hillel IT school

Lesson 14

План заняття

- Інтерфейси
- Абстракції
- Абстракнтні класи
- Інкапсуляція
- Приватні властивості классу
- Міксини
- try … cetch … finaly

Інтерфейс

Інтерфейс - це сукупність методів та правил взаємодії елементів системи. Іншими словами, інтерфейс визначає, як елементи будуть взаємодіяти між собою.

- Інтерфейс дверей – наявність ручки;
- Інтерфейс автомобіля - наявність керма, педалей, важеля коробки;
- Інтерфейс дискового телефону - трубка + дисковий набір номера.

Коли ви використовуєте ці "об'єкти", ви впевнені, що ви зможете використовувати їх подібним чином. Завдяки тому, що ви знайомі з їхнім інтерфейсом.

"Інтерфейс визначає як ми можемо використовувати об'єкт"

Головна відмінність класу від інтерфейсу - у тому, що клас складається з інтерфейсу та реалізації.

Будь-який клас завжди явно оголошує свій інтерфейс - те, що є при використанні класу ззовні.

Як такого у JS не існує інтерфейсів, тобто ви не можете явно типізувати та описаати сутність.

Абстракція

Абстракція - це спосіб створення простої моделі, яка містить лише важливі властивості з погляду контексту додатку, з більш складної моделі. Іншими словами - це спосіб приховати деталі реалізації та показати користувачам лише функціональність. Абстракція ігнорує нерелевантні деталі та показує лише необхідні. Важливо пам'ятати, що ми можемо створити екземпляр абстрактного класу.

Все програмне забезпечення – це абстракція, що приховує всю важку роботу та бездумні деталі.

Інтерфейс & Абстрактний клас

У JS не існує абстрактних класів.

Абстрактним називається такий клас, екземпляр якого не можна створити сам собою – він служить основою для інших класів.

Наприклад, геометрична фігура може представляти коло, квадрат, трикутник, але як такої геометричної фігури самої по собі не існує.

У абстрактних класах можуть існувати абстрактні методи та влаастивості.

Typescript example:

```
abstract class Figure {
    abstract x: number;
    abstract y: number;
    abstract getArea(): void;
}
```

Інтерфейс & Абстрактний клас

hillel IT school

1. Інтерфейс описує лише поведінку. Він не має стану. А в абстрактного класу стан є: він описує і те, й інше.
2. Абстрактний клас пов'язує між собою та поєднує класи, що мають дуже близький зв'язок. У той самий час, той самий інтерфейс можуть реалізувати класи, які взагалі немає нічого спільного.
3. Класи можуть реалізовувати скільки завгодно інтерфейсів, але успадковуватися можна лише від одного класу.

Інтерфейс

У об’єктно-орієнтованому програмуванні, властивості та методи розділені на дві групи:

- Внутрішній інтерфейс – методи та властивості, доступні в інших методах класу, але не ззовні.
- Зовнішній інтерфейс – методи та властивості, доступні також ззовні класу.

У JavaScript існує два типи полів об’єктів (властивостей та методів):

- Публічні: доступні з будь-якого місця. Вони складають зовнішній інтерфейс. Досі ми використовували лише публічні властивості та методи.
- Приватні: доступний тільки зсередини класу. Вони для внутрішнього інтерфейсу.

Захищені поля не реалізовані в JavaScript на рівні мови, але на практиці вони дуже зручні, тому вони емулюються.

Інкапсуляція

Інкапсуляція – це принцип, згідно з яким будь-який клас або будь-яка частина системи повинна розглядатися як «чорна скринька»: користувач класу або підсистеми повинен бачити лише інтерфейс (тобто список декларованих властивостей та методів) і не вникати у внутрішню реалізацію.

Іншими словами це означає здатність об'єкта «вирішувати», яку інформацію він розкриватиме для зовнішнього світу, а яку ні. Реалізується цей принцип через громадські та закриті властивості та методи.

У JS усі властивості об'єктів та методи за замовчуванням є публічними. «Публічне» означає можливість доступу до властивості/методу об'єкта ззовні його тіла

Інкапсуляція корисна у випадках, коли нам потрібні певні властивості або методи виключно для внутрішніх процесів об'єкта, і ми не хочемо розкривати їх зовні.

З точки зору ООП, відокремлення внутрішнього інтерфейсу від зовнішнього називається інкапсуляція.

Емуляція

Захищені властивості зазвичай починають з підкресленням _. Це не синтаксис на рівні мови, а лише відома домовленість між програмістами, що такі властивості та методи не повинні бути доступними ззовні.

Щоб контролювати змінні використовуйте get/set

Захищені поля успадковуються!

Приватна властивість

Приватні властивості і методи повинні починатися з #. Вони доступні лише з класу.

На рівні мови, # – це особливий знак того, що поле є приватним. Ми не можемо отримати доступ до нього ззовні або з наслідуваних класів.

Приватні поля не конфліктують з публічним. Ми можемо мати як приватне #salery та і публічне salery поле одночасно.

На відміну від захищених, приватні поля забезпечуються самою мовою.

Розширення вбудованих класів

Вбудовані класи, такі як Array, Map та інші, також розширюються.

Пам’ятайте!

Вбудовані класи не успадковують статику один від одного.

Перевірка класу: "instanceof"

Оператор instanceof дозволяє перевірити, чи належить об’єкт до певного класу. Він також враховує наслідування.

Алгоритм операції obj instanceof Class працює приблизно наступним чином:

1. Якщо є статичний метод Symbol.hasInstance, тоді він просто викликається: Class[Symbol.hasInstance](obj). Він повинен повернути true або false, ось і все. Ось як ми можемо задати поведінку instanceof.
2. Більшість класів не мають Symbol.hasInstance. У цьому випадку використовується стандартна логіка: obj instanceOf Class перевіряє чи Class.prototype дорівнює одному з прототипів у ланцюжку прототипів obj.

.toString для визначення типу

Можна зробити метод toString набагато потужнішим. Ми можемо використовувати його як розширений typeof і альтернативу instanceof.

Нам потрібно викликати метод toString() зновим контекстом наприклад this=arr. Всередені алгоритм toString перевіряє this і повертає відповідний результат.

Також поведінку методу об’єкта toString можна налаштувати за допомогою спеціальної властивості Symbol.toStringTag. Для більшості специфічних для середовища об’єктів така властивість є, наприклад window, document, XMLHttpRequest ...

Міксини

В JavaScript ми можемо успадкуватися лише від одного об’єкта. Може бути лише один [[Prototype]] для об’єкта. І клас може розширювати лише один інший клас.

Як визначено у Вікіпедії, mixin – це клас, що містить методи, які можуть бути використані іншими класами без необхідності успадкуватися від нього.

Іншими словами, міксин забезпечує методи, які реалізують певну поведінку, але ми не використовуємо його самостійно, ми використовуємо його, щоб додати цю поведінку до інших класів.

Try … catch

Якщо виникають помилки, то скрипти, зазвичай, “помирають” (раптово припиняють роботу) та виводять інформацію про помилку в консоль. Але існує синтаксична конструкція try...catch, що дозволяє нам “перехоплювати” помилки, що дає змогу скриптам виконати потрібні дії, а не раптово припинити роботу.

Конструкція try...catch містить два головних блоки: try, а потім catch. Це працює наступним чином:

1. В першу чергу виконується код в блоці try {...}.
2. Якщо не виникає помилок, то блок catch (err) ігнорується: виконання досягає кінця блоку try та продовжується поза блоком catch.
3. Якщо виникає помилка, тоді виконання в try припиняється і виконання коду продовжується з початку блоку catch (err). Змінна err (можна обрати будь-яке ім’я) буде містити об’єкт помилки з додатковою інформацією.

Щоб блок try...catch спрацював, код повинен запускатися. Іншими словами, це повинен бути валідний JavaScript.

try...catch працює синхронно!!!

Try … catch

Коли виникає помилка, JavaScript генерує об’єкт, що містить інформацію про неї. Потім цей об’єкт передається як аргумент в catch

Для всіх вбудованих помилок об’єкт помилки має дві головні властивості:

- name - Назва помилки. Наприклад, для невизначеної змінної назва буде "ReferenceError".
- message - Текстове повідомлення з додатковою інформацією про помилку.

Існують інші властивості, що доступні в більшості оточень. Одна з найуживаніших та часто підтримується:

- stack - Поточний стек викликів: рядок з інформацією про послідовність вкладених викликів, що призвели до помилки. Використовується для налагодження.

Пам’ятайте

Блок catch не обов’язково повинен перехоплювати інформацію про об’єкт помилки.

Try … catch … finaly

Інструкція finally дозволяє виконувати код після try та catch незалежно від результату. Блок finally використовується, якщо ми почали виконувати якусь роботу і хочемо завершити її в будь-якому разі.

Змінні визначені всередині try...catch...finally є локальними. Частина finally виконається в будь-якому разі при виході з try...catch. Навіть якщо явно викликати return.

Конструкція try...finally може не мати catch частини, що також може стати у пригоді. Така конфігурація може бути використана, коли ми не хочемо перехоплювати помилку, але потрібно завершити розпочаті задачі.

Об’єкт Error

JavaScript має вбудований об’єкт error, що надає інформацію про помилку при її виникненні.

Об’єкт error надає дві корисних властивості: ім’я та повідомлення.

Типи:

1. Error - Конструктор Error створює об'єкт помилки. Екземпляри об'єкта Error викидаються при помилках під час виконання. Приймає message помилки.
2. EvalError - Відбулась помилка в eval() функції;
3. RangeError - Відбулось число "out of range" (поза діапазоном);
4. ReferenceError - Відбулось неприпустиме посилання;
5. SyntaxError - Відбулась синтаксична помилка;
6. TypeError - Відбулась помилка типу;
7. URIError - Відбулась помилка в encodeURI();

throw

Оператор throw використовується для викидання помилки.

Рушії дозволяє використовувати будь-які значення як об’єкти помилки. Це може бути навіть примітивне значення, як число чи рядок, але краще використовувати об’єкти, що мають властивості name та message (для сумісності з вбудованим типом помилок).

catch

Блок catch повинен оброблювати тільки відомі помилки та повторно генерувати всі інші типи помилок.

Розгляньмо підхід “повторного викидання” покроково:

1. Конструкція catch перехоплює всі помилки.
2. В блоці catch (err) {...} ми аналізуємо об’єкт помилки err.
3. Якщо ми не знаємо як правильно обробити помилку, ми робимо throw err.

Зазвичай, тип помилки можна перевірити за допомогою оператора instanceof

Глобальний catch

Специфікація не згадує таку можливість, але оточення, зазвичай, надають таку функцію для зручності. Наприклад, Node.js дозволяє викликати process.on("uncaughtException") для цього. В браузері можна присвоїти функцію спеціальній властивості window.onerror, що виконається, коли виникне помилка.

Синтаксис:

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

- message - Повідомлення помилки.
- url - URL скрипту, де трапилась помилка.
- line, col - Номер рядку та колонки, де трапилась помилка.
- error - Об’єкт помилки.

Розширення Error

JavaScript дозволяє використовувати throw з будь-яким аргументом, тому технічно наші спеціальні класи помилок не повинні успадковуватись від Error. Але якщо ми успадкуємо, то стає можливим використовувати obj instanceof Error для ідентифікації об’єктів помилки. Тому краще успадкувати від нього.

Дякую за увагу