INTERVIEW

0 Підходи та концепції

0.1) ООП

Інкапсуляція (Encapsulation):

- **Опис:** Інкапсуляція полягає в упаковці даних та методів, які обробляють ці дані, в єдиний об'єкт. Інші частини програми не повинні знати, як саме внутрішній об'єкт виконує свої функції.
- **Переваги:** Забезпечує захист даних та дозволяє контролювати доступ до них. Сприяє модульності та полегшує зміни внутрішнього реалізації об'єкта без впливу на зовнішні частини програми.

Спадкування (Inheritance):

- **Опис:** Спадкування дозволяє створювати новий клас, використовуючи властивості та методи іншого класу. Новий клас називається підкласом, а клас, від якого він успадковує властивості, - батьківським класом або суперкласом.
- **Переваги:** Забезпечує повторне використання коду, спрощує розширення функціональності і полегшує утримання коду.

Поліморфізм (Polymorphism):

- **Опис:** Поліморфізм дозволяє об'єктам одного класу використовувати методи або властивості інших класів. Це може виражатися у використанні методів з однаковою назвою, але з різною реалізацією, або у використанні методів або властивостей через загальний інтерфейс.
- **Переваги:** Забезпечує гнучкість та універсальність коду. Дозволяє працювати з об'єктами без необхідності знати їхню конкретну реалізацію.

Абстракція (Abstraction):

- **Опис:** Абстракція полягає в прихованні непотрібних деталей та використанні лише необхідних для досягнення конкретної мети. Вона дозволяє визначати та використовувати лише ті характеристики об'єкта, які важливі для даного контексту.
- **Переваги:** Забезпечує спрощення та узагальнення використання об'єктів, полегшує розуміння та взаємодію між компонентами системи.

0.2)SOLID

SOLID — це набір принципів об’єктно-орієнтованого програмування, які представив Роберт Мартін (дядько Боб) у 1995 році. Їхня ідея в тому, що треба уникати залежностей між компонентами коду. Якщо є велика кількість залежностей, такий код важко підтримувати (спагеті-код). Його основні проблеми:

- жорсткість (Rigidity): кожна зміна викликає багато інших змін;
- крихкість (Fragility): зміни в одній частині ламають роботу інших частин;
- нерухомість (Immobility): код не можна повторно використати за межами його контексту.

1) Принцип єдиного обов’язку

Кожен клас повинен виконувати лише один обов’язок. Це не означає, що в нього має бути тільки один метод. Це означає, що всі методи класу мають бути сфокусовані на виконання одного спільного завдання. Якщо є методи, які не відповідають меті існування класу, їх треба винести за його межі

2) Принцип відкритості/закритості (Open/Close Principle)

Класи мають бути відкриті до розширення, але закриті для змін. Якщо є клас, функціонал якого передбачає чимало розгалужень або багато послідовних кроків, і є великий потенціал, що їх кількість буде збільшуватись, то потрібно спроєктувати клас таким чином, щоб нові розгалуження або кроки не призводили до його модифікації.
Напевно, кожен з нас бачив нескінченні ланцюжки if then else або switch. Щойно додається чергова умова, ми пишемо черговий if then else, змінюючи при цьому сам клас. Або клас виконує процес з багатьма послідовними кроками — і кожен новий крок призводить до його зміни. А це порушує Open/Close Principle.

3) Принцип підстановки Лісков (Liskov Substitution Principle)

Якщо об’єкт базового класу замінити об’єктом його похідного класу, то програма має продовжувати працювати коректно.
Ящо ми перевизначаємо похідні методи від батьківського класу, то нова поведінка не має суперечити поведінці базового класу. Як приклад порушення цього принципу розглянемо такий код:

```
class BaseClass {
  public add(a: number, b: number): number {
    return a + b;
  }
}

class DerivedClass extends BaseClass {
  public add(a: number, b: number): number {
    throw new Error('This operation is not supported');
  }
}
```

Можливий також випадок, що батьківський метод буде суперечити логіці класів-нащадків. Розглянемо наступну ієрархію транспортних засобів:

```
class Vehicle {
  accelerate() {
    // implementation
  }

  slowDown() {
    // implementation
  }

  turn(angle: number) {
    // implementation
  }
}

class Car extends Vehicle {
}

class Bus extends Vehicle {
}
```

Все працює до того моменту, допоки ми не додамо новий клас Поїзд.

```
class Train extends Vehicle {
  turn(angle: number) {
    // is that possible?
  }
}
```

Оскільки поїзд не може змінювати довільно напрямок свого руху, то turn батьківського класу буде порушувати принцип підстановки Лісков.
Щоб виправити цю ситуацію, ми можемо додати два батьківські класи: FreeDirectionalVihicle — який буде дозволяти довільний напрямок руху і BidirectionalVehicle — рух тільки вперед і назад. Тепер всі класи будуть наслідувати тільки ті методи, які можуть забезпечити.

```
class FreeDirectionalVehicle extends Vehicle {
  turn(angle: number) {
    // implementation
  }
}

class BidirectionalVehicle extends Vehicle {
}
```

4) Принцип розділення інтерфейсу (Interface Segregation Principle)

Краще, коли є багато спеціалізованих інтерфейсів, ніж один загальний. Маючи один загальний інтерфейс, є ризик потрапити в ситуацію, коли похідний клас логічно не зможе успадкувати якийсь метод. Розглянемо приклад:

```
interface IDataSource {
  connect(): Promise<boolean>;
  read(): Promise<string>;
}

class DbSource implements IDataSource {
  connect(): Promise<boolean> {
    // implementation
  }

  read(): Promise<string> {
    // implementation
  }
}

class FileSource implements IDataSource {
  connect(): Promise<boolean> {
    // implementation
  }

  read(): Promise<string> {
    // implementation
  }
}
```

Оскільки з файлу ми читаємо локально, то метод Connect зайвий. Розділимо загальний інтерфейс IDataSource:

```
interface IDataSource {
  read(): Promise<string>;
}

interface IRemoteDataSource extends IDataSource {
  connect(): Promise<boolean>;
}

class DbSource implements IRemoteDataSource {
}

class FileSource implements IDataSource {
}
```

5) Принцип інверсії залежностей (Dependency Inversion Principle)

Він складається з двох тверджень:

- високорівневі модулі не повинні залежати від низькорівневих. І ті, і ті мають залежати від абстракцій;
- абстракції не мають залежати від деталей реалізації. Деталі реалізації повинні залежати від абстракцій.

Розберемо код, який порушує ці твердження:

Висновки

Це природно, що з розвитком системи зростає її складність. Важливо завжди тримати цю складність під контролем. Інакше може виникнути ситуація, коли додавання нових фіч, навіть не дуже складних, обійдеться занадто дорого. Деякі проблеми повторюються особливо часто. Щоб їх уникати, було розроблено принципи проєктування. Якщо будемо їх дотримуватися, то не допустимо лавиноподібного підвищення складності системи. Найпростішими такими принципами є SOLID.

І на останок: Роберта Мартіна вважають Rock Star у розробці програмного забезпечення. На його книгах вже виросло декілька поколінь суперуспішних програмістів. «Clean Code» і «Clean Coder» — дві його книги про те, як писати якісний код і відповідати найвищим стандартам в індустрії

0.3)KISS

KISS (Keep It Simple, Stupid):

Опис: Принцип KISS вказує на те, що розв'язання завдань повинно бути якнайпростіше. Програмісти повинні уникати надмірної складності та використовувати прості та зрозумілі рішення.

Застосування:

- При проектуванні та написанні коду обирайте прості та зрозумілі конструкції.
- Уникайте надмірного використання складних або запутаних патернів.

1)
SOLID це концепція приумана в 1995 році , для полегшення роботи з великими проектами , вона має 5 принципів які ти назвав нижче ; Його важливість це поегшення роботи з великими проектами ;

2)
принцип єдиної відповідальності це принцип коли функція чи то клас мають відповідати тільки за одну дію , тобто клас User який відповідає за данні користувача та їх обробку не повинен обробляти до прикаду данні по типу : дозволи до фільмів , тому ще це вже дві відповідальності і краще створити клас Access який уде цим займатися;

3)
Мається наувазі що наш клас відкритий що розширення , але не зміни вже існуючого , тому що це може призвести до поганих наслідків таких як переписування тесту і рефектоинг всього коду , тому краще проєктувати клас так щоб було легше його змінювати;

4)
принцип підстановки барбари лісков це принцип , "якщо ми підставимо дочірній метод в батьківський клас він буде працювати коректно " , хоча він працює в обидві сторони і інколи проблема саме в батьківському класі а не в дочірньому і тоді вже потрібно переписувати батьківський клас коректно;

5)
Цей принцип означає що краще мати багато малих інтерфейсів ніж один великий, тому що є ризик що ми будемо передавати не потрібні поля і методи з інтерфейсу , а це користувачу не потрібно .

6)
Цей принцип полягає в тому що все залежить від абстракцій ; Що власне повязано з 4 концепцією ООП , а саме абстракціями . Я його не до кінця зрозумів , буду радий якщо поясниш краще;

7)
KISS , напряму нам каже що не треьа ускладнювати код , де можна зробити просто та зрозуміло - робимо , не варто штучно ускладнювати.

8)
Тому що DRY тісно повязаний з усіма вище перерахованими концепціями , не варто ускладнювати і дублювати код десятки разів , в сотні раз легше просто винести цей код в окремий клас , або функцію і дальше просто імпортувати її там де потрібно . Не повторюй себе .

1 Браузер

1.1) Відмальовування сторінки;

1) Браузер ініціює запит до сервера, щоб отримати HTML-код та інші ресурси, такі як стилі, скрипти та зображення.

2) Браузер обробляє HTML-код для створення DOM (Document Object Model) та обробляє CSS для створення CSSOM (CSS Object Model). : Якщо на сторінці присутній JavaScript, браузер виконує його код. Виконання JavaScript може змінювати DOM і CSSOM. Наприклад, скрипти можуть додавати нові елементи до DOM, змінювати стилі, взаємодіяти з користувачем та виконувати інші дії для забезпечення динамічності сторінки.

3) Об'єднання DOM і CSSOM відбувається для створення Render Tree, а не Render Dom. Render Tree включає в себе тільки ті елементи, які потрібно відобразити на екрані. З цього Render Tree створюється Layout та Paint, а потім відбувається відображення на екрані.

4) Після відображення на екрані браузер готовий обробляти взаємодію з користувачем, виконувати події та відправляти додаткові запити на сервер при необхідності.

Layout - це процес визначення місцезнаходження та розмірів всіх елементів на екрані. Після створення Render Tree, браузер розраховує, де кожен елемент буде розташований та які розміри він матиме. Це визначає геометричні аспекти кожного елемента, такі як положення, ширина, висота, відступи тощо.

Після Layout відбувається **Paint**, або малювання. На цьому етапі браузер визначає, як саме буде виглядати кожен елемент на екрані враховуючи його стилі та зображення. Отримана інформація використовується для зображення кожного пікселя на екрані. Процес Paint включає у себе застосування кольорів, текстур, тіней, рамок та інших візуальних аспектів елементів.

Загалом, Layout та Paint - це два етапи у процесі відображення веб-сторінки, які визначають геометричні та візуальні аспекти кожного елемента.

2 HTML

Семантичний HTML:

- **Плюси:**
- **Зрозумілість Коду:** Використання семантичних тегів (наприклад, **<header>**, **<nav>**, **<article>**, **<footer>**) полегшує зрозуміння структури документа, як для розробників, так і для інших інструментів (пошукових систем, екранних читачів і т. д.).
- **Підвищення SEO:** Пошукові системи можуть легше і краще розуміти та індексувати контент за допомогою семантичних тегів, що може поліпшити позиції в результатах пошуку.
- **Мінуси:**
- **Збільшення Розміру Коду:** Деякі семантичні теги можуть призводити до збільшення об'єму коду, але це зазвичай невелика ціна за покращену читабельність та зрозумілість.

3 CSS

3.1) Pseudo-classes та Pseudo-elements:

- Використання **:hover**, **:active**, **:nth-child**, тощо.
- Використання **::before** та **::after** для додавання контенту до елементів.

3.2)

Flexbox і Grid - це дві різні техніки розміщення та вирівнювання елементів у CSS, і вони мають різні використання та сфери застосування.

Вибір Між Flexbox та Grid:

- **Flexbox:**
- Коли потрібен простий та гнучкий одновимірний макет.
- Для вирівнювання елементів в одному рядку чи колонці.
- **Grid:**
- Коли потрібен більший контроль над розташуванням у двох вимірах.
- Для створення складних макетів, що включають в себе рядки та колонки.
- **Комбінація:**
- Часто використовується комбінація Flexbox і Grid для досягнення оптимального контролю над макетом та вирівнюванням.

3.3)Селектори :

1)Елементовий :

P , div, li ...

2)id селектор :

#about

3)класовий:

.price

4)груповий :

h1 , h2, h3

5) потомковий :

div .name{}

3.4)POSITION:

1) **static (Статичне):**
   - Елемент розташовується відносно свого нормального положення в потоці документа.
   - Властивості **top**, **right**, **bottom**, **left** не застосовуються.

2) **relative (Відносне):**
   - Елемент розташовується відносно свого нормального положення, але може зміщуватися за допомогою властивостей **top**, **right**, **bottom**, **left**.

3) **absolute (Абсолютне):**
   - Елемент виймається з потоку документа і розташовується відносно ближнього непозиціонованого предка або контейнера.
   - Якщо немає відповідних непозиціонованих предків, елемент розташовується відносно самого ближнього об'єкта потоку документа (зазвичай **<body>**).

4) **fixed (Фіксоване):**
   - Елемент виймається з потоку документа і розташовується відносно вікна перегляду.
   - Залишається на місці, навіть при прокручуванні сторінки.

5) **sticky (Клейке):**
   - Елемент визначається відносно ближнього контейнера прокрутки, або вікна, коли відстань до верхнього краю елемента стає меншою за висоту контейнера прокрутки.

4 SCSS

4.1) **Міксіни (Mixins):**

Оголошення Міксінів:

```
@mixin border-radius($radius
  -webkit border-radius  $radius
  -moz border-radius  $radius
  -ms border-radius  $radius
  border-radius  $radius
```

-

.box { @include border-radius(10px); }

4.2) Також позволяє добавляти умови та цикли

4.3) Створення вкладень, змінних і функцій

5 JavaScript

5.1) Ключові слова

Особливості	var	const	let
Область видимості	Знаходиться в межах функції	Знаходиться в межах блоку	Знаходиться в межах блоку
Переприсвоєння	Може бути переприсвоєно	Може бути переприсвоєно	Не може бути переприсвоєно
Підняття (Hoisting)	Піднімається вгору в області видимості	Не піднімається вгору в області видимості	Не піднімається вгору в області видимості
Временна зона загибелі (Temporal Dead Zone)	Відсутня	Присутня	Присутня
Приклад	**var x = 10;**	**Const x = 10;**	**let x = 10;**

Короткий пояснення:

- **Область видимості:**
- **var** має область видимості функції, тобто змінна видима лише всередині функції, де вона оголошена.
- **let** та **const** мають область видимості блоку, тобто видимі лише всередині блоку (розділеного фігурними дужками), де вони оголошені.
- **Переприсвоєння:**
- **var** і **let** можуть бути переприсвоєні новим значенням.
- **const** не може бути переприсвоєно після того, як йому було присвоєно значення. Проте, це не робить об'єкти або масиви, які в нього присвоєні, незмінними.
- **Підняття (Hoisting):**
- Змінні, оголошені за допомогою **var**, піднімаються вгору в області видимості та можуть бути використані до їхнього оголошення.
- Змінні, оголошені за допомогою **let** та **const**, також піднімаються вгору, але не ініціалізуються, і спроба отримати доступ до них перед оголошенням призведе до помилки.
- **Временна зона загибелі (Temporal Dead Zone):**
- **var** не є чутливим до цієї концепції.

- Якщо використовуються **let** або **const** перед їхнім оголошенням у тому ж блоку, в якому вони оголошені, виникне помилка.

5.2) Null та undefiend

null та **undefined** - це два спеціальні значення в JavaScript, які вказують на відсутність значення, але вони використовуються в різних контекстах і мають свої відмінності

Загалом, **undefined** зазвичай виникає при використанні неініціалізованих змінних або властивостей, тоді як **null** частіше використовується для явного вказівки на відсутність значення

5.3)Методи масивів;

Методи	Опис	Приклад
concat	Об'єднує два або більше масивів	arr1.concat(arr2)
filter	Створює новий масив з елементів, які	arr.filter(element => element > 5)
map	Створює новий масив на основі результату	arr.map(element => element * 2)
forEach	Викликає задану функцію для кожного елемента	arr.forEach(element => console.log(element))

indexOf	Повертає перший індекс елемента в масиві або -1, якщо елемент не знайдено	const fruits = ['apple', 'orange', 'banana', 'grape'];
		
		const indexOfBanana = fruits.indexOf('banana');
		console.log(indexOfBanana); // Виведе: 2

push	Додає один або кілька елементів в кінець масиву	arr.push(4)
pop	Видаляє останній елемент з кінця масиву	arr.pop()
shift	Видаляє перший елемент з початку масиву	arr.shift()
unshift	Додає один або кілька елементів в початок масиву	arr.unshift(0, 1)

slice	Повертає частину масиву як новий масив	arr.slice(1, 3

splice	Змінює вміст масиву, додаючи / видаляючи	const colors = ['red', 'blue', 'green', 'yellow'];
		
		// Видаляє 1 елемент починаючи з індексу 0 та вставляє 'purple' та 'orange' colors.splice(0, 1, 'purple', 'orange');
		
		console.log(colors); // Виведе: ['purple', 'orange', 'blue', 'green', 'yellow']

reverse	Змінює порядок елементів в масиві на зворотній	arr.reverse()
sort	Сортує елементи масиву	arr.sort()
includes	Перевіряє, чи масив містить певний елемент	arr.includes(3)
every	Перевіряє, чи всі елементи задовольняють задану умову	arr.every(element => element > 0)
some	Перевіряє, чи хоча б один елемент задовольняє задану умову	arr.some(element => element > 5)

5.4)for in | for of

for...in та **for...of** - це два різні способи ітерації в JavaScript, і вони мають

For in - по ключам; (для обєктів)

```
const obj = { a: 1, b: 2, c: 3 };
for (let key in obj) {
  console.log(key, obj[key]);
}
```

For of - по значенням; (для масивів)

```
const arr = [1, 2, 3];

for (let value of arr) {
```

console.log(value);

5.5) (Object.keys(), Object.values())

const obj = { a: 1, b: 2, c: 3 };

const values = Object.values(obj);

const keys = Object.keys(obj);

console.log(keys); // Виведе: ['a', 'b', 'c']

console.log(values); // виведе: [1, 2, 3]

5.6)try catch

Конструкцію **try...catch...finally** можна використовувати як для синхронного, так і для асинхронного коду в JavaScript. Однак є деякі важливі відмінності у їхньому використанні, залежно від того, чи ви працюєте зі синхронним або асинхронним кодом.

Якщо ви маєте справу з асинхронним кодом, важливо розуміти, що блок **catch** не зможе перехопити помилки, які виникають у віддаленому асинхронному коді. В таких випадках краще використовувати **try...catch** прямо всередині асинхронної функції або обробників промісів.

Отже, конструкцію **try...catch...finally** можна використовувати в обох випадках, але у випадку асинхронного коду важливо дотримуватися особливостей асинхронного виклику та обробки помилок.

5.7) async / await

async та **await** вводяться в ECMAScript 2017 (ES8) і надають зручний спосіб працювати з **Promise**. Ключове слово **async** визначає асинхронну функцію, яка завжди повертає **Promise**. Ключове слово **await** використовується всередині асинхронної функції і очікує завершення операції, яка повертає **Promise**.

Приклад використання async/await:

```
async function fetchData() {
  const result = await fetch("https://jsonplaceholder.typicode.com/users")
    .then((response) => response.json())
    .catch((error) => reject(error));
  return result;
}

fetchData().then((response) => console.log(response));
```

5.8)Promise

Проміси використовуються для ефективної обробки асинхронних операцій в JavaScript та React. Ось деякі практичні приклади використання промісів:

```
async function fetchData() {
  return new Promise(function (resolve, reject) {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((response) => response.json())
      .then((data) => resolve(data))
      .catch((error) => reject(error));
  });
}

fetchData().then((data) => console.log(data));
```

5.9)LocalStorage | localSession

1)LocalStorage

- **Тривалість зберігання:** Дані в **localStorage** зберігаються настійно, і вони залишаються в доступі після закриття браузера та перезавантаження комп'ютера.
- **Обсяг даних:** Зазвичай більший обсяг даних можна зберігати в **localStorage**, порівняно із **sessionStorage**.
- **Використання:**

```
// Записати дані в localStorage під ключем "key"
localStorage.setItem("key", "value");
localStorage.setItem(1, 2);
// Отримати дані з localStorage за ключем "key"
const value = localStorage.getItem("key");
const number = localStorage.getItem(1);
console.log(value); // Виведе: "value"
console.log(number); // 2 => "value"
```

2)Session Storage:

- **Тривалість зберігання:** Дані в **sessionStorage** зберігаються лише протягом сесії браузера. Якщо ви закриєте вкладку або видалите сесію, дані будуть видалені.

- **Обсяг даних:** Зазвичай менший обсяг даних можна зберігати в **sessionStorage**, порівняно із **localStorage**.
- **Використання:**

```
sessionStorage.setItem("key", "value");

// Отримати дані з sessionStorage
const value = sessionStorage.getItem("key");
console.log(value); // Виведе: "value"
```

5.10)Замикання

Замикання (closure) - це концепція в програмуванні, яка виникає, коли функція має доступ до змінних з області видимості, яка вже не існує. У JavaScript це може виникнути, коли внутрішня функція знаходиться в області видимості зовнішньої функції

Основна ідея замикання полягає в тому, що функція, визначена всередині іншої функції (зовнішньої функції), має доступ до змінних цієї зовнішньої функції, навіть після того, як зовнішня функція завершила свою роботу.

У цьому прикладі innerFunction - це замикання, оскільки вона має доступ до outerVariable, яка визначена у зовнішній функції outerFunction. Замикання може бути потужним механізмом в JavaScript, і вони широко використовуються, наприклад, для створення приватних змінних, об'єктів, а також для реалізації деяких шаблонів програмування.

```
function outerFunction() {
  let outerVariable = "I am from the outer function";
  function innerFunction() {
    console.log(outerVariable); // innerFunction має доступ до outerVariable
  }
  return innerFunction;
}
const closureExample = outerFunction();closureExample(); // Виведе: "I am from the outer function"
```

6 TYPESCRIPT

6.1)Type vs Interface

Загалом, обирайте "interface", коли вам потрібно створити контракти та декларації для об'єктів, і "type", коли вам потрібно створити аліаси для типів або використовувати об'єднання та перетини. Однак ці відмінності стають менш очевидними, і обидва можна використовувати в багатьох схожих ситуаціях.

Особливість	**type**	**interface**
Визначення об'єднань типів	Може об'єднувати (union) типи	Може також об'єднувати типи, але використовує **extends** для розширення
Визначення перетину типів	Може визначати перетин (intersection) типів	Може визначати перетин (intersection) типів
Розширення об'єктів	Може розширювати (extend) об'єктні типи	Може розширювати (extend) об'єктні типи
Робота з примітивними типами	Підтримує примітивні типи	Підтримує примітивні та об'єктні типи
Доступ до ключових слов	Не може використовувати **extends** або **implements**	Може використовувати **extends** та **implements**

6.2)Generic <T>

Generic (генерик) - це механізм у TypeScript, який дозволяє створювати компоненти (функції, класи, інтерфейси), які працюють з різними типами даних, забезпечуючи при цьому безпеку типів. Використовуючи генерики, ви можете писати більш універсальний та повторно використовуваний код.

6.3)Абстрактні класи та Interface

Абстрактні класи в TypeScript є способом надати базовий функціонал для підрозділених класів, одночасно вимагаючи від них реалізації певних методів чи властивостей. Це дозволяє визначити загальний інтерфейс та функціональність для групи класів, забезпечуючи при цьому певний рівень структурної єдності.

```typescript
abstract class Person {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    abstract sayHello(): void;
}

class Student extends Person {
    sayHello(): void {
        console.log(`Hello, my name is ${this.name}.`);
    }
}
const student = new Student("Ivan");
student.sayHello();
```

Категорія	Абстрактні класи	Інтерфейси
Реалізація коду	+ Може містити реалізацію методів	- Може містити тільки сигнатури методів
Поля класу	+ Може містити поля класу	- Не містить реалізації для властивостей
Множине успадкування	- Може успадковувати від лише одного класу	+ Може бути реалізовано множинне успадкування
Заміщення для обєктів	- Не може бути використано для опису структури	+ Частіше використовується для опису структури
Спільний код	+ Може містити спільний код	- Не містить реалізації для спільного коду
Поліморфізм	+ Дозволяє поліморфізм через успадкування	+ Дозволяє поліморфізм через реалізацію інтерфейсу
Вимагання реалізації	+ Може вимагати реалізації методів	- Вимагає реалізації всіх методів

Загалом, вибір між абстрактними класами і інтерфейсами залежить від конкретних вимог вашого проекту. Використовуйте абстрактні класи, коли вам потрібно спільний код та реалізацію, і використовуйте інтерфейси для визначення контракту без конкретної реалізації.

6.4)Типи функцій

У TypeScript ви можете вказати тип для параметрів та результату функції.

```
// Тип функції з числовими параметрами та поверненням числа
type MathFunction = (a: number, b: number) => number;

const add: MathFunction = (a, b) => a + b;
const multiply: MathFunction = (a, b) => a * b;
```

6.5)Замикання в TS

Замикання в TypeScript використовується для створення функцій, які можуть отримувати доступ до змінних, визначених в їхньому зовнішньому контексті.

```
function outerFunction(x: number): (y: number) => number {
    return function innerFunction(y: number): number {
        return x + y;
    };
}

const addFive = outerFunction(5);
console.log(addFive(3)); // Виведе: 8
```

6.6)Асинхронні функції та обіцянки

Асинхронні функції використовуються для виконання асинхронних операцій та повернення обіцянок.

Обіцянки (Promises) використовуються для обробки асинхронних операцій та отримання результату або обробки помило

7 React

- **JSX (JavaScript XML):** JSX це розширення синтаксису JavaScript, яке дозволяє писати код, схожий на XML або HTML, всередині JavaScript. Він трансформується у виклики React.createElement під час компіляції.

- **Віртуальний DOM:** Це концепція, що дозволяє React вирішувати проблеми ефективного оновлення DOM, створюючи віртуальне представлення DOM, яке порівнюється з реальним для визначення змін.

- **Компонент в React:** Це незалежна одиниця інтерфейсу, яка може бути повторно використана. Він може бути класовим або функціональним.

- **Класові та функціональні компоненти:** Класові компоненти мають власний стан і методи життєвого циклу, тоді як функціональні компоненти базуються на функціях та використовують хуки для роботи зі станом та ефектами.

- **Пропси (властивості):** Це механізм передачі даних в компоненти в React. Вони передаються в компоненти вигляді атрибутів та використовуються для взаємодії між батьківським та дочірнім компонентами.

- **Обробка подій:** Події обробляються за допомогою обробників подій, наприклад, **onClick** для кліку мишею. Функції-обробники визначаються в компоненті та передаються як пропси.

GIT

Ось детальна інструкція:

Створення репозиторію на GitHub:

Перейдіть на веб-сайт GitHub. Натисніть кнопку "New" для створення нового репозиторію. Заповніть необхідну інформацію та створіть репозиторій.

Ініціалізація локального репозиторію:

В командному рядку перейдіть в каталог вашого проекту. Використовуйте git init, щоб ініціалізувати локальний репозиторій.

code git init

Додавання віддаленого репозиторію:

Додайте віддалений репозиторій на GitHub за допомогою команди

git remote add.

git remote add origin https://github.com/IvanDeresh/crypto.git

Додавання файлів, коміт і відправка на GitHub:

Додайте файли, використовуючи git add. Зафіксуйте ваші зміни комітом, використовуючи git commit. Відправте ваші зміни на GitHub за допомогою git push

code git add .

git commit -m "Перший коміт"

git push -u origin master

Створення нової гілки для роботи над головною сторінкою:

Створіть нову гілку і переключіться на неї

code git checkout -b mainpage

Внесення змін, коміт і відправка гілки на GitHub:

Внесіть зміни у ваші **файли**. Зафіксуйте зміни комітом та відправте гілку на GitHub.

code git add .

git commit -m "Зміни для головної сторінки"

git push origin mainpage

Після цих кроків ви можете перейти на GitHub, переглянути ваш репозиторій та створити pull request для гілки "mainpage".

В ВИЛИКИХ ПРОЕКТАХ

На великих проектах, де багато розробників спільно працює над кодом, використання ефективного процесу управління версіями та співпраці з командою є ключовим аспектом. Ось загальний процес роботи з git на великих проектах:

Структура гілок:

Зазвичай великі проекти мають головну гілку (наприклад, main, master), а також різні гілки для функціональності, виправлень помилок та інших робочих груп. Це сприяє відокремленості різних аспектів розробки.

Pull Request (PR) та Code Review:

Розробники створюють власні гілки для нових функцій чи виправлень. Після завершення роботи, вони створюють Pull Request (PR) для обговорення та злиття з головною гілкою. Code Review: Інші розробники або ті, хто відповідає за Code Review, переглядають код, залишають коментарі та роблять рекомендації.

Continuous Integration (CI) та Automated Tests:

Перед злиттям коду в головну гілку запускаються автоматизовані тести (Continuous Integration). Це допомагає виявляти можливі проблеми та забезпечує стабільність кодової бази.

Злиття та Pull Merge:

Після успішного Code Review та пройдених тестах Pull Request може бути злитий (merged) в головну гілку. Pull Merge: Зазвичай це означає витягування (pull) змін з головної гілки перед створенням свого Pull Request, щоб врахувати зміни, які можуть відбутися в репозиторії під час роботи над функцією.

Версіювання та Тегування:

Зазвичай великі проекти використовують системи версіювання для зручності відслідковування та випуску версій програмного забезпечення. Після злиття важливих функцій або виправлень може використовуватися тегування для позначення конкретних версій.

Git Hooks та Спеціальні Сценарії:

Використання Git Hooks для автоматизації певних дій перед або після виконання git команд (наприклад, виконання скриптів перед злиттям). Розробка спеціальних сценаріїв для роботи з git, які відповідають особливим потребам проекту.

Важливо зазначити, що конкретні практики можуть різнитися залежно від розробничого середовища та внутрішніх процесів команди. Однак використання гілок, Pull Request, Code Review та CI є загальноприйнятими засобами для забезпечення якості та спільної роботи над великими проектами.

ПАТЕРНИ

Патерни проєктування є загальноприйнятими рішеннями для типових проблем, що виникають при розробці програмного забезпечення. Вони дозволяють створювати високоякісне, ефективне та легко супроводжуване програмне забезпечення. Ось декілька основних категорій патернів проєктування:

СТАЛЕ РІШЕННЯ ЧАСТОЇ ПРОБЛЕМИ

Патерни структури:

Модель-Вид-Контролер (Model-View-Controller, MVC): Розділення програми на три основні компоненти - модель (дані та бізнес-логіка), вид (представлення інформації користувачу) та контролер (управління взаємодією між моделлю та видом).

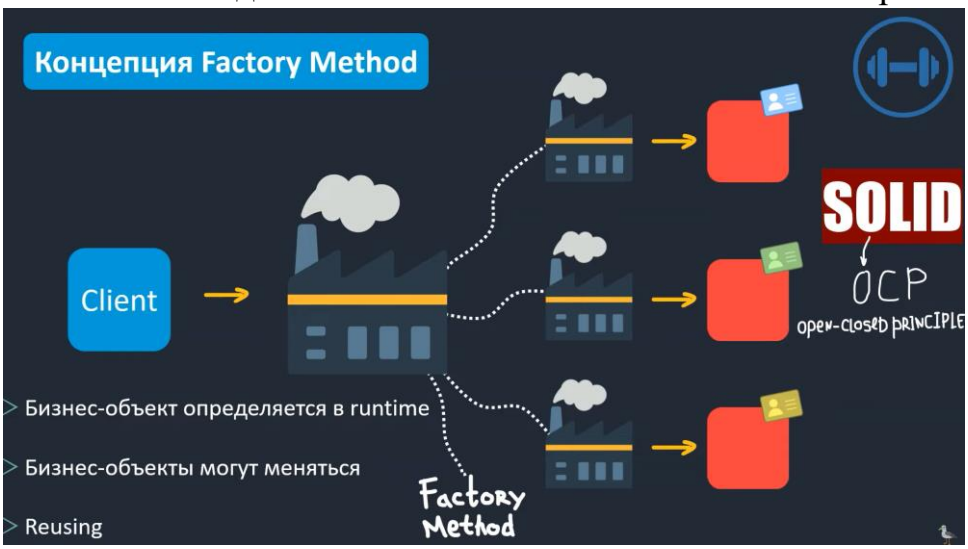
Компонент (Component): Розділення системи на незалежні компоненти, що можуть бути розроблені, тестовані та супроводжувані окремо.

Фасад (Facade): Надання простого інтерфейсу для складних систем для спрощення їх використання.

Патерни створення об'єктів:

Одиночка (Singleton): Гарантує, що клас має тільки один екземпляр та надає глобальний доступ до нього.

Фабричний метод (Factory Method): Надає інтерфейс для створення об'єктів, але залишає підкласам можливість змінювати тип створених об'єктів.



typescript Copy code

```typescript
// Інтерфейс або абстрактний клас для підписки
interface Subscription {
  getPrice(): number;
  getDetails(): string;
}

// Конкретна реалізація підписки "Базова"
class BasicSubscription implements Subscription {
  getPrice(): number {
    return 10;
  }

  getDetails(): string {
    return "Basic Subscription: Access to basic features.";
  }
}

// Конкретна реалізація підписки "Професійна"
class ProfessionalSubscription implements Subscription {
  getPrice(): number {
    return 20;
  }

  getDetails(): string {
    return "Professional Subscription: Access to advanced features.";
  }
}

// Конкретна реалізація підписки "Преміум"
class PremiumSubscription implements Subscription {
  getPrice(): number {
    return 30;
  }

  getDetails(): string {
    return "Premium Subscription: Access to premium features.";
  }
}

// Фабричний метод для створення різних видів підписок
interface SubscriptionFactory {
  createSubscription(): Subscription;
}

// Конкретна фабрика для базової підписки
class BasicSubscriptionFactory implements SubscriptionFactory {
  createSubscription(): Subscription {
    return new BasicSubscription();
  }
}

// Конкретна фабрика для професійної підписки
class ProfessionalSubscriptionFactory implements SubscriptionFactory {
  createSubscription(): Subscription {
    return new ProfessionalSubscription();
  }
}

// Конкретна фабрика для преміум підписки
class PremiumSubscriptionFactory implements SubscriptionFactory {
  createSubscription(): Subscription {
    return new PremiumSubscription();
  }
}
```

Задачі:

- **Розділення створення об'єктів від їх використання.** Фабричний метод дозволяє класам створювати екземпляри об'єктів, не пов'язуючи їхню реалізацію з кодом, який їх використовує.
- **Підтримка варіантів об'єктів.** Класи-нащадки можуть надавати свою реалізацію фабричного методу, щоб створювати різні типи об'єктів.
- **Забезпечення можливості розширення.** Зміна або додавання нових класів-нащадків не вимагає зміни коду класу, який використовує фабричний метод.

Плюси:

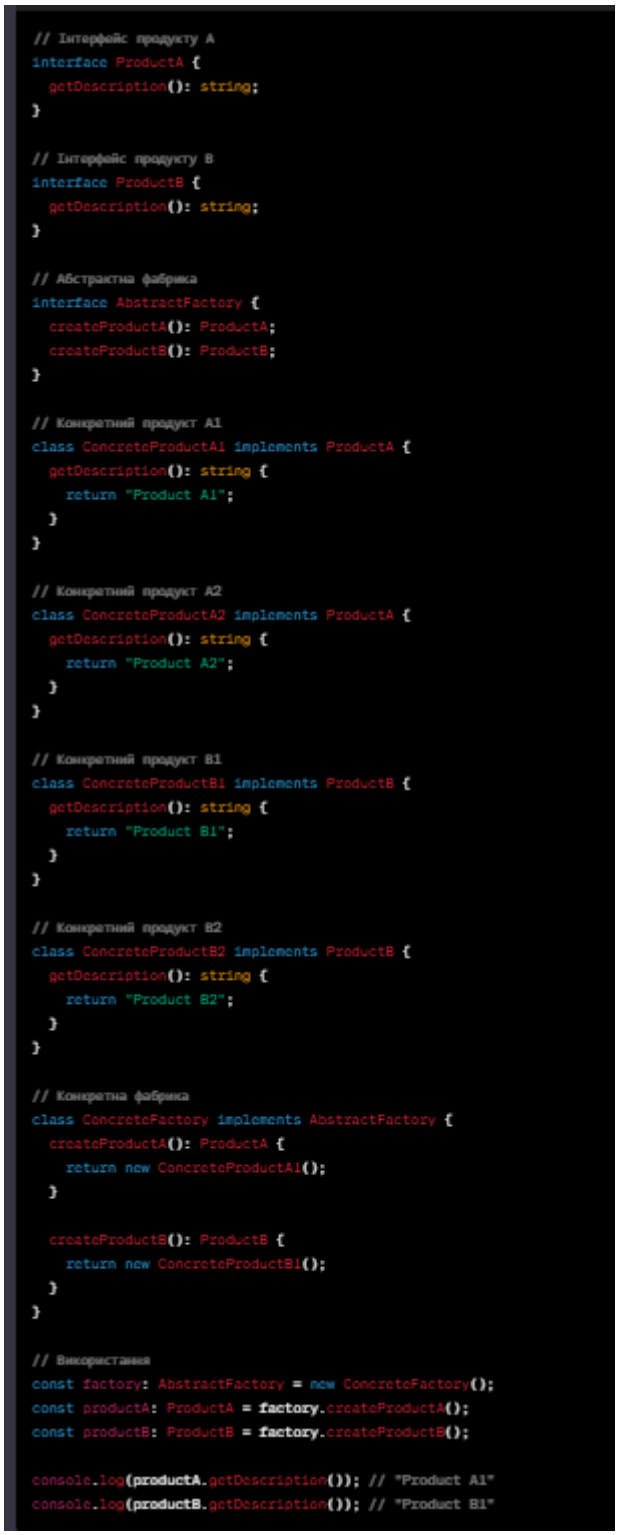
- **Гнучкість та розширюваність.** Фабричний метод дозволяє створювати об'єкти, не змінюючи код класу, що їх використовує. Нові класи-нащадки можуть легко додаватися без модифікації існуючого коду.
- **Зниження залежності від конкретних класів.** Замість прив'язки до конкретних класів, клас використовує інтерфейс або абстрактний клас для створення об'єктів. Це робить код менш залежним від конкретної реалізації.
- **Підтримка принципу відкритості/закритості.** Клас з фабричним методом відкритий для розширення (шляхом додавання нових класів-нащадків), але закритий для змін.

Мінуси:

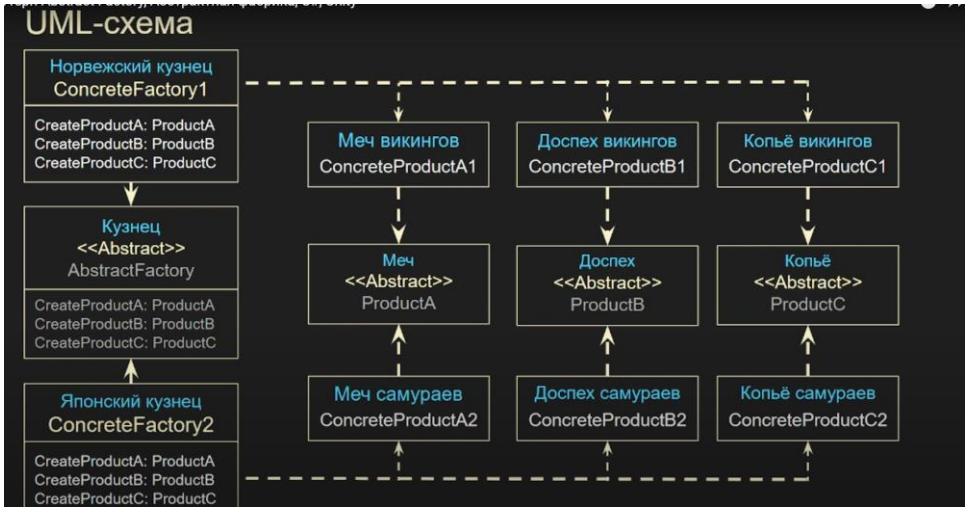
- **Дублювання коду.** Кожен клас-нащадок повинен реалізувати свій фабричний метод, що може призводити до дублювання коду.
- **Складність.** Введення фабричних методів може зробити код більш складним, особливо, якщо клас має декілька методів створення об'єктів.
- **Необхідність створення нового класу-нащадка для кожного нового типу об'єкта.** Це може призвести до збільшення кількості класів в системі.

Використання фабричного методу обговорюється та приймається на етапі проєктування, коли потрібно вирішити питання, пов'язане зі створенням об'єктів в залежності від умов чи потреб системи.

Абстрактна фабрика (Abstract Factory): Надає інтерфейс для створення сімейств взаємопов'язаних або взаємозалежних об'єктів без зазначення їх конкретних класів.



```typescript
// Інтерфейс продукту A
interface ProductA {
  getDescription(): string;
}

// Інтерфейс продукту B
interface ProductB {
  getDescription(): string;
}

// Абстрактна фабрика
interface AbstractFactory {
  createProductA(): ProductA;
  createProductB(): ProductB;
}

// Конкретний продукт A1
class ConcreteProductA1 implements ProductA {
  getDescription(): string {
    return "Product A1";
  }
}

// Конкретний продукт A2
class ConcreteProductA2 implements ProductA {
  getDescription(): string {
    return "Product A2";
  }
}

// Конкретний продукт B1
class ConcreteProductB1 implements ProductB {
  getDescription(): string {
    return "Product B1";
  }
}

// Конкретний продукт B2
class ConcreteProductB2 implements ProductB {
  getDescription(): string {
    return "Product B2";
  }
}

// Конкретна фабрика
class ConcreteFactory implements AbstractFactory {
  createProductA(): ProductA {
    return new ConcreteProductA1();
  }

  createProductB(): ProductB {
    return new ConcreteProductB1();
  }
}

// Використання
const factory: AbstractFactory = new ConcreteFactory();
const productA: ProductA = factory.createProductA();
const productB: ProductB = factory.createProductB();

console.log(productA.getDescription()); // "Product A1"
console.log(productB.getDescription()); // "Product B1"
```



Задачі:

- **Створення сімейств взаємопов'язаних або взаємозалежних об'єктів.** Абстрактна фабрика дозволяє визначити інтерфейс для створення не тільки окремих об'єктів, але й сімейства об'єктів, які взаємодіють між собою.
- **Забезпечення заміни сімейства продуктів.** Заміна абстрактної фабрики дозволяє легко перейти від одного сімейства продуктів до іншого без зміни коду класів, які використовують цю фабрику.

Плюси:

- **Гармонізація продуктів.** Абстрактна фабрика забезпечує гармонізацію продуктів, які взаємодіють між собою. Наприклад, якщо є два сімейства продуктів (A1, B1) та (A2, B2), то вони гармонійно співпрацюватимуть, але взаємодія між (A1, B2) може бути менш ефективною.
- **Легка можливість заміни сімейств продуктів.** Введення нової фабрики дозволяє легко переходити від одного сімейства продуктів до іншого, що робить систему більш гнучкою та легко адаптуючоюся до змін.
- **Виокремлення створення об'єктів від їх використання.** Класи, що використовують абстрактну фабрику, не пов'язані з конкретними класами продуктів, що полегшує розширення та зміни системи.

Мінуси:

- **Складність.** Абстрактна фабрика може виглядати дещо складною, особливо при великій кількості класів продуктів та їх варіантів.
- **Збільшення кількості класів.** Щоб додати новий продукт, потрібно створити новий клас для кожної абстракції (наприклад, якщо додається нова фабрика, яка повинна підтримувати новий продукт).
- **Складність розширення.** Додавання нового сімейства продуктів або внесення змін до існуючих може бути складним, оскільки це може вимагати модифікації багатьох класів та фабрик.

Абстрактна фабрика використовується там, де є потреба у створенні об'єктів, які взаємодіють між собою та належать до конкретного сімейства. Цей патерн особливо ефективний в складних системах, де інтерфейси продуктів та їх взаємодії можуть варіюватися від одного сценарію до іншого.

Патерни поведінки:

Стан (State): Дозволяє об'єкту змінювати свою поведінку при зміні внутрішнього стану.

Спостерігач (Observer): Забезпечує механізм визначення залежностей між об'єктами, так щоб зміна стану одного об'єкта призводила до змін у всіх йому залежних об'єктах.

Стратегія (Strategy): Визначає сімейство алгоритмів, і робить їх взаємозамінними, дозволяючи об'єкту вибрати підхід на льоту.

Патерни архітектури:

Мікросервіси (Microservices): Розбиття програмного забезпечення на невеликі та незалежні мікросервіси, які можуть функціонувати незалежно один від одного.

Шарова архітектура (Layered Architecture): Розподіл програми на різні рівні (шари), де кожен рівень відповідає конкретному виду функціональності.