• Question: Why does defining a custom constructor suppress the default constructor in C#?

**ما هو المُنشئ الافتراضي؟**

* المُنشئ الافتراضي هو مُنشئ بدون أي معلمات (Parameter less Constructor)، يتم إنشاؤه تلقائيًا من قبل المترجم إذا لم تقم بتعريف أي مُنشئ داخل الكلاس.
* وظيفته تهيئة الكائنات بقيم افتراضية (مثل 0 للأرقام وnull للكائنات).

مثال:

public class My Class

{

}

public class My Class

{

public My Class() { }

}

**لماذا يتم إلغاء المُنشئ الافتراضي عند تعريف مُنشئ مخصص؟**

* عندما تقوم بتعريف مُنشئ مخصص (سواء بوجود معلمات أو لا)، يفترض المترجم أنك تعرف ما تحتاجه لتهيئة الكائنات وأنك لا تحتاج إلى المُنشئ الافتراضي.
* لذلك، إذا كنت بحاجة إلى مُنشئ افتراضي إلى جانب المُنشئ المخصص، يجب عليك تعريفه صراحة.

مثال:

public class My Class

{

public My Class(int value)

{

// منطق التهيئة الخاص

}

public My Class() { }

}

Question: How does method overloading improve code readability and reusability?

**1. تحسين وضوح الكود (Code Readability)**

* **اسم واحد لوظائف متشابهة**: يوفر تعدد التحميل طريقة لتجميع الوظائف ذات الأغراض المتشابهة تحت اسم واحد، مما يجعل الكود أكثر تنظيماً وأسهل للفهم.
  + مثال:

public class Calculator

{

public int Add (int a, int b) {return a + b;}

public double Add (double a, double b) {return a + b;}

public int Add (int a, int b, int c) {return a + b + c;}

}

بدلاً من استخدام أسماء مثل Add Integers أو Add Doubles، يمكنك استخدام اسم واحد مثل Add لجميع العمليات، مما يجعل الكود أكثر وضوحًا للمبرمجين.

* **تحسين تدفق الكود**: لا يحتاج المبرمج إلى تذكر أسماء متعددة لدوال متشابهة؛ يكفيه معرفة اسم الدالة فقط، وسيتولى المترجم اختيار الإصدار الصحيح بناءً على المعاملات.

**2. تحسين قابلية إعادة الاستخدام (Code Reusability)**

* **تقليل تكرار الكود**: يسمح تعدد التحميل بتقليل الحاجة إلى كتابة دوال جديدة بنفس الوظيفة لكن بمعاملات مختلفة، مما يقلل من تكرار الكود.
  + مثال:

public class Printer

{

public void Print (string text) {Console. WriteLine(text);}

public void Print (int number) {Console. WriteLine (number);}

public void Print (double value) {Console. WriteLine (value);}

}

هنا يمكنك استخدام دالة Print لطباعة أنواع مختلفة من البيانات دون الحاجة إلى كتابة دوال منفصلة لكل نوع.

* **توفير مرونة أكبر**: مع تعدد التحميل، يمكن استخدام نفس الدالة لسيناريوهات مختلفة مع تحسين قابلية التوسع وإعادة الاستخدام.

Question: What is the purpose of constructor chaining in inheritance?

**الغرض من سلسلة المُنشئات (Constructor Chaining) في الوراثة**

سلسلة المُنشئات (**Constructor Chaining**) هي تقنية تُستخدم في البرمجة لتسهيل استدعاء مُنشئ (Constructor) من مُنشئ آخر، سواء داخل نفس الكلاس أو بين الكلاسات في حالة الوراثة. الهدف الأساسي منها في الوراثة هو **ضمان تهيئة الكائنات بشكل صحيح** من خلال التأكد من تهيئة الكلاس الأساسي (Base Class) أولاً قبل تهيئة الكلاس المشتق (Derived Class).

**أهداف سلسلة المُنشئات في الوراثة:**

1. **تهيئة الكلاس الأساسي (Base Class):**
   * عندما يرث كلاس مشتق (Derived Class) من كلاس أساسي، يجب تهيئة جميع أعضاء الكلاس الأساسي قبل تنفيذ منطق الكلاس المشتق.
   * يتم استخدام سلسلة المُنشئات للتأكد من استدعاء مُنشئ الكلاس الأساسي أولاً.

**مثال:**

public class Base Class

{

public Base Class()

{

console. WriteLine ("Base Class Constructor");

}

}

public class Derived Class: Base Class

{

public Derived Class ()

{

console. WriteLine ("Derived Class Constructor");

}

}

1. **إعادة استخدام الكود:**
   * بدلاً من كتابة نفس منطق التهيئة في كل مُنشئ، يمكن استدعاء مُنشئ آخر يحتوي على المنطق المشترك.
   * هذا يساعد على تجنب التكرار وزيادة كفاءة الكود.

public class Person

{

public string Name;

public int Age;

public Person (string name)

{

Name = name;

}

public Person (string name, int age): this(name)

{

Age = age;

}

}

1. **مرونة أكبر في التهيئة:**
   * يُتيح للمطورين تمرير معلومات مختلفة للكلاس الأساسي من خلال مُنشئات الكلاس المشتق.
   * هذا يعزز مرونة التهيئة عند إنشاء الكائنات.

**مثال مع تمرير معطيات للكلاس الأساسي:**

public class Animal

{

public string Species;

public Animal (string species)

{

Species = species;

}

}

public class Dog: Animal

{

public Dog (): base("Dog")

{

console. WriteLine ("Derived Class Dog Initialized");

}

}

• Question: How does new differ from override in method overriding?

**1. override (تجاوز الدالة الأساسية):**

* تُستخدم لتوفير **تنفيذ جديد** لدالة معرفة مسبقًا في الكلاس الأساسي.
* تتطلب أن تكون الدالة الأساسية مُعلّمة بالكلمة المفتاحية virtual أو abstract أو override.
* يظل اسم الدالة الأساسي صالحًا للإشارة إلى التنفيذ الجديد عبر الكائنات من نوع الكلاس الأساسي (Polymorphism).

**مثال:**

public class Base Class

{

public virtual void Display ()

{

Console. WriteLine ("Base Class Display");

}

}

public class derived Class : Base Class

{

public override void Display ()

{

console. WriteLine ("Derived Class Display");

}

}

Base Class obj = new Derived Class();

obj.Display();

**الخصائص:**

* يدعم **تعدد الأشكال (Polymorphism)**.
* يتم استدعاء النسخة المشتقة من الدالة عند استخدام كائن من النوع الأساسي يشير إلى كائن من النوع المشتق.

**2. new (إخفاء الدالة الأساسية):**

* تُستخدم لإخفاء (Hide) الدالة الأساسية بدلاً من تجاوزها.
* لا تتطلب أن تكون الدالة الأساسية مُعلّمة بـ virtual.
* عند استدعاء الدالة باستخدام مرجع إلى الكلاس الأساسي، يتم تنفيذ النسخة الأساسية للدالة، وليس النسخة المشتقة.

**مثال:**

public class Base Class

{

public void Display ()

{

Console. WriteLine ("Base Class Display");

}

}

public class Derived Class : Base Class

{

public new void Display ()

{

console. WriteLine ("Derived Class Display");

}

}

Base Class obj = new Derived Class();

obj.Display();.

Derived Class obj Derived = new Derived Class();

objDerived.Display();

**الخصائص:**

* لا يدعم **تعدد الأشكال (Polymorphism)**.
* النسخة الأساسية من الدالة تُنفذ عند استدعاء الدالة باستخدام مرجع من النوع الأساسي.

• Question: Why is To String () often overridden in custom classes?

**لماذا يتم غالبًا تجاوز الدالة ToString () في الكلاسات المخصصة؟**

الدالة **ToString ()** في لغة C# تُستخدم لتحويل كائن إلى تمثيل نصي (String Representation). الكلاس الأساسي **Object** يوفر تنفيذًا افتراضيًا لهذه الدالة، ولكن هذا التنفيذ غالبًا ما يكون غير مفيد للكلاسات المخصصة، حيث يقوم فقط بإرجاع اسم نوع الكائن (Type Name). لذلك، يتم تجاوزها لتقديم تمثيل نصي أكثر وضوحًا وذات معنى للكائن.

**أسباب تجاوز الدالة ToString () في الكلاسات المخصصة:**

1. **توفير تمثيل نصي ذو معنى للكائن:**
   * التمثيل الافتراضي الذي تُعيده ToString () غير كافٍ في معظم الحالات. تجاوز الدالة يسمح لك بإرجاع معلومات مفيدة عن الكائن، مثل قيم خصائصه أو حالته.
   * **مثال:**

public class Person

{

public string Name {get; set;}

public int Age {get; set;}

public override string ToString ()

{

return $"Name: {Name}, Age: {Age}";

}

}

Person person = new Person {Name = "Ahmed", Age = 25};

Console.WriteLine(person. ToString ());

1. **تسهيل تصحيح الأخطاء (Debugging):**
   * عند طباعة الكائنات أو عرضها في أدوات التصحيح (Debugging Tools)، يتم استدعاء ToString() افتراضيًا. تجاوز الدالة يجعل التصحيح أسهل من خلال توفير معلومات واضحة ومحددة عن الكائن.
2. **تخصيص السلوك عند تحويل الكائن إلى نص:**
   * تجاوز الدالة يسمح لك بالتحكم الكامل في كيفية تمثيل الكائن كنص، بما يتناسب مع احتياجات التطبيق.
3. **تحسين قراءة الكود:**
   * عند طباعة الكائنات مباشرةً (مثلاً باستخدام Console.WriteLine())، يُستدعى ToString() تلقائيًا. تجاوزها يجعل الإخراج النصي للكائنات أكثر قابلية للفهم.

**التمثيل الافتراضي مقابل التمثيل المخصص:**

* **التمثيل الافتراضي:**

public class Person

{

public string Name {get; set;}

public int Age {get; set;}

}

Person person = new Person { Name = "Ahmed", Age = 25 };

Console. WriteLine(person.ToString());

* **التمثيل المخصص:**

public override string To String ()

{

return $"Name: {Name}, Age: {Age}";

}

• Question: Why can't you create an instance of an interface directly?

في لغة **C#**، لا يمكنك إنشاء كائن (instance) من **واجهة (interface)** بشكل مباشر لنفس الأسباب الأساسية الموجودة في لغات البرمجة الأخرى مثل Java. السبب الرئيسي هو أن الواجهة (interface) لا تحتوي على أي تنفيذ (implementation) للوظائف التي تعرّفها، وإنما تُستخدم لتحديد عقد (contract) يجب على الكلاسات (classes) التي تنفذها الالتزام به.

**الأسباب بالتفصيل:**

**. الواجهة مجردة (Abstract) .1**

الواجهات في C# مجردة بالكامل، أي أنها تحتوي فقط على تعريف (توقيع) الدوال (methods) أو الخصائص (properties) دون أي كود فعلي ينفذها. لذلك، لا يمكن إنشاء كائن منها لأنه لا يوجد أي منطق يمكن تنفيذه.

مثال:

public interface IVehicle

{

void StartEngine();

}

إذا حاولت:

IVehicle vehicle = new IVehicle ();

**الغرض من الواجهات. 2**

الغرض من الواجهات هو تقديم **عقد (contract)** للسلوكيات التي يجب أن تلتزم بها الكلاسات التي تنفذ الواجهة. الكلاسات التي تنفذ الواجهة هي التي تُوفر تنفيذ الوظائف والخصائص المحددة في الواجهة.

مثال:

public class Car: IVehicle

{

public void StartEngine ()

{

Console. WriteLine ("تم تشغيل محرك السيارة!");

}

}

هنا، الكلاس Car يُنفذ الواجهة I Vehicle ويُوفر التنفيذ الفعلي للدالة StartEngine.

**استخدام الواجهة كمؤشر (Reference Type) .3**

في C#، يمكنك استخدام واجهة كمؤشر (reference type) للإشارة إلى كائن من كلاس يُنفذ هذه الواجهة. هذا يسمح باستخدام ميزات التعددية (polymorphism).

مثال:

I Vehicle my Car = new Car();

myCar.StartEngine(); // النتيجة: تم تشغيل محرك السيارة!

my Car هو متغير من نوع الواجهة I Vehicle، لكنه يشير إلى كائن من نوع Car الذي يُنفذ الواجهة. هنا،

• Question: Why is it useful to use an interface reference to access implementing class methods?

استخدام مرجع من نوع **واجهة (interface)** للوصول إلى أساليب (methods) الكلاسات التي تنفذ الواجهة مفيد جدًا لأنه يوفر مزايا متعددة تتعلق بالتصميم البرمجي، مثل **التجريد (abstraction)**، **المرونة**، و**التعددية الشكلية (polymorphism)**.

**الفوائد بالتفصيل:**

**. دعم التعددية الشكلية (Polymorphism). 1**

* التعددية الشكلية تتيح كتابة كود يمكنه التعامل مع عدة أنواع من الكلاسات باستخدام مرجع واحد مشترك من نوع الواجهة.
* مرجع الواجهة يمكن أن يشير إلى أي كائن من كلاس ينفذ هذه الواجهة، مما يجعل الكود أكثر مرونة وقابلية للتوسيع.

**مثال**:

public interface IShape {

void Draw ();

}

public class Circle: IShape {

public void Draw () {

Console.WriteLine("رسم دائرة");

}

}

public class Rectangle: IShape {

public void Draw () {

Console.WriteLine("رسم مستطيل");

}

}

class Program {

static void Main (string [] rags) {

IShape shape1 = new Circle ();

IShape shape2 = new Rectangle ();

shape1.Draw();

shape2.Draw();

}

}

* هنا، shape1 وshape2 من نوع IShape، ولكن يمكنهما الإشارة إلى كائنات من كلاس Circle وRectangle.

**. تعزيز التجريد (Abstraction). 2**

* يتيح لك البرمجة ضد **عقد (contract)** أو واجهة بدلاً من البرمجة ضد كلاس محدد.
* هذا يعزل الكود الذي يستخدم الواجهة عن التفاصيل الداخلية للتنفيذ في الكلاسات.

**مثال**:

public void Render Shape (IShape shape) {

shape. Draw ();

}

* في هذا المثال، يمكن للوظيفة Render Shape التعامل مع أي كائن ينفذ الواجهة IShape دون الحاجة لمعرفة تفاصيل الكلاس.

**. سهولة تغيير وتنفيذ الكود (Flexibility). 3**

* إذا أردت إضافة كلاس جديد ينفذ الواجهة، لن تحتاج إلى تعديل الكود الذي يعتمد على الواجهة. يكفيك إضافة الكلاس الجديد.
* هذا يجعل الكود أكثر قابلية للتوسيع وإعادة الاستخدام.

**مثال**:

public class Triangle: IShape {

public void Draw () {

Console.WriteLine("رسم مثلث");

}

}

* عند إضافة Triangle، يمكن استخدامه مباشرة مع الكود الذي يعتمد على IShape دون أي تعديل.

**. تحسين الاختبار (Testability). 4**

* يمكن بسهولة استبدال تنفيذ الواجهة بكائن وهمي (mock) أو نموذج مبسط (stub) أثناء اختبار الكود، دون الحاجة إلى الكلاس الفعلي.

**مثال باستخدام مكتبة Mock**:

var mock Shape = new Mock<IShape>();

mockShape.Setup(shape => shape.Draw()).Callback(() => Console.WriteLine("رسم كائن وهمي"));

* هذا يسمح باختبار منطق الكود دون الحاجة إلى تنفيذ حقيقي.

**. .تشجيع استخدام مبادئ التصميم الجيدة5**

* البرمجة باستخدام مراجع الواجهة تطبق مبدأ **اعتماد الكود على التجريد (Dependency Inversion Principle)**، وهو أحد مبادئ التصميم الشهيرة (SOLID).
* الكود يصبح أكثر مرونة وقابلية للتغيير لأنه يعتمد على واجهة مجردة بدلاً من كلاس معين.

• Question: How does C# overcome the limitation of single inheritance with interfaces?

لغة **C#** تتغلب على قيود **الوراثة الأحادية** (Single Inheritance) من خلال السماح للكلاس بتنفيذ **أكثر من واجهة (Interface)**. في C#، يمكن للكلاس أن يرث من كلاس أساسي واحد فقط، ولكنه يستطيع تنفيذ عدد غير محدود من الواجهات، مما يتيح تحقيق مزايا الوراثة المتعددة دون التعقيدات المرتبطة بها.

**النقاط الرئيسية:**

**1.الواجهات توفر سلوك الوراثة المتعددة**

* + يسمح للكلاس بتنفيذ وظائف مختلفة عن طريق تنفيذ واجهات متعددة. هذا يجعل الكلاس يجمع بين سلوكيات متنوعة.

**مثال:**

public interface I Movable {

void Move ();

}

public interface I Rotatable {

void Rotate ();

}

public class Robot: I Movable, I Rotatable {

public void Move () {

Console.WriteLine("الروبوت يتحرك");

}

public void Rotate () {

Console.WriteLine("الروبوت يدور");

}

}

* + في هذا المثال، الكلاس Robot ينفذ الواجهتين I Movable وI Rotatable، مما يتيح له دمج السلوكيات المحددة في كل واجهة.

**تجنب مشكلة "الماس"2.**

* + في الوراثة المتعددة (مثل C++)، قد تحدث مشكلة "الماس" عندما يرث كلاس من اثنين من الكلاسات التي تشترك في نفس الكلاس الأساسي، مما يسبب غموضًا. C# تتجنب هذه المشكلة لأن الواجهات لا تحتوي على حالة (state) أو تنفيذ (implementation).

**3.المرونة وإعادة الاستخدام**

* + الواجهات تفصل بين تعريف السلوك وتنفيذه، مما يجعل الكود أكثر مرونة ويسهل إعادة استخدامه مع تقليل التبعية بين الكلاسات.

• Question: What is the difference between a virtual method and an abstract

**الطريقة الافتراضية** تُستخدم عند الحاجة إلى تنفيذ افتراضي يمكن تعديله.

public class Base Class {

public virtual void Display () {

Console.WriteLine("تنفيذ افتراضي من الكلاس الأساسي");

}

}

public class Derived Class : base Class {

public override void Display () {

Console.WriteLine("تنفيذ مخصص من الكلاس المشتق");

}

}

**الطريقة المجردة** تُستخدم عندما يكون السلوك غير معرف بالكامل ويجب تحديده في الكلاسات المشتقة.

public abstract class Abstract Class {

public abstract void Display ();

}

public class Concrete Class: Abstract Class {

public override void Display () {

Console.WriteLine("تنفيذ من الكلاس المشتق");

}

}