояснювальної записки	25.05.2023

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Модуль для ізоляції контекстів JavaScript для віртуальних машин що працюють у браузері*»

Київ – 2023

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

JS	(JavaScript) Мова програмування JavaScript
API	(Application Programming Interface) Набір правил і інструкцій, які дозволяють різним програмним компонентам взаємодіяти між собою
DOM	(Document Object Model) Представлення веб-сторінки у вигляді деревоподібної структури об'єктів, де кожен елемент на сторінці є об'єктом
JSON	(JavaScript Object Notation) Формат обміну даними, заснований на синтаксисі JavaScript
URL	(Uniform Resource Locator) адреса, яка ідентифікує ресурс у мережі Інтернет
HTTP	(Hypertext Transfer Protocol) Протокол передачі гіпертекстових документів у мережі Інтернет

ВСТУП

JavaScript є однією з найбільш часто використовуваних мов програмування для розробки веб-додатків і взаємодії з користувачем у веб-середовищі. Одним із головних недоліків мови JavaScript є глобальний контекст, який може змінювати користувач і де зберігаються всі компоненти стандартної бібліотеки.

Метою цього дипломного проекту є розробка модуля ізоляції контексту JavaScript для віртуальних машин які працюють у браузера.

Ізольований контекст JavaScript – це область, у якій змінні, функції та об’єкти існують і працюють незалежно від решти коду. Такий контекст виконує фрагмент коду, який не впливає на глобальний контекст чи інші контексти.

Основні особливості ізольованого контексту включають:

1) Область видимості: Змінні та функції, оголошені в ізольованому контексті, доступні лише в ньому. Це допомагає уникнути конфліктів і непередбачених змін у глобальному масштабі.
2) Відокремлене середовище: Ізольований контекст може мати свої власні об'єкти та змінні, які не впливають на інші контексти. Це дозволяє створювати незалежні середовища для виконання різних фрагментів коду, що сприяє модульності та роздільності в програмах.
3) Робота з бібліотеками та фреймворками: В ізольованому контексті можна використовувати зовнішні бібліотеки та фреймворки без впливу на глобальний контекст. Це дозволяє запускати код, який використовує різні версії бібліотек або фреймворків, не впливаючи на інші частини програми.
4) Тестування та налагодження: Ізольований контекст полегшує тестування та налагодження коду. Ви можете виконувати тести або налагоджувати окремі функції чи модулі в ізольованому середовищі без впливу на решту програми.

РОЗДІЛ 1
ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Загальний огляд мови програмування Javascript

JavaScript (JS) — це інтерпретована мова програмування високого рівня, яка широко використовується для розробки веб-додатків. Вона використовується для створення динамічного вмісту на веб-сторінках, взаємодії з користувачами, маніпулювання елементами сторінки та виконання асинхронних запитів до серверів.

JavaScript з'явився в 1995 році, коли Брендан Айк розробив його в Netscape Navigator. Мова спочатку називалася LiveScript, але пізніше була перейменована на JavaScript, щоб збільшити її популярність за допомогою прив'язки до мови Java.

Синтаксис JavaScript простий і зрозумілий, і початківці можуть легко його вивчити. Це динамічно типізована мова, що означає, що змінні можуть змінювати свій тип даних під час виконання програми. Це дає розробникам гнучкість, але також вимагає обережності, щоб уникнути помилок.

Вона підтримує широкий спектр функціональних можливостей, включаючи роботу з рядками, масивами, об’єктами, датами, математичними операціями, регулярними виразами, а також надає доступ до багатьох вбудованих об’єктів і методів, які спрощують усі аспекти веб-розробки.

1.2 Javascript на платформі веб-переглядачів

JavaScript переважно використовується на платформах браузерів. Веб-браузери, такі як Google Chrome, Mozilla Firefox, Safari, Microsoft Edge,

забезпечують середовище для виконання коду JavaScript безпосередньо на стороні клієнта.

Коли користувач завантажує веб-сторінку, браузер завантажує структуру HTML сторінки, а також код JavaScript, що міститься на сторінці. При досягненні частини коду JavaScript браузер виконує відповідні дії.

JavaScript дозволяє створювати інтерактивні елементи на ваших сторінках, такі як кнопки, форми, меню, слайдери тощо. Це дозволяє реагувати на такі події, як натискання кнопок, наведення курсора, введення користувача тощо.

Використовуючи JavaScript, ви можете робити асинхронні запити до сервера, що дозволяє отримувати або надсилати дані без перезавантаження сторінки. Це дозволяє створювати динамічні веб-додатки, які отримують оновлені дані з сервера без затримки.

```
const name = prompt("Як тебе звати?");
const greeting = `Привіт, ${name}!`;
document.getElementById("greeting").textContent = greeting;
```

Рисунок 1.1 – Приклад JS коду на платформі веб-переглядачів

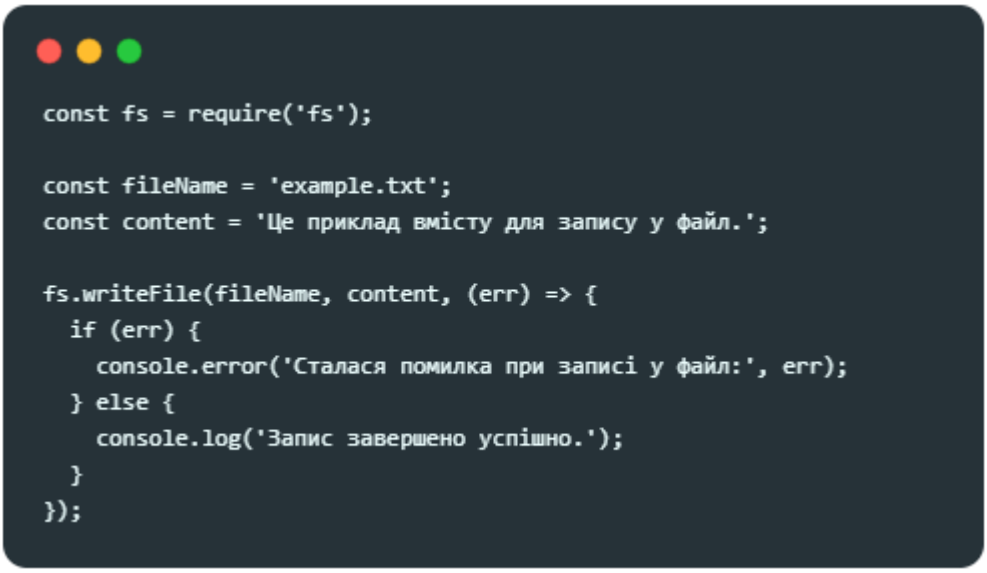
У цьому прикладі ми використовуємо функцію prompt для відображення діалогового вікна, в якому користувач може ввести своє ім'я. Далі ми знаходимо HTML елемент з ідентифікатором "greeting" за допомогою getElementById і змінюємо його текстовий вміст на рядок привітання.

1.3 Javascript на платформі Node.js

Node.js — це середовище виконання JavaScript поза браузером. Це дозволяє вам виконувати JavaScript на стороні сервера та розробляти програми на стороні сервера, веб-програми, інструменти командного рядка тощо. Node.js побудовано на принципах однопотокових і неблокуючих операцій введення-

виведення. Це означає, що ви можете виконувати багато операцій асинхронно, не блокуючи виконання інших операцій. Це покращує продуктивність і масштабованість серверних програм.

Він забезпечує доступ до системних ресурсів, таких як файлова система, мережеві операції, потоки введення/виведення, обробка запитів HTTP, виконання команд командного рядка тощо. Ви можете створювати власні сервери, обробляти файли, взаємодіяти з базами даних, створювати API та робити багато іншого за допомогою JavaScript на платформі Node.js.



```
const fs = require('fs');

const fileName = 'example.txt';
const content = 'Це приклад вмісту для запису у файл.';

fs.writeFile(fileName, content, (err) => {
  if (err) {
    console.error('Сталася помилка при записі у файл:', err);
  } else {
    console.log('Запис завершено успішно.');
  }
});
```

Рисунок 1.2 – Приклад JS коду на платформі Node.js

У цьому прикладі ми використовуємо модуль fs для запису файла. Ми вказуємо ім'я файлу і його вміст, який ми хочемо записати. Функція writeFile асинхронно записує його у файл.

1.4 Двигун v8

V8 — це високопродуктивний двигун JavaScript, який використовується для виконання JavaScript у таких середовищах, як веб-браузери та серверні програми. Він був розроблений Google і вперше використовувався в браузері Google Chrome.

Однією з головних особливостей V8 є те, що він виконує JavaScript як машинний код, а не інтерпретує його. Це значно збільшує швидкість виконання коду JavaScript, оскільки машинний код працює швидше, ніж інтерпретований код.

V8 також має вбудований механізм оптимізації коду під назвою JIT (Just-In-Time Compilation). Механізм аналізує виконання коду JavaScript у режимі реального часу та перетворює його на ефективний машинний код. Це дозволяє підвищити продуктивність програми, особливо під час виконання великих обчислювальних завдань або обробки великих обсягів даних.

Крім того, V8 підтримує виконання JavaScript за допомогою асинхронного програмування, включаючи використання Promise і функцій async/await. Це дозволяє розробникам створювати ефективні програми без блокування, які можуть обробляти багатопотокові операції, не блокуючи основний потік виконання.

V8 має відкритий код і доступний не лише в Google Chrome, але й в інших браузерах, таких як Opera та Node.js. Він має широку підтримку стандартів JavaScript, включаючи ECMAScript 6 і пізніших версій.

Завдяки швидкості виконання та потужним можливостям оптимізації V8 став популярним вибором для розробників, які працюють з JavaScript. Це дозволяє створювати швидкі та ефективні веб-додатки, серверні додатки та інші системи, використовуючи JavaScript як мову програмування.

1.5 Стандарт ECMAScript

ECMAScript (також відомий як ES) — це стандарт, на основі якого JavaScript реалізовано багатьма сучасними веб-платформами та браузерами. ECMAScript визначає специфікацію мови для створення динамічного вмісту на веб-сторінках.

Перша версія ECMAScript була випущена в 1997 році, відповідаючи специфікації JavaScript 1.1. З того часу було випущено кілька версій ECMAScript, кожна з яких містить нові функції, покращення та мовні зміни. Найпоширенішою версією сьогодні є ECMAScript 5, яка була випущена в 2009 році.

ECMAScript визначає правила і синтаксис JavaScript, що дозволяє розробникам створювати програми та веб-додатки з використанням мови JavaScript.

- Включає різні типи даних, такі як числа, рядки, булеві значення, об'єкти, масиви, функції та інші.
- Має можливість приведення типів та операції над типами даних.
- Має механізми для обробки помилок, такі як виключення (try/catch/finally), які дозволяють контролювати та обробляти помилки під час виконання програми.
- Підтримує об'єктно-орієнтоване програмування, що дозволяє створювати класи, об'єкти, використовувати наслідування та поліморфізм.
- Дозволяє визначати функції та використовувати їх як об'єкти.
- Надає можливості для операцій над рядками, такі як конкатенація, розбиття, пошук та заміна тексту.
- Підтримує використання регулярних виразів для роботи з текстом.
- З введенням ECMAScript Modules (ESM), стандарт отримав вбудовану систему модулів, що дозволяє імпортувати та експортувати функції, класи, об'єкти та змінні між різними модулями.
- Має механізми для асинхронного програмування, такі як Promise та async/await, які дозволяють створювати асинхронні операції та керувати їхнім виконанням.

1.6 Javascript контекст

1.6.1 Види контекстів

У Javascript контекст відноситься до середовища, в якому виконується деякий код. Контекст визначає доступні змінні, функції та об’єкти, які можна використовувати в певному місці коду. У Javascript є два типи контекстів: глобальний контекст і контекст виклику функції.

- Глобальний контекст: Це контекст, який існує на рівні всього документа або веб-сторінки. Усі змінні та функції, оголошені на цьому рівні, стають глобальними змінними та доступні з будь-якої точки коду.
- Контекст виклику функції: Щоразу, коли викликається функція, створюється новий контекст виклику функції. Контекст виклику функції містить параметри функції, локальні змінні, це значення та посилання на зовнішні контексти.

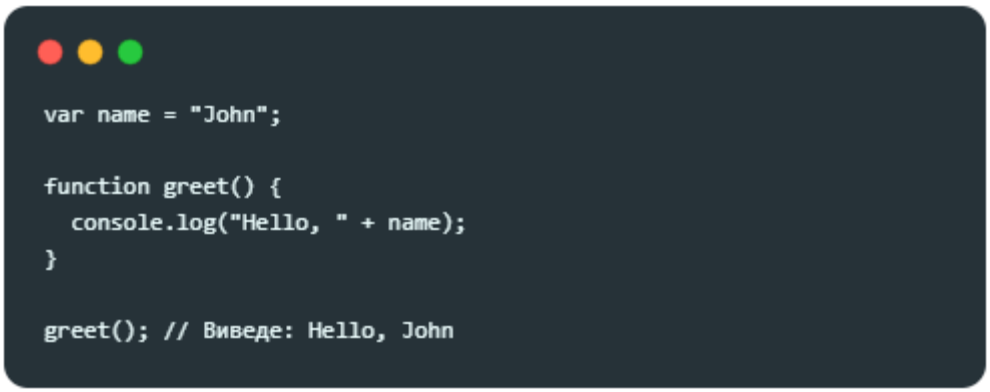
Контекст визначає, як код взаємодіє зі змінними, функціями та об’єктами в певних точках виконання коду. Наприклад, під час доступу до змінної Javascript спочатку шукає її в поточному контексті, якщо не знайдено, він переходить до батьківського контексту та інших зовнішніх контекстів, доки не знайде змінну або досягне глобального контексту.

Контекст також впливає на значення ключового слова this, яке вказує на поточний об’єкт, пов’язаний із функцією чи методом. Значення може змінюватися залежно від способу виклику функції та місця її оголошення.

Розпізнавання контексту в Javascript допомагає керувати областю змінних, уникати конфліктів імен і гарантувати, що ваш код правильно взаємодіє з об’єктами та функціями.

У прикладі Рисунок 1.3 name є глобальною змінною, оголошеною в глобальному контексті. Функція greet також доступна в глобальному контексті.

При виклику функції greet(), вона має доступ до змінної name, оскільки вона знаходиться в тому ж глобальному контексті.



```
var name = "John";

function greet() {
  console.log("Hello, " + name);
}

greet(); // Виведе: Hello, John
```

Рисунок 1.3 – Глобальний контекст

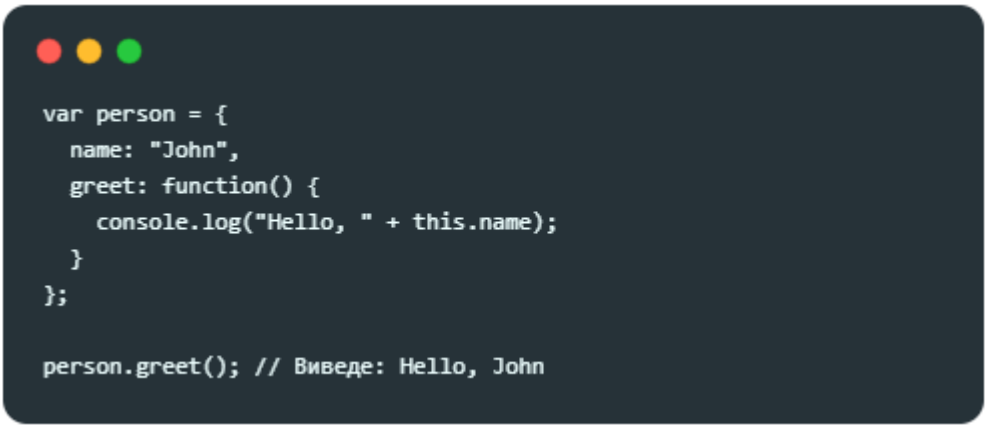
У прикладі Рисунок 1.4 функція greet() оголошена в контексті виклику функції sayHello(). Коли викликається sayHello(), створюється новий контекст виклику функції. При виклику greet() всередині sayHello(), контекст виклику функції greet() отримує доступ до своїх локальних змінних, таких як name. Тому виведення буде "Hello, John", оскільки greet() викликається всередині контексту sayHello().

```
function greet() {
  var name = "John";
  console.log("Hello, " + name);
}

function sayHello() {
  var name = "Sarah";
  greet();
}

sayHello(); // Виведе: Hello, John
```

Рисунок 1.4 – Контекст виклику функції

У прикладі Рисунок 1.5 об'єкт person має властивість name і метод greet(). Коли викликається метод greet(), значення this вказує на об'єкт person, з яким

пов'язана ця функція. Тому this.name отримує значення "John" з властивості name об'єкта person.



```
var person = {
  name: "John",
  greet: function() {
    console.log("Hello, " + this.name);
  }
};

person.greet(); // Виведе: Hello, John
```

Рисунок 1.5 – Контекст виклику функції

Контекст в JavaScript впливає на доступність змінних та функцій, а також на значення this в межах виконання коду. Розуміння контексту допомагає ефективно використовувати змінні та функції та забезпечує правильну взаємодію коду з об'єктами та функціями в JavaScript.

1.6.2 Глобальний контекст ECMAScript

В ECMAScript глобальний контекст представлений об'єктом globalThis, який є глобальним об'єктом, доступним у всіх частинах програми. Цей об'єкт містить глобальні змінні, функції та об'єкти, які можуть бути використані у всьому коді.

В глобальному контексті ECMAScript можна оголошувати змінні та функції, які будуть доступні у всьому коді. Наприклад, глобальні змінні можна оголосити за допомогою ключового слова var.

Він містить різні вбудовані об'єкти та функції, такі як Object, Array, Math, JSON, parseInt, setTimeout та інші. Вони надають корисні методи та функціональність, яку можна використовувати у всьому коді.

В контексті ECMAScript можна оголошувати змінні та функції в рамках модулів, які забезпечують локальну область видимості. Це дозволяє розділяти код на незалежні модулі та контролювати доступ до змінних та функцій.

1.6.3 Глобальний контекст веб-переглядачів

У глобальному контексті веб-браузера глобальний об’єкт представлений об’єктом window. Це глобальний контейнер, який містить багато властивостей і методів, що представляють функціональність браузера. Наприклад, window містить методи для взаємодії з DOM, керування вікнами та фреймами, обробки подій, використання таймерів тощо.

Глобальний контекст браузера також надає доступ до DOM, який представляє структуру сторінки HTML у вигляді дерева об’єктів. Доступ до об’єктів, що відповідають елементам HTML, можна отримати та змінити з JavaScript за допомогою властивостей і методів, наданих DOM API.

Він надає доступ до вбудованих об’єктів і функцій, таких як parseInt, setTimeout і різних API браузера, таких як XMLHttpRequest, fetch, localStorage, геолокація, Canvas, Web Workers тощо. Ці API дозволяють вам взаємодіяти з Інтернетом, зберігати дані локально, отримувати географічне розташування користувача та виконувати різноманітні операції, пов’язані з браузером.

1.6.4 Глобальний контекст Node.js

У Node.js глобальний контекст відрізняється від глобального контексту веб-браузера. Його описує глобальний об'єкт. Цей об’єкт містить різні глобальні змінні, функції та об’єкти, які можна використовувати в коді. Наприклад, глобальний об’єкт global містить вбудовані об’єкти та функції, такі як Buffer, console, process, require, module, exports тощо.

Однією з відмінностей у глобальному контексті Node.js є підтримка модульної системи CommonJS. Ця система дозволяє вам розділити ваш код на модулі та використовувати механізми для імпорту та експорту функцій,

об’єктів або змінних між модулями. Модулі можна завантажувати та використовувати за допомогою глобального об’єкта require і module.exports або exports.

Він також надає доступ до об’єкта process, який надає інформацію про процес виконання Node.js, таку як аргументи командного рядка, змінні середовища, потоки введення/виведення тощо.

Глобальний контекст містить велику кількість вбудованих модулів, які дозволяють вам взаємодіяти з кожним аспектом системи. Наприклад, модуль fs дозволяє працювати з файловою системою, модуль http дозволяє створювати HTTP-сервери та клієнти, а модуль crypto забезпечує функції шифрування та дешифрування даних

ВИСНОВОК ДО РОЗДІЛУ 1

У цьому розділі ми досліджували різні аспекти мови програмування JavaScript та її використання на різних платформах. Починаючи із загального огляду мови, було преставлено основні функції JavaScript та його виконання на платформах браузера та Node.js. Розгялунули стандарт ECMAScript, який визначає синтаксис і функціональні можливості JavaScript і компонентів стандартної бібліотеки.

Зробили огляд поняття контексту, особливості глобальних контекстів на різних платформах, розглянули різні види контекстів та їх реалізацію в мові Javascript, яка дозволяє проводити досить різноманітні маніпуляції з властивостями глобального контексту які можуть бути досить небезпечними, наприклад модифікація компонетів стандартної бібліотеки.

Таким чином досить цікавою за інженерної точки зору та корисною є задача створення ізольованого контексту Javascript в браузері. Це дозволить уникнути модифікації компонетів стандартної бібліотеки сторонім кодом, надасть можливість досить просто будувати проекти яким потрібно модифікувати глобальний контекст, наприклад для імітації глобального контексту Node.js в браузері або видалення деяких властивостей глобального контексту для навчальний цілей.

РОЗДІЛ 2
АНАЛІЗ ТЕХНОЛОГІЙ СТВОРЕННЯ КОНТЕКСТІВ

2.1 Бібліотека Node.js vm

Модуль vm у стандартній бібліотеці Node.js є одним із найкращих прикладів реалізації ізольованих контекстів у Javascript. Цей модуль надає інтерфейс для керування та створення віртуальних машин у середовищі Node.js. Він дозволяє виконувати код JavaScript в ізольованому середовищі, що дозволяє контролювати взаємодію із зовнішніми ресурсами та їх виконання. API бібліотеки vm містить такі класи та методи:

- vm.Script: Цей клас надає можливість компілювати фрагменти JavaScript-коду в об'єкти Script, які можна пізніше виконати. Це досягається за допомогою конструктора new vm.Script(code[, options]), де code - рядок з JavaScript-кодом, а options (необов'язковий) - об'єкт з додатковими параметрами компіляції.
- vm.createContext: Ця функція створює новий ізольований контекст виконання в Node.js, в якому можна виконувати JavaScript-код. Синтаксис: vm.createContext([sandbox]), де sandbox є необов'язковим параметром який є об'єктом і використовується як початковий контекст для виконання коду.
- script.runInContext: Цей метод виконує об'єкт класу Script в заданому контексті виконання. Синтаксис цього методу передбачає виклик script.runInContext з параметром context, який представляє контекст виконання, а також необов'язковим параметром options, який є об'єктом з додатковими параметрами для виконання коду.

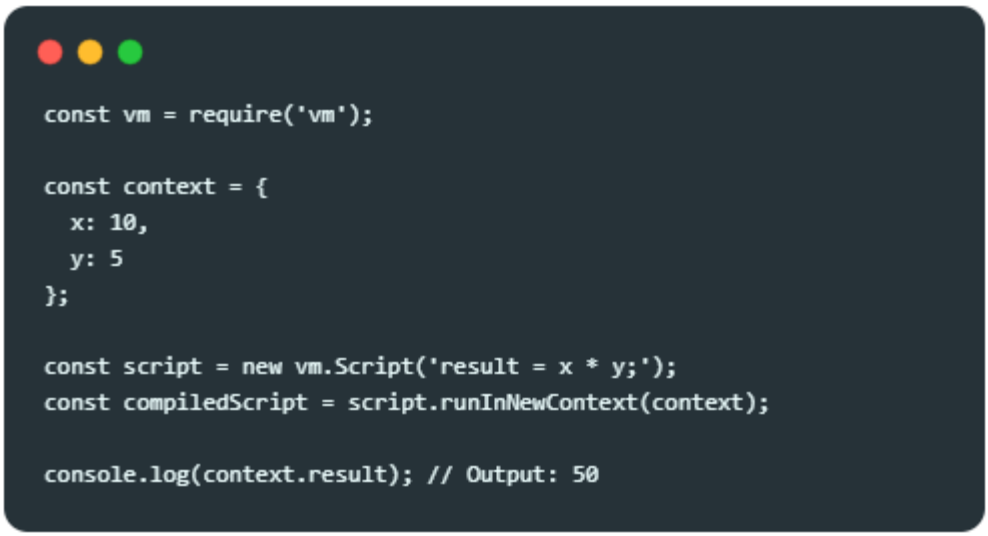
Бібліотека vm у Node.js використовує механізм V8 для виконання коду JavaScript. Він надає відповідні обгортки та інтерфейси для створення та

керування віртуальними машинами. Реалізація бібліотеки vm базується на концепції ізольованих контекстів виконання. Коли новий контекст створюється за допомогою vm.createContext(), для цього контексту створюється внутрішній об’єкт V8. Контекст V8 містить окрему глобальну область і стек викликів.

Коли код компілюється за допомогою vm.Script, він перетворюється на внутрішню структуру даних V8 під назвою «байт-код». Цей байт-код є представленням коду JavaScript, який ефективно виконується двигуном V8.

Метод runInContext виконує байт-код після компіляції відповідного сценарію в контексті виконання. Це спричиняє виклик виконавця V8. Він інтерпретує байт-код і виконує дії на основі коду JavaScript.

Безпека виконання є важливим аспектом реалізації віртуальної машини. Це дозволяє ізолювати виконання коду в контексті JavaScript, забезпечуючи контроль над доступом до ресурсів і обмежуючи можливість небезпечних операцій. Крім того, модуль vm надає можливість використовувати пісочницю, яка також обмежує доступ до небезпечних об’єктів і функцій. Усі ці механізми реалізовано в двигуні V8, а бібліотека vm надає зручний API для використання цих функцій у середовищі Node.js.



```
const vm = require('vm');

const context = {
  x: 10,
  y: 5
};

const script = new vm.Script('result = x * y;');
const compiledScript = script.runInNewContext(context);

console.log(context.result); // Output: 50
```

Рисунок 2.1 - Виконання коду зі змінними контексту

У прикладі Рисунок 2.1 створюється новий контекст context, який має змінні x та y. result = x * y - компілюється і виконується в новому контексті за допомогою

runInNewContext(). Результат виконання зберігається в змінній result контексту, і його можливо вивести.

```
const vm = require('vm');

const sandbox = {
  console: {
    log: (message) => {
      console.log('Output from sandbox:', message);
    }
  }
};

const script = new vm.Script('console.log("Hello from sandbox!");');
script.runInNewContext(sandbox);
```

Рисунок 2.2 - Використання песочниці для безпечного виконання коду

У прикладі Рисунок 2.2 створюється sandbox з обмеженими властивостями. Передається об'єкт sandbox, якому перевизначається функція console.log() для виведення повідомлення за префіксом. Функція runInNewContext() виконує скрипт "Hello from sandbox!" в пісочниці, і виводить результат за допомогою заміненої функції console.log()

```
const vm = require('vm');

const context1 = vm.createContext({ x: 10 });
const context2 = vm.createContext({ x: 5 });

const script = new vm.Script('console.log(x);');
script.runInContext(context1); // Output: 10
script.runInContext(context2); // Output: 5
```

Рисунок 2.3 – Приклад використання різних контекстів

У прикладі Рисунок 2.3 створюється два різних контексти context1 і context2 зі змінною x. Компілюється однаковий скрипт console.log(x) і

виконується в кожному контексті окремо, що демонструє виконання коду з різними значеннями змінних у різних контекстах.

Модуль vm має зручний та добре спроектований API. Він містить прості та логічні методи, які використовувати та легко розуміти. Основні методи runInContext(), та createContext() надають гнучкість та контроль над створенням та виконанням ізольованих контекстів JavaScript. Створення подібного API, як у модулі vm є чудовим рішенням і ось чому:

- Зручність використання: Його API вже буде добре знайомим для розробників які використовують vm в своїх проектах. Якщо модуль має схожий API, це дозволяє швидко впроваджувати його без необхідності знайомитися з новим інтерфейсом, що полегшує перехід і збережує час розробки.
- Спроектованість: Добре спроектований API з логічною структурою та послідовністю методів добре сприяє розумінню та використанню модуля, якщо використати подібну структуру та послідовність методів, це полегшить розуміння та використання модуля.
- Сумісність: Подібне API добре сприяє сумісності з існуючими інструментами та бібліотеками, що використовують vm для створення ізольованих контекстів JavaScript. Цей модуль буде легко інтегрувати в бібліотеку з існуючими кодовими базами та інфраструктурою

2.2 Web assembly

2.2.1 Загальний опис Web assembly

WebAssembly (або скорочено wasm) — це формат байт-коду, призначений для запуску веб-додатків у середовищі браузера. WebAssembly вперше було оголошено в 2015 році, а в 2019 році він став стандартом W3C (World Wide Web Consortium).

WebAssembly розроблено, щоб забезпечити швидке завантаження та виконання, а також покращити продуктивність розробки складних програм у веб-середовищі. Він може бути скомпільований з різних мов програмування, включаючи C, C++, Rust та багатьох інших, що робить його незалежним від мови форматом

Основна ідея WebAssembly полягає в тому, щоб мати ефективний і компактний формат, який можна швидко інтерпретувати та виконувати в браузері. Код WebAssembly можна виконувати в контексті JavaScript. Він також має можливість працювати безпосередньо в браузері, використовуючи спеціальний інтерфейс, який забезпечує доступ до функціональності браузера.

Розробники можуть використовувати мови програмування низького рівня, такі як C++, для створення потужних і швидких програм, які запускаються в браузері, відкриваючи кращу масштабованість, продуктивність і портативність для веб-програм

Таким чином, WebAssembly — це формат байт-коду, який дозволяє запускати потужні та швидкі веб-програми у веб-середовищі. Це відкриває нові можливості для розробників і забезпечує кращу продуктивність і масштабованість веб-додатків незалежно від використовуваної мови програмування.

2.2.2 Процес використання Web assembly в браузері

По-перше, розробники пишуть код мовою програмування, яку підтримує WebAssembly, наприклад Rust, C++ або іншими. Цей код може містити структури даних, класи, функції та інші компоненти програми

Код на мові програмування компілюється в байт-код WebAssembly. Для цього вам потрібно використовувати спеціальні компілятори, такі як Emscripten для C/C++ або wasm-pack для Rust. Компілятор перетворює вихідний код на мові програмування в незалежний від платформи байт-код WebAssembly.

Отриманий файл із розширенням .wasm (байт-код WebAssembly) завантажується в браузер. Це можна зробити за допомогою функції fetch(), яка завантажує файл із сервера або локального джерела.

Завантажений файл .wasm передається у функцію WebAssembly.instantiate(), яка створює екземпляр модуля WebAssembly. Модуль містить виконуваний код, імпортовані та експортовані функції та змінні.

Після створення екземпляра модуля з ним можна взаємодіяти через інтерфейс JavaScript. Отримавши доступ до експортованих функцій і змінних модуля, можна викликати функції WebAssembly, передавати параметри та отримувати результати або змінювати стан системи.

Результат виконання функції WebAssembly можна використовувати в коді JavaScript браузера. Наприклад, результати можна відобразити на веб-сторінці, передати в іншу функцію JavaScript або використати для подальшої обробки даних.

Розглянемо на прикладі використання WebAssembly для виклику функції add, яка додає два числа, функція написана на мові C++

```cpp
// add.cpp
extern "C" {
    int add(int a, int b) {
        return a + b;
    }
}
```

Рисунок 2.4 – Код на C++ для Web assembly

Спочатку ми напишемо функцію add у файлі з розширенням .cpp. Нехай вона буде додавати два цілі числа:

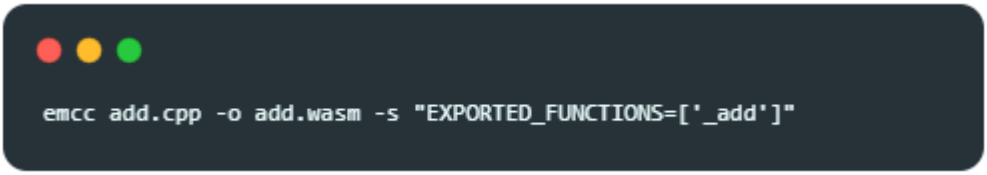


Рисунок 2.5 – Компіляція в web assembly

Потім ми скомпілюємо цей файл C++ у WebAssembly за допомогою компілятора, наприклад Emscripten. Для цього потрібно використати наступну команду Рисунок 2.5, яка компілює add.cpp у add.wasm із експортом функції _add.

Тепер, коли у ми маємо файл WebAssembly (add.wasm), можливо завантажити його у браузер та викликати функцію add. Ось приклад JavaScript-коду, який показує, як це можливо зробити:

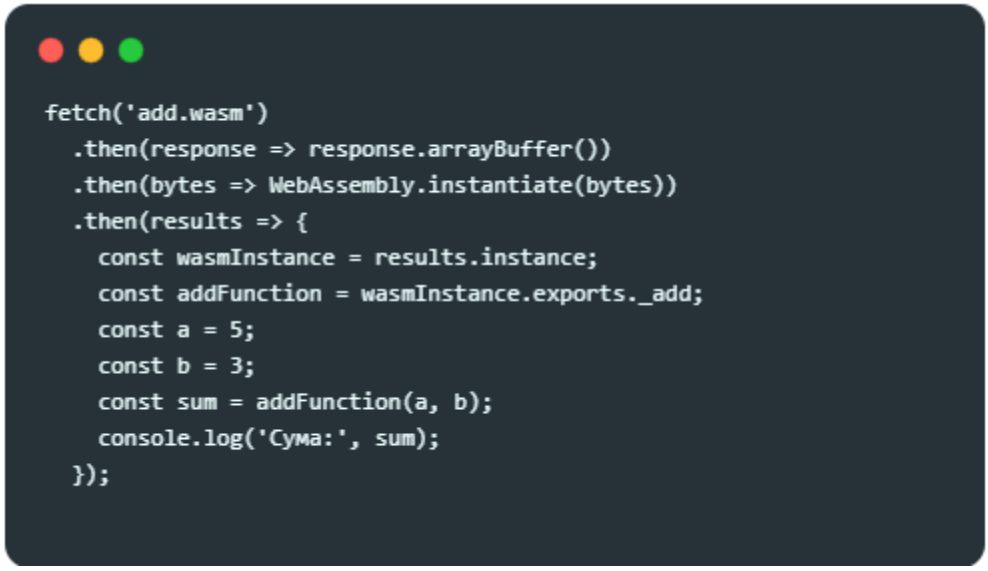


Рисунок 2.6 – Завантаження web assembly файлу

У прикладі Рисунок 2.6 ми завантажуємо файл add.wasm, інстанціюємо його та отримуємо доступ до експортованої функції _add. Потім ми передаємо два аргументи a і b до функції add і отримуємо результат. Розглянемо його більш детально

- fetch('add.wasm'): Цей рядок починає завантаження файлу add.wasm, який містить байткод WebAssembly. fetch() є вбудованою

функцією JavaScript, яка виконує HTTP-запит для отримання ресурсу з сервера.

- .then(response => response.arrayBuffer()): Далі викликається метод arrayBuffer(), щоб отримати відповідь сервера
- .then(bytes => WebAssembly.instantiate(bytes)): У цьому рядку bytes, представляє отриманий ArrayBuffer який передається у функцію WebAssembly.instantiate(). Ця функція створює екземпляр WebAssembly модуля.
- .then(results => { ... }): В цьому рядку ми отримуємо результати створення екземпляру WebAssembly модуля. Об'єкт results містить властивість instance, яка представляє екземпляр модуля.
- const sum = addFunction(a, b);: У цьому рядку ми викликаємо функцію addFunction з переданими аргументами a та b.
- console.log('Сума:', sum);: В кінці ми виводимо результат обчислення на консоль.

2.2.3 Web assembly для ізольованих контекстів

Використовуючи Web assembly, цілком можливо реалізувати модуль для створення ізольованого контексту JavaScript. Web assembly дозволяє написати повноцінний движок, але це досить складне завдання.

Ви можете використовувати існуючі реалізації двигуна, такі як V8, SpiderMonkey або JavaScriptCore, або написати свій власний. Важливо пам’ятати, що написання власного механізму JavaScript є дуже складним завданням, а ще складнішим є підтримка нової версії Javascript для цього механізму

Після написання механізму JavaScript мовою, яка підтримує веб-складання, такої як C++ або Rust, або вибору існуючого механізму його потрібно скомпілювати в WebAssembly.

Після компіляції JavaScript двигуна у Web assembly модуль, його можливо завантажити в JavaScript модуль та запускати код користувача всередині двигуна який реалізован на Web assembly, і тому має свій глобальний контекст.

Написання власного JavaScript двигуна з використанням WebAssembly є дуже складною задачею. Це вимагає розуміння низькорівневих деталей WebAssembly та досвіду у розробці двигунів.

Підтримка WebAssembly може мінятися в залежності від браузера та різних версій. Тому доведеться враховувати цей факт під час розробки та забезпечувати альтернативні шляхи виконання для браузерів які непідтримують web assembly.

Налагодження коду, що виконується в WebAssembly, є більш складним завданням, ніж налагодження звичайного JavaScript коду. Інструменти для налагодження WebAssembly все ще розвиваються, і є менш функціональними та зручними в порівнянні з інструментами для налагодження і розробки на JavaScript.

2.3 Web worker

2.3.1 Загальний опис Web worker

Web Worker — це механізм веб-розробки, який виконує обчислення у фоновому потоці, окремому від основного потоку веб-сторінки. Основний потік відповідає за відображення веб-сторінок, взаємодію з користувачами та обробку подій, тоді як Web worker може виконувати важкі обчислення, обробку даних або інші трудомісткі завдання.

Він працює в окремому потоці, що є однією з його головних переваг. Відмова від блокування основного потоку веб-сторінки означає, що трудомісткі обчислення можуть виконуватися паралельно з відображенням веб сторінки, без впливу на її відзивчивість та взаємодію з користувачем.

Web Workers доступні в браузерах, які підтримують стандарт HTML5. Веб-воркери створюються за допомогою конструктора Worker() і виконують код, вказаний у JS файлі.

Після створення Web Worker може спілкуватися з основним потоком веб-сторінки та іншими Web Worker. Обмін повідомленнями здійснюється за допомогою методу postMessage() і події message.

Типи даних, якими можна обмінюватися між основним потоком і веб-воркерами, повинні бути серіалізовані, оскільки повідомлення передаються між різними потоками. Це означає, що їх потрібно перетворити на тип, який можна представити у форматі JSON. Наприклад, це можуть бути примітивні типи даних (рядки, числа, логічні значення), об'єкти, масиви тощо.

Веб-працівники мають власний глобальний об’єкт під назвою self. Цей об’єкт схожий на глобальний об’єкт window в головному потоці веб-сторінки. Об’єкт self існує в окремому веб-потоці та не може отримати доступ до глобальних об’єктів, змінних або функцій основного веб-потоку. Це допомагає уникнути конфліктів із глобальним контекстом основного потоку. Web Workers не мають прямого доступу до DOM документа HTML. Він доступний лише в головному потоці веб-сторінки.

Основні кроки в обміні повідомленнями між головним потоком і веб-воркером такі:

- Головний потік створює веб-воркера, вказуючи шлях до файлу, який містить JavaScript-код веб-воркера.
- Головний потік або інший веб-воркер можуть викликати метод postMessage() на об'єкті веб-воркера, передаючи дані як параметр.
- У веб-воркері потрібно встановити обробник події message, щоб отримати та обробити отримані повідомлення.
- Веб-воркер може відправити відповідь назад до головного потоку, викликавши postMessage() на об'єкті self.

- У головному потоці веб-сторінки слід встановити обробник події message на об'єкті веб-воркера, щоб отримати та обробити відповідь.

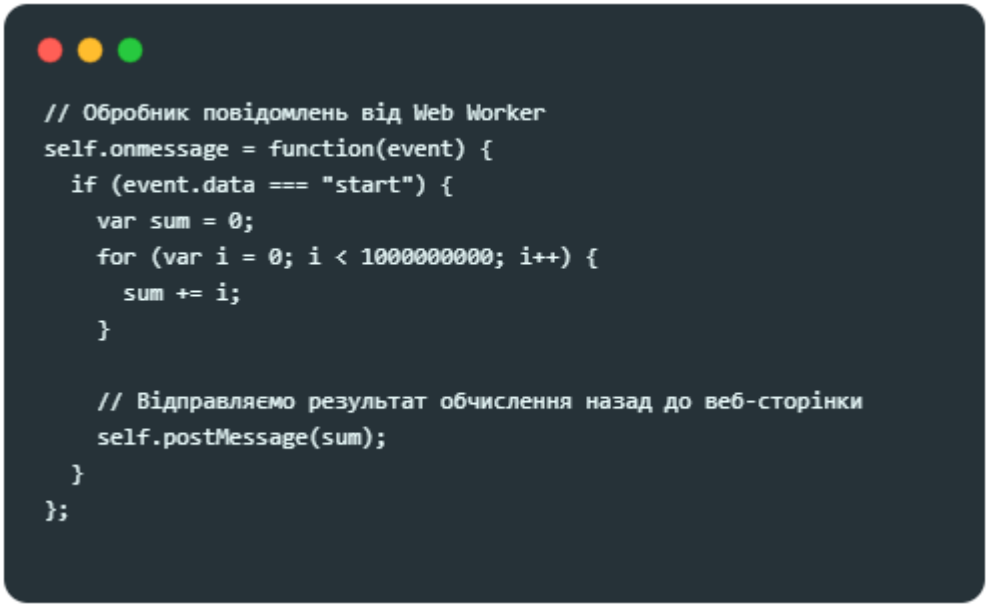
2.3.2 Приклад використання Web worker

У веб-сторінці ми створимо кнопку, яка буде викликати Web Worker для обчислення суми чисел великих розмірів.

```
<!DOCTYPE html>
<html>
<head>
  <title>Web Worker Example</title>
</head>
<body>
  <button onclick="startWorker()">Start Calculation</button>
  <p id="result">Result: </p>

  <script>
    var worker;

    function startWorker() {
      // Створюємо Web Worker
      worker = new Worker("worker.js");

      // Встановлюємо обробник події при отриманні повідомлення від
Web Worker
      worker.onmessage = function(event) {
        document.getElementById("result").textContent = "Result: " +
event.data;
      };

      // Відправляємо повідомлення до Web Worker
      worker.postMessage("start");
    }
  </script>
</body>
</html>
```

Рисунок 2.7 – Приклад Web worker в HTML файлі

Коли користувач натисне на кнопку, обчислення будуть виконуватися у фоновому потоці, не блокуючи головний потік веб-сторінки.



```
// Обробник повідомлень від Web Worker
self.onmessage = function(event) {
  if (event.data === "start") {
    var sum = 0;
    for (var i = 0; i < 1000000000; i++) {
      sum += i;
    }

    // Відправляємо результат обчислення назад до веб-сторінки
    self.postMessage(sum);
  }
};
```

Рисунок 2.8 – Приклад Web worker в JS файлі

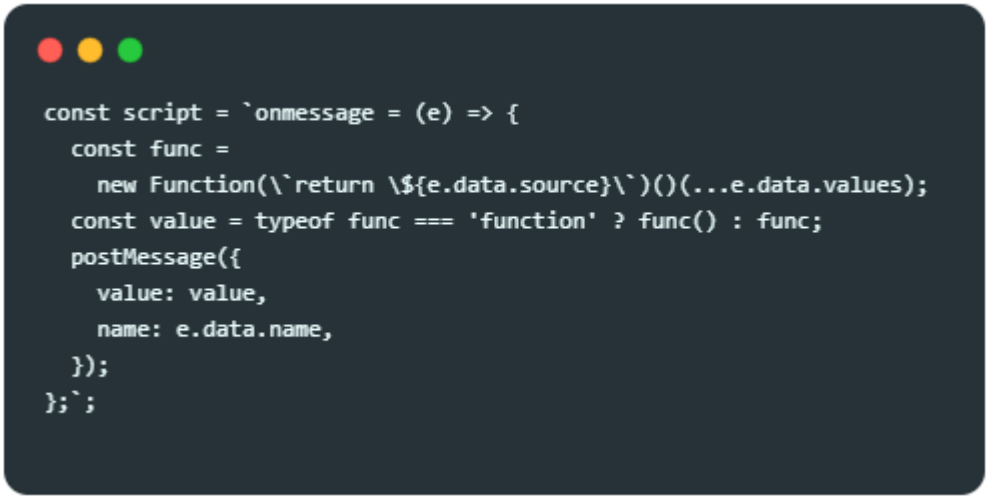
У прикладі Рисунок 2.7 та Рисунок 2.8, коли користувач натискає на кнопку "Start Calculation", створюється Web Worker, який обчислює суму чисел від 0 до 999999999. Після завершення обчислення, Web Worker надсилає повідомлення з результатом назад до веб-сторінки, яке відображається у розділі з id "result".

2.3.2 Прототип реалізіації ізольованого контексту Web worker

API прототипа складається з двох простих функій:

1. createContext(context) - створює ізольований контекст
2. runInContext(src, context) – запускає код в створеному ізольованому контексту

Основою реалізації цього прототипа є код який запускається в самому web worker та клас VMWorker який створює, налаштовує воркер і запускає в ньому код користувача.



```
const script = `onmessage = (e) => {
  const func =
    new Function(\`return \${e.data.source}\`)()(...e.data.values);
  const value = typeof func === 'function' ? func() : func;
  postMessage({
    value: value,
    name: e.data.name,
  });
};`;
```

Рисунок 2.9 - Код воркера

Код з Рисунок 2.9 запускається всередині воркера. Він приймає повідомлення від основного потоку, виконує функцію, яка передається як рядок e.data.source разом з своїми параметрами e.data.values. Результат виконання функції надсилається назад до основного потоку за допомогою postMessage().

- onmessage = (e) => { ... }: Ця частина визначає обробник події onmessage, який викликається, коли воркер отримує повідомлення.
- const func = new Function(\return ${e.data.source}`)()(...e.data.values): Цей рядок створює нову функцію з використанням конструктора Function. Вміст функції передається у вигляді рядка, який отримується з e.data.source. Рядок return ${e.data.source}використовує шаблонні рядки для вставки значення e.data.source. Після створення функції вона викликається зі значеннями (...e.data.values). Отримане значення зберігається у змінній func.
- const value = typeof func === 'function' ? func() : func: Цей рядок перевіряє тип func. Якщо тип - function, то func викликається як функція, і отримане значення зберігається у змінній value. У

протилежному випадку, якщо тип не є function, func саме стає значенням value.

- postMessage({ value: value, name: e.data.name }): Нарешті, цей рядок відправляє повідомлення назад до основного потоку за допомогою функції postMessage(). Об'єкт, який містить результат виконання коду користувача.

Реалізація, яка використовує Web Worker для створення ізольованого контексту та виконання коду, має свої переваги і недоліки. Основним плюсом є надійна ізоляціє контексту тому що він працює у відокремленому потоці. Основними мінусами є

- Веб-воркери не мають прямого доступу до DOM, що означає, що вони не можуть взаємодіяти з елементами сторінки, змінювати їх стилі або отримувати їх властивості. Це обмеження ускладнює виконання деяких завдань, які потребують маніпуляцій з DOM.
- Веб-воркери не можуть повертати посилання на об'єкти або функції, оскільки вони працюють у відокремленому контексті. Це означає, що вони не підтримують обмін даними з основним потоком через передачу посилань на функції та об'єкти.
- Код, що виконується у веб-воркері, працює асинхронно. Це означає, що немає прямого способу синхронізувати його виконання з основним потоком. Якщо потрібно отримати результат виконання коду з веб-воркера, потрібно використовувати обробники подій або Promise.

Загалом, використання веб-воркерів для створення ізольованого контексту та виконання коду має свої переваги, але також приносить досить значні обмеження.

2.4 Iframe

2.4.1 Загальний опис Iframe

Iframe (Inline Frame) - це елемент HTML, який дозволяє вставляти одну веб-сторінку в іншу веб-сторінку. Він створює пряме вбудоване вікно на поточній сторінці, яке може відображати вміст з інших джерел.

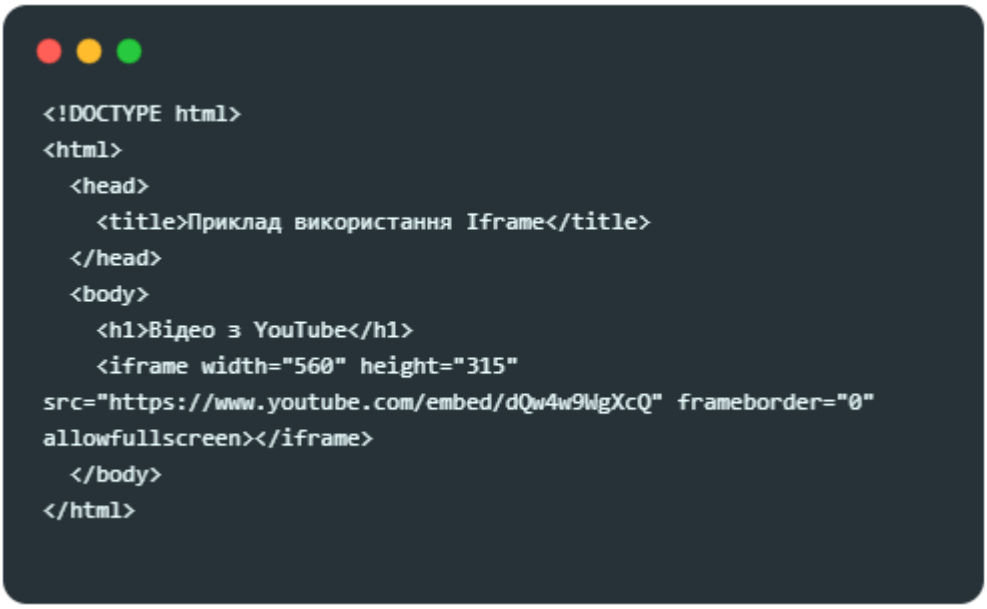
Iframe є потужним інструментом, оскільки він дозволяє поєднувати декілька різних джерел на одній сторінці. Він широко використовується для вбудовування карт, відео, аудіо, каналів соціальних мереж, веб-документів та інших ресурсів.

Глобальний контекст Iframe відноситься до JavaScript-середовища, яке створюється при завантаженні вмісту Iframe. Кожен Iframe має свій власний глобальний контекст, який може мати власні змінні, об'єкти та функції.
Це означає, що змінні та об'єкти, визначені в глобальному контексті Iframe, не будуть доступні в глобальному контексті батьківської сторінки і навпаки.

Кожен контекст має власну незалежну область пам'яті. Iframe має власну область виконання, де інтерпретується та виконується код JavaScript. Це дозволяє iframe мати власну логіку, обробляти події та взаємодіяти з користувачем незалежно від батьківської сторінки.

2.4.2 Приклад використання Iframe

Приклад Рисунок 2.10 демонструє, як Iframe можна використовувати для вбудовування контенту з іншого джерела, такого як YouTube, на веб-сторінку. Можно використовувати Iframe для вбудовування інших типів контенту, таких як карти, соціальні медіа-стрічки або вміст з інших веб-сторінок.



```
<!DOCTYPE html>
<html>
  <head>
    <title>Приклад використання Iframe</title>
  </head>
  <body>
    <h1>Відео з YouTube</h1>
    <iframe width="560" height="315"
src="https://www.youtube.com/embed/dQw4w9WgXcQ" frameborder="0"
allowfullscreen></iframe>
  </body>
</html>
```

Рисунок 2.10 – Приклад використання Iframe

Створюється сторінка з заголовком "Відео з YouTube". Внутрішнім вмістом сторінки є Iframe, який вбудовує відео з YouTube. URL https://www.youtube.com/embed/dQw4w9WgXcQ вказує на відео, яке буде відображено у Iframe. Атрибути width та height визначають розміри Iframe, щоб відео було відображено зі специфікованими розмірами. frameborder="0" встановлює рамку Iframe на значення "0", щоб вона не відображалась.

2.4.3 Прототип реалізіації ізольованого контексту Iframe

Так само як і прототип з використанням Web worker, API має дві функції createContext(context), runInContext(src, context) які створюють ізольований контекст та виконують код користувача в ньому.

```
const createContext = (userContext) => {
  const iframe = document.createElement('iframe');
  iframe.style.display = 'none';
  document.body.appendChild(iframe);
  const iframeContext = iframe.contentWindow;
  for (const [key, value] of Object.entries(userContext)) {
    iframeContext[key] = value;
  }
  return iframeContext;
};
```

Рисунок 2.11 – Приклад створення Iframe контексту

Функція createContext приймає об'єкт userContext як параметр і створює ізольований контекст за допомогою Iframe.

- Спочатку функція створює новий елемент Iframe, використовуючи метод document.createElement('iframe').
- Функція задає атрибут style.display на none, що означає, що Iframe не буде видимим на сторінці.
- Функція використовує document.body.appendChild(iframe) для додавання Iframe до тіла сторінки.
- За допомогою iframe.contentWindow отримується доступ до глобального контексту Iframe. Це дозволяє взаємодіяти з Iframe, виконувати JavaScript-код і працювати з його змінними та функціями.
- Функція копіює змінні з userContext до контексту Iframe. Вона використовує Object.entries(userContext) для отримання масиву ключ-значення з об'єкта userContext і цикл for...of для визначення кожної пари ключ-значення до відповідних змінних у контексті Iframe.

- В кінці функція повертає контекст Iframe, що дозволяє зовнішньому коду взаємодіяти з Iframe, виконувати код в його контексті та отримувати доступ до присвоєних змінних.

```
class Script {
  constructor(src) {
    this.name = name;
    this.src = src;
  }

  runInContext(context) {
    return context.eval(this.src);
  }
}
```

Рисунок 2.12 - Приклад класу Script в парі з використанням Iframe контексту

Метод runInContext(context) виконує код скрипту в Iframe контексті, переданому як аргумент context. Він використовує метод eval() контексту для виконання коду, що міститься в властивості src скрипту. Метод eval() виконує переданий рядок коду як JavaScript-код у зазначеному контексті.

Функція eval() може виконувати динамічно створений або отриманий JavaScript-код. Вона приймає рядок з кодом як аргумент і виконує його в контексті виклику. Наприклад, якщо ви викликаєте context.eval('2 + 2'), вона обчислить вираз 2 + 2 і поверне результат, який у цьому випадку буде 4.

Одже Iframe дозволяє виконувати код в ізольованому контексті, реалізація є достатньо простою у порівнянні з Веб-воркером. Є можливість виконувати код користувача синхронно та асинхронно. Цей прототип також дозволяє повертати посилання на функії та об'єкти з ізольованого контексту

ВИСНОВОК ДО РОЗДІЛУ 2

У цьому розділі розглядаються різні підходи до реалізації ізольованого контексту JavaScript для віртуальної машини, що працює в браузері. Аналіз виявив кілька підходів, таких як використання Web assebmly, Web worker та iframe, кожен зі своїми перевагами та обмеженнями.

WebAssembly — це потужний інструмент, але створити з ним власний механізм JavaScript може бути складно. Такий підхід може вимагати від розробника великих зусиль і досвіду. Однак, якщо такий механізм буде успішно реалізований, він зможе забезпечити високошвидкісне виконання коду JavaScript в ізольованому контексті.

Зі свого боку веб-працівники пропонують можливість виконувати код JavaScript у фоновому потоці, що дозволяє уникнути блокування основного потоку браузера. Однак веб-працівники мають обмежену можливість обмінюватися даними з основним потоком. Неможливість повернути посилання на функції та об’єкти від веб-воркерів і відсутність доступу до DOM обмежують їх використання в певних сценаріях. Крім того, неможливо синхронно виконати код користувача в контексті веб-воркерів.

Iframe — це дуже простий у реалізації метод створення ізольованих контекстів JavaScript. Він надає можливість повертати посилання на функції та об’єкти з iframe, а також надає доступ до DOM. Iframe також дозволяє синхронне та асинхронне виконання коду користувача, що робить iframe більш гнучким у порівнянні з Web worker

Отже, дослідивши різні підходи до реалізації ізольованих контекстів JavaScript, ми можемо зробити висновок, що кожен підхід має свої переваги та обмеження. WebAssembly може забезпечити високу продуктивність, але складність його реалізації може бути проблемою. Web Workers дозволяють виконувати код у фонових потоках, але мають обмежені можливості обміну

даними та обмеження доступу до DOM. iframe простий у використанні та забезпечує повний доступ до функцій, об’єктів і DOM, що робить його зручним вибором для реалізації ізольованих контекстів.

На додаток до методів, розглянутих вище, модуль Node.js vm також розглядався як частина дослідження. Цей модуль надає API для створення та керування віртуальними контекстами JavaScript у середовищі Node.js. Модуль vm Node.js має добре розроблений API для створення ізольованих контекстів JavaScript. Використовуючи цей модуль, ви можете створити новий контекст і виконати в ньому код JavaScript. Він надає можливість контролювати доступ до глобальних об'єктів, змінювати контекст виконання та встановлювати обмеження на виконання певних операцій.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

3.1 Огляд проекту

Проаналізувавши інструменти за допомогою який можливо стоворити ізольований контекст в браузері, які були представлені в розділі 2, вибір був зроблений на користь Iframe як головного механізму для створення ізольованого контексту.

Дизайн API був створений на основі бібліотеки vm із Node.js. Він є достатньо простим, задоволняє всім вимогам та вже є знайомим багатьом розробникам.

API складається з всього 3 функій: createContext, isContext, runInContext та одного класу Script.

- createContext(context) – отримує об'єкт властивості якого мають стати глобальними властивостями всередині створеного ізольованого контексту. Повертає об'єкт який має всі передані властивості, які вже зв'язані з властивостями всередині ізольованого контексту.
- isContext(contesxt) – отримує об'єкт який має бути контекстом, тобто створеним функцією createContext. Перевіряє що переданий об'єкт дійсно є контекстом.
- Script – клас який зберігає в собі код користувача який буде виконаний в ізольованому контексті та має метод для виконання кода в ізольованому контексті
- runInContext – отримує код користувача та контекст. Виконує код користувача в заданому контексті.

Всього проект складається з 4 модулів: script, vm, realm, context про деталі якиї буде розказано далі в цьому розділі.

3.2 Модуль Realm

3.2.1 Концеція Realm в ECMAScript

Концепція Realm взята зі специфікації ECMAScript.

Realm в специфікації ECMAScript — це концепція, яка використовується для опису абстрактного середовища виконання, в якому функції, об’єкти та інші мовні конструкції можуть існувати та взаємодіяти один з одним. Це незалежний контекст виконання з власними властивостями та поведінкою.

Він визначає, як об’єкти, функції та інші компоненти мови взаємодіють один з одним. Він включає глобальні об’єкти, змінні, функції, області та інші елементи, які впливають на виконання коду. Кожен Realm має власну пам’ять для зберігання об’єктів і даних, а також має доступ до інших Realm, що дозволяє спілкуватися між ними.

Realm також визначає правила для створення, ініціалізації та керування об’єктами та функціями. Наприклад, він визначає, як створюються нові об’єкти, виконуються функції та вирішуються конфлікти імен.

Ця концепція дозволяє ізолювати виконання коду в різних середовищах, таких як браузери, Node.js або інші реалізації ECMAScript. Це допомагає забезпечити передбачуване та безпечне виконання коду, оскільки кожна область має власну область дії та незалежну пам’ять.

Підсумовуючи, Realm у специфікації ECMAScript — це абстрактне середовище виконання, яке визначає, як об’єкти, функції та інші компоненти мови взаємодіють один з одним, а також визначає правила для створення, ініціалізації та керування цими компонентами. Використання концепцій Realm

допомагає забезпечити передбачуване та безпечне виконання коду в різних середовищах.

3.2.2 Патерн Замісник

Клас Realm по своей суті є реалізацією патерна proxy (замісник).

Замісник (Proxy) - це структурний шаблон проектування, який надає спеціальний проксі-об’єкт для керування доступом до реального об’єкта.

Цей шаблон підходить для ситуацій, які вимагають додаткового контролю або потребують додатковий можливостей під час доступу до об'єкта. Це дозволяє вам замінити основний об’єкт, щоб виконувати різні операції до, або після над основним об’єктом, не змінюючи його код.

У паттерні Proxy присутні дві основні складові:

- Замісник (Proxy): це клас, що реалізує інтерфейс, який використовується клієнтом для доступу до основного об'єкта. Він зберігає посилання на об'єкт-суб'єкт (справжній об'єкт) та може мати додаткову логіку передачі запитів до нього.
- Суб'єкт (Subject): це клас, який представляє справжній об'єкт, до якого здійснюється доступ через проксі. Цей клас визначає спільний інтерфейс, який реалізує як проксі, так і справжній об'єкт.

Коли клієнт взаємодіє з замісником, він передає йому свої запити. У свою чергу, замісник може виконати додаткову логіку перед тим, як передати запит реальному об'єкту.

Шаблон Proxy дозволяє вам розділити обов’язки замісником і реальним об’єктом, додати додаткові функції або контролювати доступ до об’єкта. Це також спрощує взаємодію з об’єктами, оскільки клієнти використовують замісник, не знаючи про існування реального об’єкта.

3.2.3 Методи класа Realm

- set(key, options = {}): Визначає властивість за переданим ключем та опціями в ізольованому контексті
- defineDeclarative(key, attributes): Визначає декларативну властивість за переданим ключем та атрибутами в ізольованому контексті.
- defineHiddenProperty(key, value): Визначає приховану властивість за переданим ключем та значенням в ізольованому контексті.
- defineProperty(key, attributes): Визначає властивість за переданим ключем та атрибутами в ізольованому контексті.
- get(key): Повертає значення властивості за переданим ключем.
- has(key): Перевіряє, чи існує властивість за переданим ключем.
- hasDeclaratives(): Перевіряє, чи існують декларативні властивості.
- isProperty(key): Перевіряє, чи існує властивість за переданим ключем.
- isDeclarative(key): Перевіряє, чи існує декларативна властивість за переданим ключем.
- isWritableProperty(key): Перевіряє, чи є властивість за переданим ключем змінною.
- isRedefined(key): Перевіряє, чи була властивість змінена

3.2.4 Дескриптор властивостей

Щоб зрозуміти, як працює клас Realm і чому ми не можемо просто використовувати об’єкт глобального контексту iframe, необхідно розуміти, що всі глобальні змінні є лише властивостями глобального об’єкта, а це означає, що кожна з них має свій власний дескриптор.

У JavaScript дескриптори властивостей використовуються для опису та керування поведінкою властивостей об’єктів. Дескриптори властивостей

дозволяють установлювати різні аспекти властивості, такі як читання, запис, перерахування та видалення, забезпечуючи зручність і гнучкість у управлінні.

Кожна властивість у JavaScript має пов’язаний з нею дескриптор властивості, який складається з таких компонентів:

- Value (значення): Вказує на поточне значення властивості. Може бути будь-яким типом даних, включаючи примітиви або об'єкти.
- Writable (змінність): Вказує, чи можна змінювати значення властивості. Якщо встановлено в true, значення може бути змінено, а якщо встановлено в false, значення залишається незмінним.
- Enumerable (перераховуваність): Вказує, чи буде властивість видимою під час перебору властивостей об'єкта. Якщо встановлено в true, властивість буде видимою, інакше вона буде прихована під час перебору.
- Configurable (конфігурованість): Вказує, чи можна змінювати дескриптор після його встановлення. Якщо встановлено в true, можна змінювати будь-який аспект дескриптора, а якщо встановлено в false, то дескриптор стає незмінним, і більше не можна змінювати його конфігурацію або видаляти властивість.

Для роботи з дескрипторами властивостей JavaScript надає низку вбудованих методів, таких як Object.defineProperty(), Object.defineProperties(), Object.getOwnPropertyDescriptor(), які дозволяють вам установлювати, отримувати або змінювати дескриптори властивостей об’єктів.

Дескриптори властивостей використовуються для різних цілей, таких як налаштування доступу до властивостей, керування значеннями або захист властивостей від неправильного використання. Наприклад, ви можете зробити певні властивості недоступними для читання або запису або приховати їх від зовнішнього коду для забезпечення безпеки та інкапсуляції.

3.2.5 Незмінні властивості в глобальному контексті

В браузерному контексті існують деякі незмінні властивості, які можна використовувати для доступу до різних функцій та інформації. Ось кілька найважливіших незмінних властивостей:

- window: Об'єкт window представляє головне вікно браузера або вкладку, в якій відкрита поточна сторінка. Він надає доступ до багатьох функцій, таких як робота з документом, управління таймерами, відкриття нових вікон і т.д.
- document: Це об'єкт, який представляє веб-сторінку, відображену в поточному вікні. Він дозволяє отримувати доступ до елементів сторінки, змінювати їх вміст, створювати нові елементи, обробляти події та інше.
- navigator: Об'єкт navigator містить інформацію про браузер, в якому відображається сторінка, таку як назва браузера, версія, мова браузера, наявність підтримки певних функцій (наприклад, геолокації) та інші характеристики.
- location: Об'єкт location містить інформацію про URL-адресу поточної сторінки і надає можливість отримувати доступ до різних її частин, таких як протокол, хост, шлях, параметри і т.д. Він також дозволяє перенаправляти користувача на іншу сторінку.
- history: Об'єкт history зберігає історію переходів між сторінками в рамках поточного вікна браузера. Він дозволяє навігувати назад і вперед по цій історії, а також керувати записами в ній.

Наприклад, спроба змінити значення window на інше значення, таке як 1, буде призводити до помилки. Давайте розглянемо цей приклад.

```
window = 1;
console.log(window);
```

Рисунок 3.1 - Спроба змінити властивість window

У випадку спроби зміни значення window на 1, консоль поверне помилку

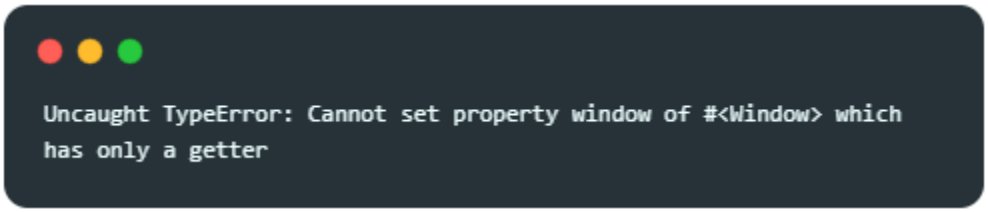


Рисунок 3.2 - Помилка при спробі змінити властивість window

Це обмеження існує для забезпечення безпеки браузерного середовища і унеможливлення потенційно шкідливої поведінки. Аналогічно, інші незмінні властивості, такі як document, navigator, location та history, також не можна перезаписувати.

Механізм, який недозволяє зміну наведених властивостей в JavaScript, базується на концепції дескрипторів властивостей. Дескриптори цих властивостей встановлені таким чином, що забороняють їхню зміну.

Коли атрибут writable (змінність) властивості встановлений на false, то об'єкт відкидає будь-яку спробу змінити значення властивості. Це забезпечує стабільність і незмінність цих властивостей в браузерному контексті.

3.2.6 Концепція declaratives в ECMAScript

В специфікації ECMAScript, Realm має різні середовища для зберігання об'єктів глобального контексту та зміних або функцій які визначені за допомогою спеціальних ключових слів.

ObjectRecord та DeclarativeRecord є внутрішніми записами, які використовуються в двигуні JavaScript для зберігання інформації про змінні та їх значення в контексті виконання коду.

ObjectRecord представляє запис для об'єкта, який містить змінні, оголошені як властивості об'єкта або методи, що належать об'єкту. Внутрішній запис ObjectRecord пов'язує ім'я змінної з відповідним значенням. Звернення до змінної через ObjectRecord передбачає доступ до властивості об'єкта за ім'ям.

Це зазвичай використовується для глобальних змінних, властивостей об'єктів та лексичних змінних, які були взяті з зовнішніх областей.

DeclarativeRecord представляє запис для змінних, оголошених за допомогою ключових слів var, let або const. Кожна змінна має відповідний внутрішній запис DeclarativeRecord, який містить інформацію про ім'я змінної, її значення та інші властивості. DeclarativeRecord зберігається в області дії, яка має доступ до змінних, оголошених в цій області.

Наприклад, розглянемо наступний код:

```
var x = 10;
let y = 20;

function example() {
  var z = 30;
  console.log(x + y + z);
}

example();
```

Рисунок 3.3 - Приклад для ObjectRecord та DeclarativeRecord

У прикладі Рисунок 3.3, ObjectRecord буде містити інформацію про глобальну змінну x, яка є властивістю глобального об'єкта, та локальну змінну y, яка була оголошена з використанням let. Звернення до x і y в функції example буде виконуватись через ObjectRecord.

DeclarativeRecord буде містити інформацію про локальну змінну z, оголошену в функції example.

Обидва типи внутрішніх записів, ObjectRecord та DeclarativeRecord, використовуються двигуном JavaScript для доступу до змінних та їх значень під час виконання коду.

3.2.7 Конструювання Realm

Розгянемо конструктор класу Realm.

```
class Realm {
  constructor(globalObject, options = {}) {
    this.realmOptions = options;
    this.properties = globalObject;
    this.declaratives = {};
    this.externals = [];
    this.eval = globalObject.eval;
    this.defineHiddenProperty(GLOBAL_THIS, globalObject);
  }
}
```

Рисунок 3.4 - Конструктор класу Realm

У конструкторі створюється екземпляр класу Realm з наступними властивостями:

- properties - об'єкт globalObject, який є глобальним об'єктом window, дістаним із Iframe. Цей об'єкт відповідає ObjectRecord із специфікації ECMAScript
- declaratives - порожній об'єкт, який буде використовуватися для зберігання декларативних властивостей, про реалазіцію яких буде розказано пізніше. Цей об'єкт відповідає DeclarativeRecord із специфікації ECMAScript
- externals - порожній масив, який буде використовуватися для зберігання імен всіх властивостей глобального контексту переданих користувачем.
- eval - посилання на глобальну функцію eval з глобального об'єкта window із Iframe. Вона дай можливість виконувати код всередині глобального об'єкта Iframe.

В кінці конструктора робиться виклик метода defineHiddenProperty. Для того щоб зберігти посилання на глобальний об'єкт всередині самого глобального об'єкт. Це потрібно тому що користувач може перезаписати його своїм значанням.

3.2.8 Визначення властивостей в Realm

Метод set відповадає за визначення властивостей в Realm. Саме цей метод виконується коли користувач додає властавість до глобального контексту.

```
set(key, options = {}) {
  const descriptor = {
    value: options.value,
    writable: options.writable ?? true,
    enumerable: options.enumerable ?? true,
    configurable: options.configurable ?? true,
  };
  if (this.isWritableProperty(key)) {
    this.defineProperty(key, descriptor);
  } else {
    this.defineDeclarative(key, descriptor);
  }
}
```

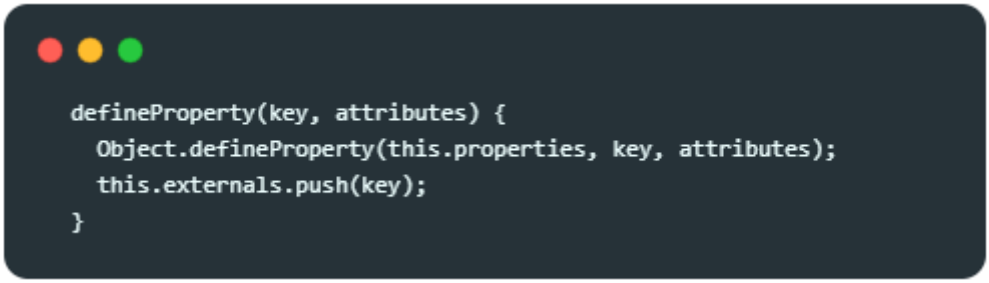
Рисунок 3.5 - Метод set класа Realm

Метод set перевіряє, чи можна змінювати (записувати) властивість з вказаним ключем, використовуючи метод isWritableProperty. Якщо це можливо, то метод defineProperty використовується для визначення нової властивості з вказаним ключем і дескриптором. В іншому випадку, коли властивість не може бути зміненою, викликається метод defineDeclarative, який визначає декларативну властивість за вказаним ключем і дескриптором.

У методі set створюється об'єкт descriptor на основі переданих параметрів options. Якщо параметр writable не вказаний, то встановлюється значення true за замовчуванням. Те саме стосується і параметра enumerable та configurable.

Остаточно, метод set додає нову властивість до контексту, використовуючи один з двох методів: defineProperty або defineDeclarative, залежно від того, чи можна змінювати властивість з вказаним ключем.

Визначення властивості є дуже простою операцією



.

Рисунок 3.6 - Визначення властивості

Властивість визначається за допомогою вбудованої функції Object.defineProperty та її імя зберігається в масиві externals.

3.2.9 Backing fields в мові програмування Kotlin

Для реалізації декларативів була запозичена ідея із мови програмуваня Kotlin, за домогою якої там реалізовуються властивості класу.

У Kotlin, backing field (опорне поле) є спеціальним механізмом, який використовується для забезпечення доступу до властивостей класу. Він дозволяє зберігати значення властивості і забезпечувати потрібну логіку для доступу до цього значення.

Коли ви оголошуєте властивість, компілятор автоматично генерує backing field за кулісами. Він створюється для кожної властивості, яка має явно вказаний access modifier (наприклад, private, protected, internal або public) або має не стандартний getter або setter.

```
class Example {
    var value: Int = 0
        get() {
            println("Getting the value")
            return field
        }
        set(newValue) {
            println("Setting the value to $newValue")
            field = newValue
        }
}
```

Рисунок 3.7 - Приклад backing fields в Kotlin

У прикладі Рисунок 3.7 для властивості value компілятор автоматично створить backing field з іменем field. В методі get() ми можемо звертатися до backing field через field, і аналогічно в методі set() можна присвоювати значення field. При використанні властивості value з об'єкта Example, компілятор автоматично викликатиме методи get() і set() із відповідним доступом до backing field.

3.2.10 Визначення властивостей в Realm

Метод defineDeclarative в класі Realm використовується для визначення декларативної властивості в реалмі. Він отримує два параметри: key (ключ) і attributes (атрибути властивості).

```
defineDeclarative(key, attributes) {
  const backingKey = `__${key}__`;
  const value = attributes.value;
  this.declaratives[key] = { backingKey, attributes };
  this.defineHiddenProperty(backingKey, value);
  this.externals.push(key);
}
```

Рисунок 3.8 - Метод defineDeclarative в класу Realm

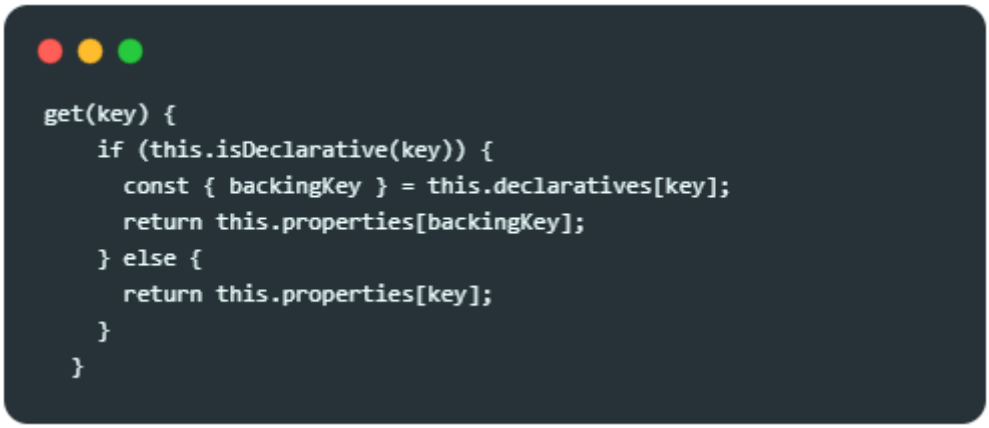
Метод defineDeclarative виконує наступні кроки:

- Створюється змінна backingKey, яка містить значення __ key__, що є ключем для зберігання значення декларативної властивості всередині контексту.
- Значення декларативної властивості отримується з атрибуту value переданого дискриптора який називається attributes.
- Властивість з ключем key додається до об'єкта declaratives, де зберігається посилання на ключ backingKey та дескриптор або інакше кажучи атрибути властивості.
- Викликається метод defineHiddenProperty з аргументами backingKey і value, щоб визначити прихований backing field у контексті
- Назва властивості зберігається в масиві externals, щоб вказати, що ця властивість додана користувачем

Отже, метод defineDeclarative визначає декларативну властивість в контексті, створюючи backing field для зберігання її значання всередині контексту після чого всі backing field оброблюються у модулі script і виглядають для користувача наче властивості контексту. Детальніше про це в розділі Модуль Script.

3.2.11 Отримання властивості в Realm

Метод get в класі Realm використовується для отримання значення властивості за вказаним ключем.



```
get(key) {
    if (this.isDeclarative(key)) {
      const { backingKey } = this.declaratives[key];
      return this.properties[backingKey];
    } else {
      return this.properties[key];
    }
  }
```

Рисунок 3.9 - Метод get класа Realm

Метод get перевіряє, чи властивість з вказаним ключем є декларативною за допомогою методу isDeclarative. Якщо так, то отримується ключ backingKey, який зберігається в об'єкті declaratives для вказаної декларативної властивості. Далі, метод повертає значення backing field цієї декларативної властивості за ключем backingKey.

Якщо властивість не є декларативною, метод просто повертає значення властивості з об'єкта properties за вказаним ключем.

Отже, метод get дозволяє отримати значення властивості за ключем в реалмі, будь то декларативна властивість або звичайна властивість глобального об'єкту.

3.3 Модуль Context

3.3.1 Огляд модуля Context

Модуль context відповідає за створення контексту: Iframe та Realm.

Модуль складається з 3 функцій: createContext, isContext, getRealm.

- createContext – створює контекст, що включає створення Iframe та Realm
- isContext – перевіряє що переданий об'єкт дійсно є контекстом
- getRealm – внутрішний метод який дозволяє отримати Realm

3.3.2 Створення контексту

Функція createContext створює новий контекст для виконання коду у ізольованому середовищі.

```
const createContext = (context, options = {}) => {
  const iframe = createIframe(document, options.name);
  const realm = new Realm(iframe.contentWindow, options);
  for (const key of Object.keys(context)) {
    const attributes = Reflect.getOwnPropertyDescriptor(context, key);
    realm.set(key, attributes);
  }
  const contextObject = createContextifiedObject(copyRoot(context), realm);
  contexts.set(contextObject, realm);
  cleanupRegistry.register(contextObject, { document, id: iframe.id });
  return contextObject;
};
```

Рисунок 3.10 - Функція createContext

У функції спочатку створюється iframe елемент за допомогою функції createIframe. Цей iframe елемент використовується для створення ізольованого середовища для виконання коду користувача.

Далі створюється новий об'єкт realm класу Realm з використанням iframe.contentWindow як глобального об'єкту.

Потім проходить ітерація по ключам об'єкту context, і для кожного ключа отримуються атрибути за допомогою Reflect.getOwnPropertyDescriptor. Отримані атрибути додаються до realm за допомогою realm.set(key, attributes).

Далі створюється новий об'єкт contextObject. Цей замісник перехоплює доступ до властивостей context і забезпечує доступ до властивостей в realm за допомогою методів realm.get(key) і realm.set(key, attributes).

3.3.3 Зберігання та очищення Iframe

Об'єкт contextObject разом зі своїм realm зберігаються в contexts, який є WeakMap. WeakMap є вбудованим класом у JavaScript, який надає можливість створення асоціативного масиву, де ключами можуть бути тільки об'єкти, а значеннями - довільні дані. Однак, у відміну від звичайного Map, WeakMap не заважає збирачу сміття (garbage collector) збирати ключі, якщо вони стають недосяжними з іншої частини коду. Це зроблено для підтримки автоматичного очищення пам'яті та запобігання витокам пам'яті.

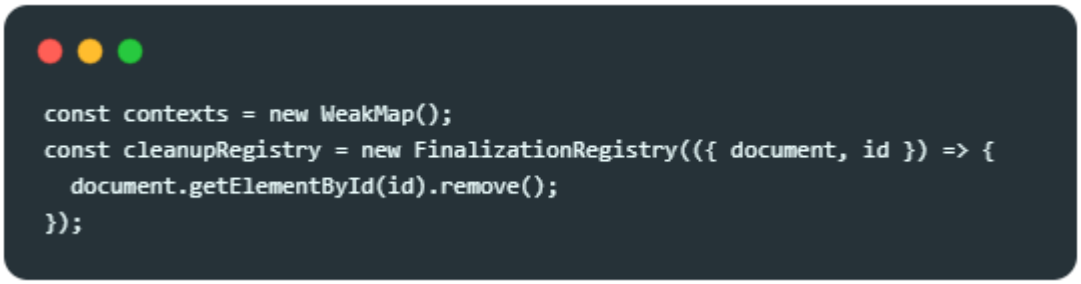


Рисунок 3.11 - Використання WeakMap та FinalizationRegistry

contextObject використовується як ключ, а realm - як відповідне значення. Коли contextObject стає недосяжним з іншої частини коду (наприклад, коли відповідна змінна виходить за межі області видимості або при знищенні контексту), contextObject автоматично видаляється з WeakMap разом зі своїм realm. Це дозволяє збирачеві сміття видалити contextObject і всі об'єкти realm, що пов'язані з ним, звільнивши пам'ять.

Щоб забезпечити видалення iframe елементу при недосяжності contextObject, використовується FinalizationRegistry. FinalizationRegistry дозволяє виконувати певний зворотний виклик (callback) при знищенні об'єкта, який був зареєстрований. Коли створюється новий iframe елемент, він реєструється у cleanupRegistry разом з contextObject. Коли contextObject стає недосяжним, FinalizationRegistry автоматично викликає вказаний зворотний виклик, де document і id отримуються з параметрів реєстрації. Зворотний виклик видаляє iframe елемент за допомогою вбудованої функції. Це гарантує, що iframe буде видалено, коли відповідний contextObject стає недосяжним.

Отже, за допомогою поєднання WeakMap та FinalizationRegistry, функція createContext забезпечує автоматичне видалення iframe елементу, коли відповідний контекст стає недосяжним, що сприяє управлінню пам'яттю та запобіганню витокам пам'яті.

3.4 Модуль Script

3.4.1 Клас Script

Клас Script зберігає код користувача, та виконує його в ізольованому контексті за допомогою метода runInContext.

```
runInContext(context) {
  if (!isContext(context)) {
    throw new Error('options.context - is not contextified object');
  }
  const realm = getRealm(context);
  if (realm.hasDeclaratives()) {
    proxifyGlobals(realm);
  }
  const script = enchanceScript(realm, this.src);
  return realm.eval(script);
}
```

Рисунок 3.12 - Метод runInContext класа Script

В методі runInContext спочатку перевіряється, чи є переданий context дійсно контекстом, викликом функції isContext. Якщо це не так, генерується помилка Error.

За допомогою функції getRealm(context) отримується контекст realm який належить до переданого context.

Перевіряється, чи є в контексті декларативи за допомогою методу hasDeclaratives() об'єкта realm. Якщо вони є то, виконується функція

proxifyGlobals(realm), яка огортає всі властивосі які посилаються на глобальний об'єкт замісником, який перенаправляє всі виклика до realm.

enchanceScript(realm, this.src) – видозмінює код користувача визначаючи змінні (const або let) які є декларативами.

Виконується метод eval() об'єкта realm зі складеним скриптом. Та повертається результат виконання скрипту.

3.4.2 Об'єкт Proxy в Javascript

Для розуміння реалізації функції proxifyGlobals потрібно знати про об'єкт proxy в Javascript

У JavaScript proxy — це об’єкт, який дозволяє перехоплювати та контролювати доступ до інших об’єктів. Він забезпечує зручний механізм для зміни поведінки об’єктів і динамічного впливу на них.

Замісник використовується для перехоплення різних операцій, таких як доступ до властивостей об’єкта, виклик методів, обхід об’єктів тощо. Це дає змогу виконувати певні перевірки, фільтри або змінювати дані до того, як вони досягнуть цільового об’єкта.

```
// Цільовий об'єкт
const target = {
  name: 'John',
  age: 30
};

// Створення проксі
const handler = {
  get: function(target, property) {
    console.log(`Читається властивість ${property}`);
    return target[property];
  },
  set: function(target, property, value) {
    console.log(`Записується властивість ${property} зі значенням
${value}`);
    target[property] = value;
  }
};

const proxy = new Proxy(target, handler);

console.log(proxy.name); // Читається властивість name, виводиться "John"
proxy.age = 35; // Записується властивість age зі значенням 35
console.log(proxy.age); // Читається властивість age, виводиться 35
```

Рисунок 3.13 - Приклад використання Proxy

У прикладі Рисунок 3.13 ми створюємо proxy, який перехоплює доступ до властивостей об'єкта target. У об'єкті handler ми визначаємо два методи: get та set, які викликаються при читанні та записуванні властивостей об'єкта відповідно. Коли ми звертаємось до властивостей name та age замісник, викликаються відповідні методи проксі, які виводять повідомлення у консоль та обробляють доступ до властивостей.

Замісник також підтримує багато інших методів, які можуть бути використані для перехоплення різних операцій, таких як виклик функцій (apply), перевірка властивостей (has), перебір властивостей (ownKeys), видалення властивостей (deleteProperty) та багато іншого.

3.4.3 Огортання глобальних об'єктів замісником

Існує чотири властивості, які посилаються на глобальний об'єкт. Ці властивості є наступними: 'globalThis', 'window', 'self' та 'frames'.

```
const proxifyGlobalObject = (target, realm) =>
  new Proxy(target, {
    get(target, key) {
      return realm.get(key);
    },
    set(target, key, newValue) {
      realm.set(key, newValue);
    },
    has(target, key) {
      return realm.has(key);
    },
  });

const proxifyGlobals = (realm) => {
  const isNotRedefined = (key) => !realm.isRedefined(key);
  const globals = GLOBALS.filter(isNotRedefined);
  for (const key of globals) {
    const global = realm.get(key);
    const value = proxifyGlobalObject(global, realm);
    realm.set(key, { value, writable: false, configurable: false });
  }
};
```

Рисунок 3.14 - Функція proxifyGlobals

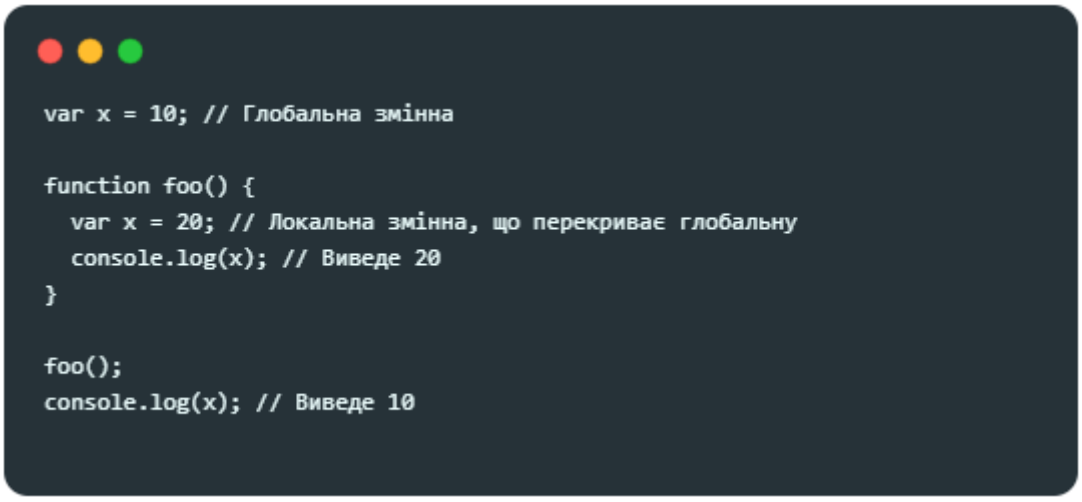
Функція proxifyGlobals обгортає ці глобальні об'єкти в замістники. В результаті, коли код використовує ці властивості, він фактично взаємодіє з замістниками, а не з оригінальними глобальними об'єктами. Це дозволяє контролювати та перехоплювати доступ до цих об'єктів та їх властивостей.

Замістники, створені за допомогою proxifyGlobalObject, мають методи get, set та has, які перехоплюють відповідні операції з оригінальним глобальним об'єктом і звертаються realm для отримання або встановлення значення властивості, включаючи декларативи

3.4.4 Variable shadowing в JavaScript

Для того щоб пояснити функію enchanceScript, потрібн спершу ознайомитися з концепцією перекривання змінних.

Variable shadowing в JavaScript відбувається, коли змінна з тим самим ім'ям, що і змінна в більш високому рівні області видимості, оголошується в більш низькому рівні області видимості. При цьому нова змінна "затіняє" (перекриває) змінну більш високого рівня.



```
var x = 10; // Глобальна змінна

function foo() {
  var x = 20; // Локальна змінна, що перекриває глобальну
  console.log(x); // Виведе 20
}

foo();
console.log(x); // Виведе 10
```

Рисунок 3.15 - Приклад Variable shadowing в JavaScript

У прикладі Рисунок 3.15 ми оголошуємо глобальну змінну x зі значенням 10. Далі, всередині функції foo(), оголошується локальна змінна x зі значенням 20. Ця локальна змінна перекриває глобальну змінну з тим самим ім'ям. При виклику console.log(x) всередині функції foo(), буде виведено 20, оскільки вона має доступ до локальної змінної. Поза функцією foo(), при виклику console.log(x), буде виведено 10, оскільки глобальна змінна не була перекрита локальною змінною всередині функції.

3.4.5 Реалізація декларативів

Функція enhanceScript призначена для реалізаціх декларативів шляхом додавання змінних до початку скрипту

```
const enchanceScript = (realm, userCode) => {
  if (userCode.length === 0) {
    return '';
  }
  let script = `'use strict';\n`;
  for (const name in realm.declaratives) {
    const { backingKey, attributes } = realm.declaratives[name];
    const varType = attributes.writable ? 'let' : 'const';
    const value = `${GLOBAL_THIS}['${backingKey}']`;
    script += `${varType} ${name} = ${value};\n`;
  }
  script += userCode;
  return script;
};
```

Рисунок 3.16 - Функція enchanceScript

Спочатку в функції enchanceScript додається 'use strict' до початку скрипту. Ця директива гарантує використання строгого режиму виконання скрипту.

Циклом for...in обходяться оголошені змінні в контексті (realm). Для кожної змінної створюються наступні елементи:

- backingKey: Ключ, який використовується для доступу до backing field.
- attributes: Дескриптор (атрибути) змінної, наприклад writable.
- varType: Тип змінної (let або const) в залежності від атрибуту writable.
- value: Рядок, який представляє доступ до внутрішнього об'єкта контексту за допомогою глобального об'єкта GLOBAL_THIS та backingKey.
- Додаються рядки ${varType} ${name} = ${value};\n до скрипту. Наприклад, якщо існує змінна з іменем foo та значенням 'bar', то доданий рядок буде const foo = globalThis[‘__foo__’];\n.
- Додається код користувача userCode до скрипту.

- Повертається покращений скрипт, який містить оголошені змінні з контексту та код користувача.

Таким чином, функція enhanceScript доповнює скрипт, що виконується, оголошеними змінними з контексту, що дозволяє їм бути доступними у коді користувача.

3.5 Модуль Vm

Останій модуль vm є надзвичайно простим. Він є єдиною точкою входу для користувача

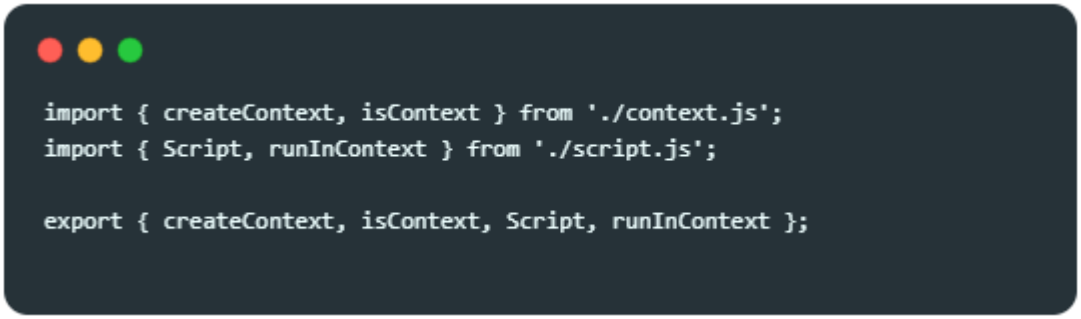


Рисунок 3.17 - Модуль vm

Vm імпортує всі потрібні функції та класи з інших модулів та експортує їх, таким чином створюючи API розробленого проекту.

ВИСНОВОК ДО РОЗДІЛУ 3

У данному розділі було розглянуто чотири модулі (context, realm, script, vm) для створення ізольованих контекстів в браузері.

У модулі realm було використано різноманітні технології та підходи, включаючи патерн Замісник, ідею backing field з Kotlin та концепцію Realm з специфікації ECMAScript. Використання цих технологій допомогло створити ізольований контекст

У модулі context використано WeakMap та FinalizationRegistry. Ці технології дозволяють створювати контексти, які можуть автоматично очищатися після втрати посилань на них. Це сприяє ефективному управлінню ресурсами та допомагає уникнути проблем з витоком пам'яті.

У модулі script було використано об'єкт Proxy та механізм variable shadowing. Використання Proxy дозволило створювати об'єкти, які можуть перехоплювати виклики до контексту. Variable shadowing дозволило створювати локальні змінні, які маскують зовнішні змінні з аналогічними іменами. Це дозволяє “змінювати” незмінні властивості глобального контексту.

Комбінація різних підходів та технологій забезпечує гнучкість та ефективність у створенні ізольованих середовищ для веб-додатків.

ВИСНОВКИ

Цей дипломний проект мав на меті реалізацію модуля для створення ізольованих контексів Javascript в браузерах.

Було проаналізовано декілька технологій які дозволяють отримати ізолюваний контекст, таких як WebAssembly, Web Worker, Iframe.

Під час розробки було використано велике різноманіття підходів, принципів, та методів. Проект є модульним, він розбит на 4 модулі: context, realm, script, vm кожен із яких виконує свою функцію. Основним патерном проектування є Замістник, за домопогою якого реалізована головна функціональність по ізоляції контекстів. Було запозичено багато цікавих ідей з інших сфер, таких як мова програмування Kotlin та специфікація ECMAScript. Для реалізації певних моментів було використані досить продвинуті компонети мови Javascript такі як WeakMap та FinalizationRegistry та досить цікави механізми такі як variable shadowing

Як результат проект надає можливість створювати ізольовані контексти Javascript в браузері, змінювати всі змінні глобального контексту, навіть ті які неможливо змінити в звичайних умовах, додавати нові змінні до контексту, а також автоматично очищає ресурси витрачені на підтримку ізольованого контексту, після того як він стає більше непотрібен користувачу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OpenJS Foundation. About Node.js [Електронний ресурс] / OpenJS Foundation – Режим доступу до ресурсу: https://nodejs.org/en/about.
2. What is V8? [Електронний ресурс] – Режим доступу до ресурсу: https://v8.dev/.
3. Introduction [Електронний ресурс] // ECMAScript – Режим доступу до ресурсу: https://262.ecma-international.org/13.0/#sec-intro.
4. The Global Object [Електронний ресурс] // tc39 – Режим доступу до ресурсу: https://tc39.es/ecma262/multipage/global-object.html#sec-global-object.
5. Window [Електронний ресурс] // MDN – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/Window.
6. Node.js v20.2.0 documentation [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/api/globals.html#globals_global.
7. VM [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/api/vm.html.
8. WebAssembly [Електронний ресурс] // W3C Community Group – Режим доступу до ресурсу: https://webassembly.org/.
9. Web Workers API [Електронний ресурс] // MDN – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.
10. The Inline Frame element [Електронний ресурс] // MDN – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.
11. Realms [Електронний ресурс] // tc39 – Режим доступу до ресурсу: https://tc39.es/ecma262/#sec-code-realms.

12. Proxy / E.Gamma, R. Helm, R. Johnson, J. Vlissides // Design Patterns Elements of Reusable Object-Oriented Software / E.Gamma, R. Helm, R. Johnson, J. Vlissides., 1994. – С. 207–223.
13. Property (JavaScript) [Електронний ресурс] // MDN – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Glossary/Property/JavaScript.
14. Declarative Environment Records [Електронний ресурс] // tc39 – Режим доступу до ресурсу: https://tc39.es/ecma262/#sec-declarative-environment-records.
15. Properties [Електронний ресурс] // Kotlin Foundation – Режим доступу до ресурсу: https://kotlinlang.org/docs/properties.html.
16. WeakMap [Електронний ресурс] // MDN – Режим доступу до ресурсу: https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/WeakMap.
17. FinalizationRegistry [Електронний ресурс] // MDN – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/FinalizationRegistry.
18. Proxy [Електронний ресурс] // MDN – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy
19. Nishimoto M. Understanding ‘Variable Shadowing’ with JavaScript [Електронний ресурс] / Mayumi Nishimoto // Medium – Режим доступу до ресурсу: https://mayuminishimoto.medium.com/understanding-variable-shadowing-with-javascript-58fc108c8f03.

ДОДАТОК 1

Модуль для ізоляції контекстів JavaScript для віртуальних машин що праюють у браузері

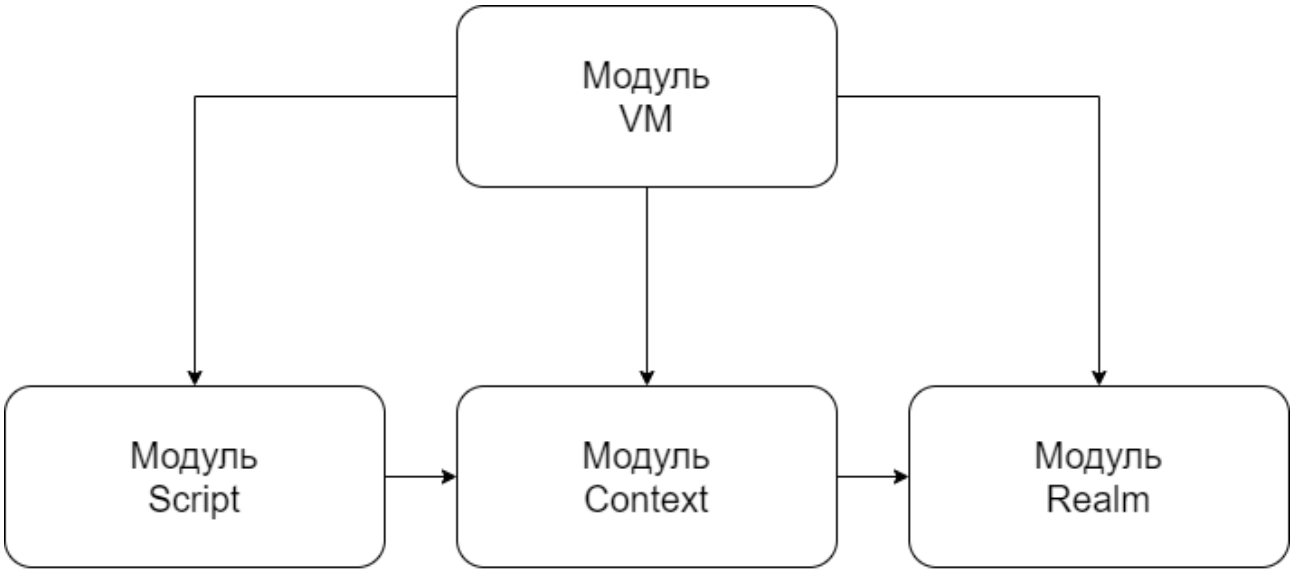
Діаграма модулів

(Структурна схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Перевалов М.О.			Модуль для ізоляції контекстів JavaScript для віртуальних машин що працюють у браузері Діаграма модулів (Структурна схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Шемсидинов Т.Г.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94	
Н. Контр.		Волокита А. М.					
Затвердив							

ДОДАТОК 2

Модуль для ізоляції контекстів JavaScript для віртуальних машин що праюють у браузері

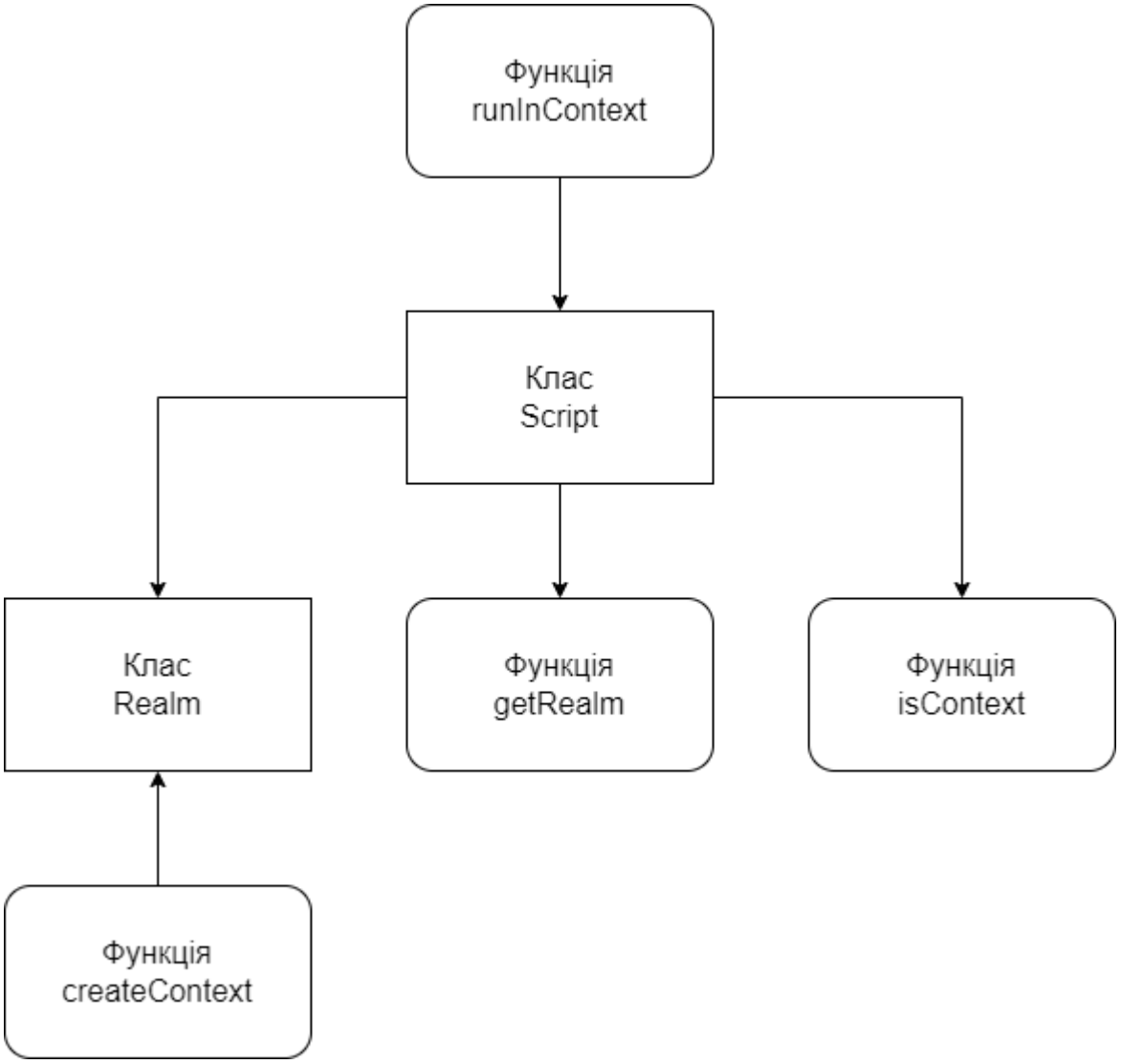
Діаграма компонентів

(Функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Перевалов М. О.			Модуль для ізоляції контекстів JavaScript для віртуальних машин що праюють у браузері Діаграма компонентів (Функціональна схема)	Літ. / Аркуш / Аркушів: 1 / 1
Перевірив		Шемсединов Т.Г.				
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94
Н. Контр.		Волокита А. М.				
Затвердив						

ДОДАТОК 3

Модуль для ізоляції контекстів JavaScript для віртуальних машин що праюють у браузері

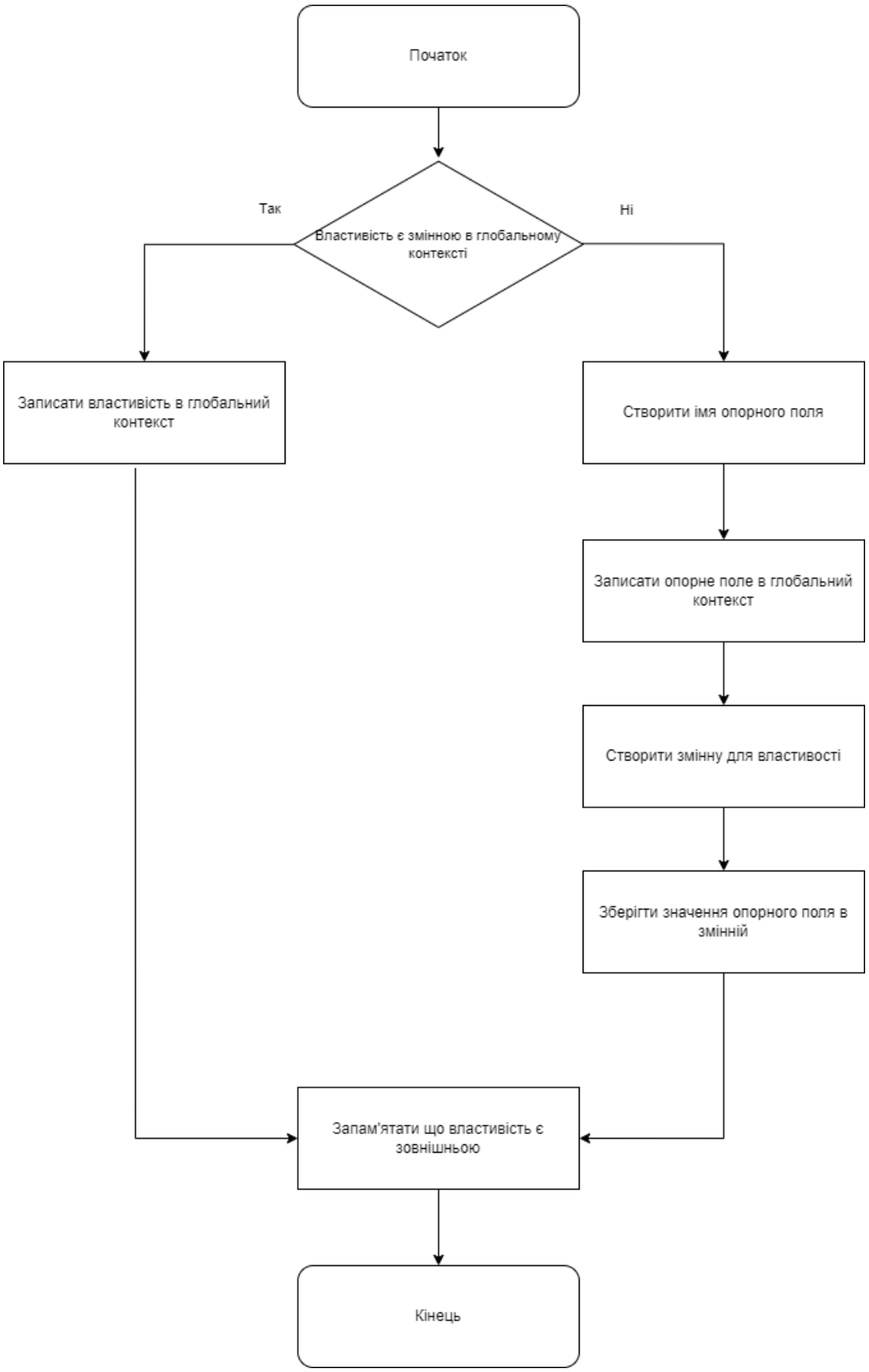
Послідовність дій визначення властивості

(Принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Перевалов М. О.			Модуль для ізоляції контекстів JavaScript для віртуальних машин що праюють у браузері Послідовність дій визначення властивості (Принципова схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Шемсединов Т.Г.					
Н. Контр.		Волокита А. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-94	
Затвердив							

ДОДАТОК 4

Модуль для ізоляції контекстів JavaScript для віртуальних машин що праюють у браузері

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 4

Київ 2023 р

```
// модуль context
```

```
import { Realm } from './realm.js';

const contexts = new WeakMap();
const cleanupRegistry = new FinalizationRegistry(({ document, id }) => {
  document.getElementById(id).remove();
});
let counter = 0;

const copyRoot = (object) => {
  const copy = {};
  Object.assign(copy, object);
  if (Object.isFrozen(object)) {
    Object.freeze(copy);
  }
  if (Object.isSealed(object)) {
    Object.seal(copy);
  }
  if (!Object.isExtensible(object)) {
    Object.preventExtensions(copy);
  }
  return copy;
};

const createIframe = (document, name) => {
  const element = document.createElement('iframe');
  element.style.display = 'none';
  element.id = `iframe-vm:${counter++}`;
  element.name = name ?? element.id;
  element.className = 'iframe-vm';
  document.body.appendChild(element);
  return element;
};

const createContextifiedObject = (context, realm) =>
  new Proxy(context, {
    get(target, key) {
      return Reflect.has(target, key) ? realm.get(key) : undefined;
    },
    set(target, key, value) {
      const defined = Reflect.set(target, key, value);
      if (defined) {
```

```
        const attributes = Reflect.getOwnPropertyDescriptor(target, key);
        realm.set(key, attributes);
      }
      return defined;
    },
    defineProperty(target, key, attributes) {
      const defined = Reflect.defineProperty(target, key, attributes);
      if (defined) {
        realm.set(key, attributes);
      }
      return defined;
    },
  });

const getRealm = (contextObject) => contexts.get(contextObject);

const isContext = (contextObject) => contexts.has(contextObject);

const createContext = (context, options = {}) => {
  const iframe = createIframe(document, options.name);
  const realm = new Realm(iframe.contentWindow, options);
  for (const key of Object.keys(context)) {
    const attributes = Reflect.getOwnPropertyDescriptor(context, key);
    realm.set(key, attributes);
  }
  const contextObject = createContextifiedObject(copyRoot(context), realm);
  contexts.set(contextObject, realm);
  cleanupRegistry.register(contextObject, { document, id: iframe.id });
  return contextObject;
};

export { createContext, getRealm, isContext };

// модуль realm

const GLOBAL_THIS = '___globalThis___';

class Realm {
  constructor(globalObject, options = {}) {
    this.realmOptions = options;
    this.properties = globalObject;
    this.declaratives = {};
    this.externals = [];
```

```
  this.eval = globalObject.eval;
  this.defineHiddenProperty(GLOBAL_THIS, globalObject);
}

set(key, options = {}) {
  const descriptor = {
    value: options.value,
    writable: options.writable ?? true,
    enumerable: options.enumerable ?? true,
    configurable: options.configurable ?? true,
  };
  if (this.isWritableProperty(key)) {
    this.defineProperty(key, descriptor);
  } else {
    this.defineDeclarative(key, descriptor);
  }
}

defineDeclarative(key, attributes) {
  const backingKey = `__${key}__`;
  const value = attributes.value;
  this.declaratives[key] = { backingKey, attributes };
  this.defineHiddenProperty(backingKey, value);
  this.externals.push(key);
}

defineHiddenProperty(key, value) {
  this.defineProperty(key, {
    value,
    writable: false,
    enumerable: false,
    configurable: false,
  });
}

defineProperty(key, attributes) {
  Object.defineProperty(this.properties, key, attributes);
  this.externals.push(key);
}

get(key) {
  if (this.isDeclarative(key)) {
    const { backingKey } = this.declaratives[key];
```

```
      return this.properties[backingKey];
    } else {
      return this.properties[key];
    }
  }

  has(key) {
    return this.isProperty(key) || this.isDeclarative(key);
  }

  hasDeclaratives() {
    return Object.keys(this.declaratives) !== 0;
  }

  isProperty(key) {
    return Reflect.has(this.properties, key);
  }

  isDeclarative(key) {
    return Reflect.has(this.declaratives, key);
  }

  isWritableProperty(key) {
    const attributes = Object.getOwnPropertyDescriptor(this.properties, key);
    return this.isProperty(key) && attributes.writable;
  }

  isRedefined(key) {
    const isExternal = this.externals.includes(key);
    return this.has(key) && isExternal;
  }
}

export { Realm, GLOBAL_THIS };

// модуль script

import { getRealm, isContext } from './context.js';
import { GLOBAL_THIS } from './realm.js';

const GLOBALS = ['globalThis', 'window', 'self', 'frames'];

const enchanceScript = (realm, userCode) => {
```

```
  if (userCode.length === 0) {
    return '';
  }
  let script = `'use strict';\n`;
  for (const name in realm.declaratives) {
    const { backingKey, attributes } = realm.declaratives[name];
    const varType = attributes.writable ? 'let' : 'const';
    const value = `${GLOBAL_THIS}['${backingKey}']`;
    script += `${varType} ${name} = ${value};\n`;
  }
  script += userCode;
  return script;
};

const proxifyGlobalObject = (target, realm) =>
  new Proxy(target, {
    get(target, key) {
      return realm.get(key);
    },
    set(target, key, newValue) {
      realm.set(key, newValue);
    },
    has(target, key) {
      return realm.has(key);
    },
  });

const proxifyGlobals = (realm) => {
  const isNotRedefined = (key) => !realm.isRedefined(key);
  const globals = GLOBALS.filter(isNotRedefined);
  for (const key of globals) {
    const global = realm.get(key);
    const value = proxifyGlobalObject(global, realm);
    realm.set(key, { value, writable: false, configurable: false });
  }
};

class Script {
  constructor(src) {
    this.src = src;
  }

  runInContext(context) {
```

```
    if (!isContext(context)) {
      throw new Error('options.context - is not contextified object');
    }
    const realm = getRealm(context);
    if (realm.hasDeclaratives()) {
      proxifyGlobals(realm);
    }
    const script = enchanceScript(realm, this.src);
    return realm.eval(script);
  }
}

const runInContext = (code, context) => new Script(code).runInContext(context);

export { Script, runInContext };

// модуль vm

import { createContext, isContext } from './context.js';
import { Script, runInContext } from './script.js';

export { createContext, isContext, Script, runInContext };
```