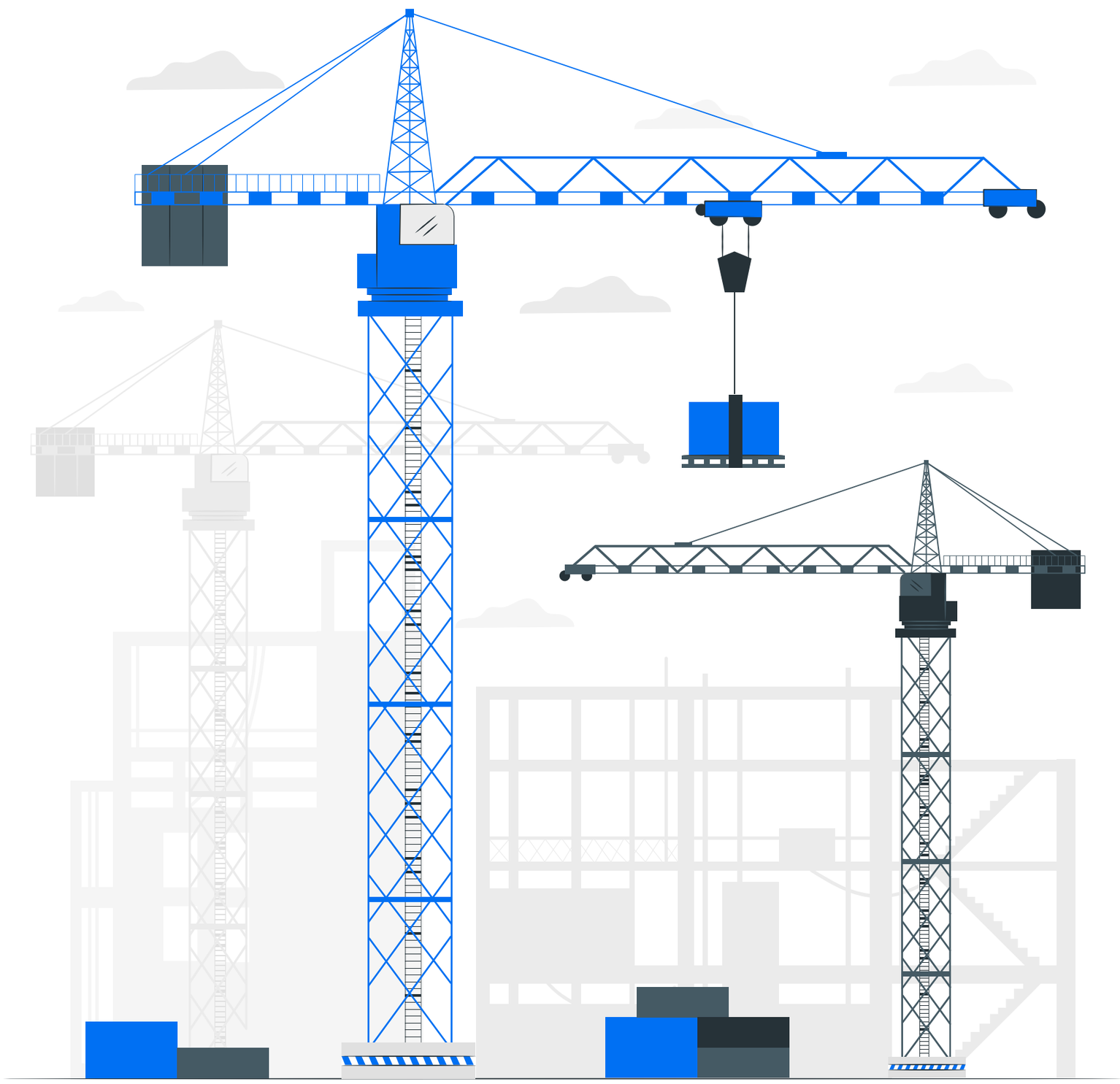
ط

ما الفرق بين
الـ Abstract Class والـ Interface ؟

هل يمكننا أن نستغنى عن أحدهما ونفضل الآخر؟



Ahmed El-Tabarani

Back-End Developer

ما معنى Abstract Member

قبل كل شيء عليك ان تعرف ما هو

الـ Abstract Member، هو مسمى يشير إلى

الـ Abstract Property أو Abstract Method

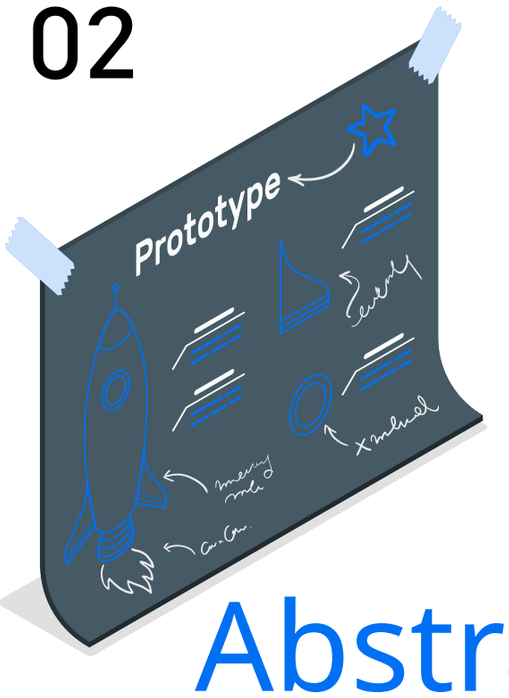
و أي Abstract Member لا يتم وراثته، بل يتم

اعادة بناءه وعمل overriding له

وimplementation بشكل اجباري

Ahmed El-Tabarani

Back-End Developer

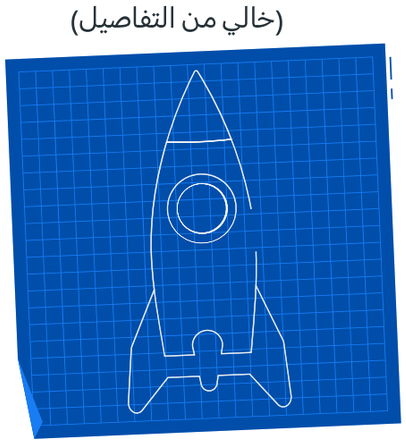


أولًا ما العوامل المشترك بين الـ Abstract Class والـ Interface ؟

- يندرجان تحت مفهوم الـ Abstraction وهو المفهوم الذي يركز على إخفاء أي تفاصيل أو implementation للأشياء

- كلاهما يُستخدما في تمثيل أشياء أو كيانات عامة مثل الإنسان، السيارة أو الحيوان أو غيرها من الوصف العام لأي كيان خالي من أي تفاصيل، مثل أن الأسد هو الحيوان

Interface
(خالي من التفاصيل)



implementation لتفاصيل كيان

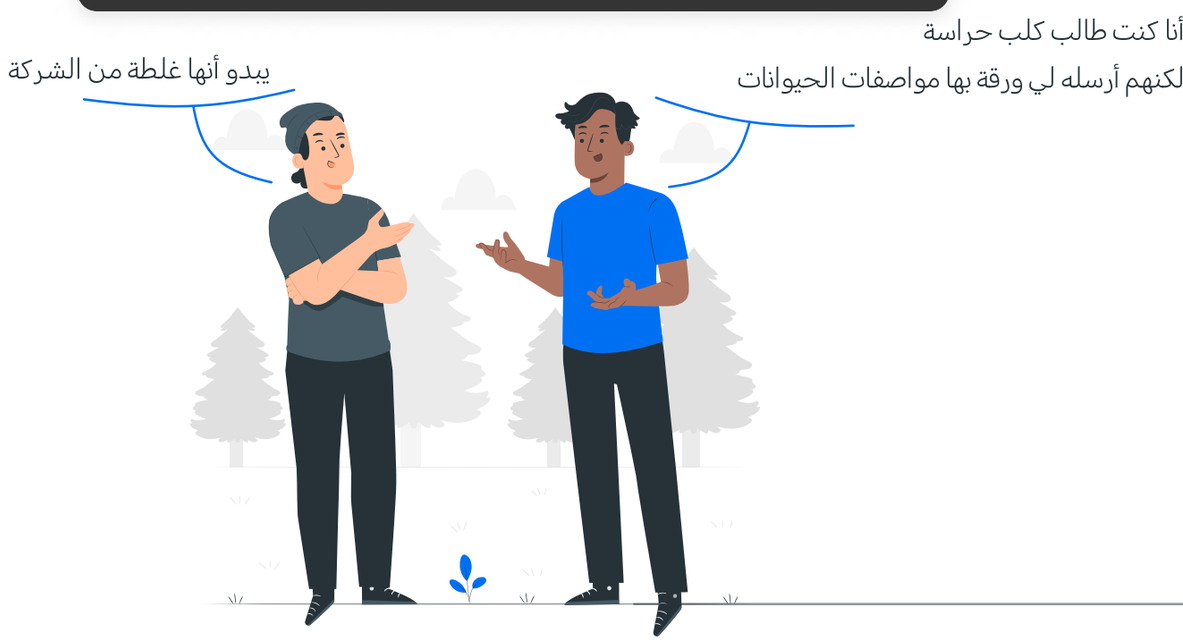
Implementation
(تفاصيل)

Ahmed El-Tabarani

Back-End Developer

- لا يمكن إنشاء object منهما لعدم احتواءهما على تفاصيل

```
new Animal(); // Error!
```



- يمكنهما إجبار الكلاسات التي سترثهما بعمل implementation اجباري لـ abstract member

Ahmed El-Tabarani

Back-End Developer

ثانيًا ما أهم الإختلافات بين الـ Abstract Class والـ Interface ؟

Abstract Class	Interface
يخضع لمفهوم الـ Inheritance بالتالي يمكنك وراثة دوال تحتوي على تفاصيل يتم استخدام extends لكي تقوم الكلاسات بوراثة كل ما يقدمه وتبني فقط الـ abstract method `class Lion extends Animal`	لا يخضع لمفهوم الـ Inheritance بالتالي لا يتم وراثة أي شيء يحتوي على تفاصيل يتم استخدام implements لكي تقوم الكلاسات ببناء كل ما يقدمه بشكل اجباري `class Lion implementes IAnimal`
قد يملك abstract member واحدة على الأقل وقد يملك دوال عادية لها implementation لاحظ أن بعض الدوال **لها** body وImplementation `abstract class Animal { constructor(protected name: string) { } abstract printInfo(): void; public getName() { return this.name; } }`	جميع دواله ومتغيراته تكون abstract member بشكل إجباري لاحظ الدوال **ليس لها** body ولا Implementation `interface IAnimal { // forced to be abstract properties name: string biome: string // forced to be abstract methods setName(name: string): void; setBiome(biome: string): void; }`
يحتوي على constructor، لكن لا يمكن إنشاء منه أي object `class Lion extends Animal { constractor(){ // Ok! Can call super constractor super(); // Ok! Can call non-abstract method super.printInfo(); } } // Error! But, can't make an object new Animal()`	لا يحتوي على constructor، لا يمكن إنشاء منه أي object `class Lion implement IAnimal { constractor(){ // Error! There is no super constractor super(); } } // Error! Can't make an object new IAnimal()`
كل كلاس يستطيع أن يرث Abstract Class واحد فقط لا غير `class Lion extends Animal, Wild { } // Error! Classes can extend only one class` السبب يكمن في صعوبة وراثة أكثر من constructor لكلاسات مختلفة في آن واحد والتعامل مع مشكلة مشهورة في عالم الـ Inheritance الخاصة بالكلاسات وهي الـ Multiple InherInheritance	يمكن للكلاس الواحد أن يبني أكثر من Interface `class Lion implementes IAnimal, IWild { /* Forced to implement everything ... */ }`

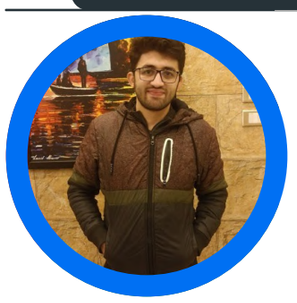
Ahmed El-Tabarani
Back-End Developer

لاحظ الآتي

الـ interface ليس كلاس، بالتالي كل شيء قد تعرفه عن الكلاسات لن يتواجد فيه

بل هو مجرد نموذج وصفي بحت، مفرغ تمامًا من أي implementation أو أي تفاصيل

على عكس شقيقه الـ Abstract Class الذي ينطبق عليه فكرة الكلاس وقد يحتوى على بعض التفاصيل والـ implementation



Ahmed El-Tabarani

Back-End Developer

استخدام كلًا من
الـ Abstract Class والـ Interface في آن واحد

يمكننا ببساطة استخدامُهما معًا بسهولة

يمكننا جعل الـ Abstract Class هو الذي
سيقوم بعمل implements للـ Interface
ثم نجعل الكلاسات تتعامل
مع الـ Abstract Class وترث منه

```ts
interface IEmployee {
  name: string;
  companyName: string;

  work(): void;
  getSalary(): number;
  getCompanyName(): string;
}
```

النموذج الأساسي

قام ببناء النموذج وأضاف بعض التفاصيل

```ts
abstract class Employee implements IEmployee {
  constructor(public name: string, public companyName: string) {

  // abstract methods
  public abstract work(): void;
  public abstract getSalary(): number;

  // non-abstract method
  public getCompanyName() {
    return this.companyName;
  }
}
```

التفاصيل الذي يضيفها قد تكون:

- عمل constractor
- يقوم بعمل override لدالة
- ينشيء دالة جديدة عامة

Ahmed El-Tabarani
Back-End Developer

الآن يمكننا عمل كلاس كـ CareemEmployee يقوم بوراثة الـ Employee بشكل اعتيادي

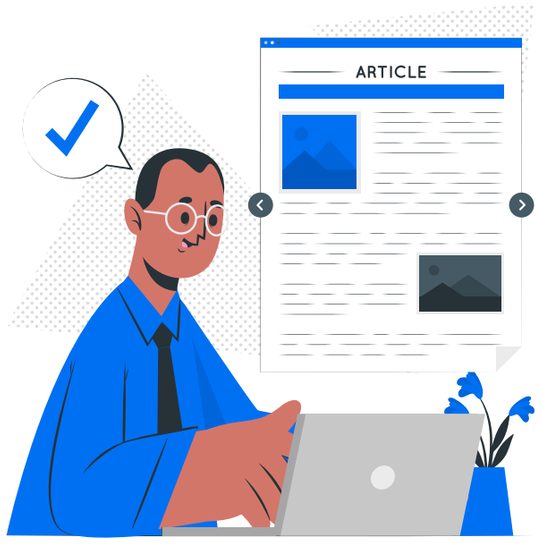
```
class CareemEmployee extends Employee {
  public getSalary(): number {
    return 8000;
  }
  public work() {
    console.log(`${this.name} helps people get around!`);
  }
}

let e1 = new CareemEmployee('Ahmed', 'Careem');
let salary = e1.getSalary();

e1.work(); // Ahmed helps people get around!
console.log(`Net salary is: ${salary}`); // Net salary is 8000
```

لاحظ أن الـ Interface كان هو الواجهة الذي يجب أن يتم اتباعها، ثم قام الـ Abstract Class ببناءه ووضع بعض التفاصيل له مثل إضافة constractor وعمل implementation لدالة getCompanyName لأنها دالة لها وظيفة ثابتة وعامة كما تلاحظ

ثم يأتي CareemEmployee ليرث من الـ Abstract Class

Ahmed El-Tabarani
Back-End Developer

طـ



خاتمة!

كما ترى كلاهما لهما مميزات ويمكنك استخدامهما مع بعض بطريقة سلسلة فالـ Interface هي تضع الأساسيات والـ Abstract Class يقوم ببناءها

وكلاهما يطبقان مفهوم الـ Abstraction لكن بطرق مختلفة فمعرفة مميزات ووظيفة كل منهما سيجعل العمل اكثر تنظيما وقابل للتعديل وخالي من الأخطاء قدر المستطاع

Ahmed El-Tabarani
Back-End Developer

أنا أحمد الطبراني، مهندس برمجيات SWE 😁

مبرمج متخصص في عالم الـ Backend ⚙️💻

أحب دائمًا أن أُشارك معرفتي المتواضعة مع الآخرين لعله عسى أن يستفيد شخص ما بما أكتبه 🤗
أكتب بعض المقالات عن أشياء مختلفة في عالم البرمجة، أرجوا أن تستفيدوا وتستمتعوا 😉

تابعني علي لينكدإن او علي مدونتي الشخصية