Contents

Object Oriented Programming – concepten

Abstractie

Definitie

Abstractie is een fundamenteel concept in objectgeoriënteerd (OO) ontwerp en programmeren. Het houdt in dat complexe real-world systemen worden vereenvoudigd door de essentiële kenmerken te isoleren en de details die niet essentieel zijn voor het huidige perspectief te verbergen. Hier is een gedetailleerde uitleg van abstractie in OO:

Wat is Abstractie?

Abstractie in OO draait om het vereenvoudigen van complexiteit door alleen de relevante kenmerken en gedrag van een object te benadrukken en de niet-essentiële details te verbergen. Dit helpt ontwikkelaars om te focussen op wat een object doet in plaats van hoe het dit doet.

Voordelen van Abstractie

1. **Vermindering van Complexiteit:**
   Abstractie helpt bij het beheersen van de complexiteit door alleen relevante details te tonen en de rest te verbergen.

2. **Modulariteit:**
   Door abstractie kunnen programma's worden opgedeeld in onafhankelijke modules, wat het onderhoud en de herbruikbaarheid van code vergemakkelijkt.

3. **Eenvoudigere Probleemoplossing:**
   Ontwikkelaars kunnen zich concentreren op een hoger niveau van de probleemruimte zonder zich te bekommeren om de onderliggende details.

4. **Verbeterde Veiligheid:**
   Abstractie kan helpen om de details te verbergen die niet toegankelijk hoeven te zijn voor andere delen van een programma, wat leidt tot beter beheer van de toegang en veranderingen.

(ChatGPT 3.5 – 03/07/2024)

Understanding Abstraction in OOP

Abstraction, in the context of OOP, refers to the ability to hide complex implementation details and show only the necessary features of an object. This simplifies the interaction with objects, making programming more intuitive and efficient. It provides a clear separation between what an object does and how it achieves its functionality, fostering a higher level of understanding and collaboration among developers.

(https://medium.com/@FirstBitSolutions/what-is-an-abstraction-in-object-oriented-programming-28a9f9501e23)

Abstraction is a way of simplifying things. In programming, it means showing only the necessary parts and hiding the rest. Imagine you're playing a video game, and you only see the game character on the screen, not all the coding and calculations happening behind it. That's abstraction—making things easier to use.

(https://www.almabetter.com/bytes/articles/abstraction-in-oops)

Imagine you have a car. You don't need to know how the engine, transmission, or other complex systems work in detail to drive it. All you need is a simplified interface — the steering wheel, pedals, and dashboard — to interact with the car. In this analogy, abstraction is like that simplified interface. It allows you to interact with objects in a straightforward way, without worrying about the intricate inner workings.

In OOP, abstraction provides a similar concept. It allows you to create classes or interfaces that define the essential features and behaviors of an object, while hiding the complex implementation details. These abstract representations serve as a blueprint for creating objects. They outline what an object can do, but don't specify how it does it.

(https://stackify.com/oop-concept-abstraction/)

Voorbeeld - Bestelling

Een bestelling kan bestaan uit verschillende producten (en hun aantal). We kunnen aan deze bestelling producten toevoegen en verwijderen. Daarnaast kan de prijs van de bestelling worden opgevraagd. Dat kan bijvoorbeeld resulteren in de volgende klasse.

```
public class Bestelling
{
    private Dictionary<Product, int> producten = new();
    public void VoegProductToe(Product product,int aantal)...
    public void VerwijderProduct(Product product,int aantal) ...
    public List<(Product product, int aantal)> GeefProducten()...
    public decimal BerkenPrijs()...
}
```

Op welke manier deze methodes worden geïmplementeerd en op welke manier de gegevens worden bijgehouden, is voor de gebruiker van onze klasse niet van belang, deze dient enkel te weten hoe de klasse er uitziet en wat ze kan.

```
public void VoegProductToe(Product product,int aantal)
{
    if (product == null) throw new BestellingException("product is null");
    if (aantal <= 0) throw new BestellingException("aantal <= 0");
    if (producten.ContainsKey(product))
    {
        producten[product] += aantal;
    }
    else
    {
        producten.Add(product, aantal);
    }
}
```

Encapsulation

Definitie

Inkapseling, ook wel encapsulation genoemd, is een fundamenteel concept in objectgeoriënteerde programmering (OOP) dat gaat over het verbergen van de interne details van een object en het bieden van een duidelijke interface voor interactie met dat object. Het doel van inkapseling is om de interne toestand van een object te beschermen tegen ongewenste of onbedoelde wijzigingen van buitenaf en om de toegang tot die interne toestand te controleren en te beheersen.

Kenmerken van Inkapseling

1. **Verbergen van interne details:**
   De interne data en implementatiedetails van een object zijn verborgen voor de buitenwereld. Dit betekent dat de buitenwereld geen directe toegang heeft tot de velden (attributen) van een object.

2. **Gedefinieerde toegangsmethoden:**
   Toegang tot de interne data wordt geboden via methoden die speciaal zijn ontworpen om deze data te lezen of te wijzigen. Deze methoden staan bekend als getters en setters.

3. **Controle en validatie:**
   Inkapseling maakt het mogelijk om controles en validaties uit te voeren wanneer de data wordt gelezen of gewijzigd. Dit zorgt voor een consistente en correcte toestand van het object.

Voordelen van Inkapseling

1. **Bescherming van data:**
   Door de interne details te verbergen, wordt de data beschermd tegen ongeautoriseerde of onbedoelde wijzigingen.

2. **Verbeterde modulariteit:**
   Inkapseling helpt bij het opdelen van een programma in onafhankelijke modules. Elke module (klasse) beheert zijn eigen data en gedrag, wat leidt tot betere onderhoudbaarheid en herbruikbaarheid.

3. **Flexibiliteit en onderhoud:**
   Door de interne implementatie te verbergen, kan deze implementatie in de toekomst worden gewijzigd zonder dat dit invloed heeft op andere delen van het programma die de publieke interface gebruiken.

4. **Toegangscontrole:**
   Inkapseling maakt het mogelijk om de toegang tot de data te controleren en om alleen veilige en geverifieerde wijzigingen toe te staan.

(ChatGPT 3.5 – 03/07/2024)

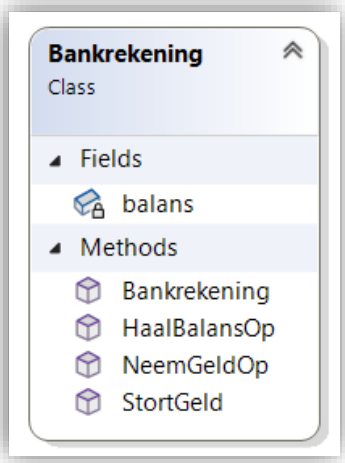
In software systems, **encapsulation** refers to the bundling of data with the mechanisms or methods that operate on the data. It may also refer to the limiting of direct access to some of that data, such as an object's components.[1] Essentially, encapsulation prevents external code from being concerned with the internal workings of an object.

Encapsulation allows developers to present a consistent interface that is independent of its internal implementation. As one example, encapsulation can be used to hide the values or state of a structured data object inside a class. This prevents clients from directly accessing this information in a way that could expose hidden implementation details or violate state invariance maintained by the methods.

(https://en.wikipedia.org/wiki/Encapsulation_(computer_programming))

Voorbeeld - Bankrekening

Laten we dit concept uitleggen aan de hand van een voorbeeld in C#. We maken een eenvoudige Bankrekening klasse die gebruikmaakt van encapsulatie om de balans van een rekening te beschermen.




```
public class Bankrekening
{
    // Private veld om de balans te bewaren
    private decimal balans;
    // Constructor om de bankrekening te initialiseren met een beginbalans
    public Bankrekening(decimal beginBalans)
    {
        if (beginBalans < 0)
        { throw new ArgumentException("Beginbalans mag niet negatief zijn.");}
        balans = beginBalans;
    }
    // Methode om geld te storten
    public void StortGeld(decimal bedrag)
    {
        if (bedrag <= 0)
        { throw new ArgumentException("Stortbedrag moet positief zijn.");}
        balans += bedrag;
    }
    // Methode om geld op te nemen
    public void NeemGeldOp(decimal bedrag)
    {
        if (bedrag <= 0)
        { throw new ArgumentException("Opneembedrag moet positief zijn.");}
        if (bedrag > balans)
        { throw new InvalidOperationException("Onvoldoende saldo."); }
        balans -= bedrag;
    }
    // Methode om de huidige balans op te vragen
    public decimal HaalBalansOp()
    {
        return balans;
    }
}
```

```
Bankrekening mijnRekening = new Bankrekening(1000);

// Geld storten
mijnRekening.StortGeld(200);
Console.WriteLine("Huidige balans na storting: " + mijnRekening.HaalBalansOp());

// Geld opnemen
mijnRekening.NeemGeldOp(150);
Console.WriteLine("Huidige balans na opname: " + mijnRekening.HaalBalansOp());

// Proberen om meer geld op te nemen dan er op de rekening staat
try
{
    mijnRekening.NeemGeldOp(2000);
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Fout: " + e.Message);
}
```

Uitleg

1. **Private Veld:**
   - De balans variabele is privé, wat betekent dat het niet direct toegankelijk is van buiten de Bankrekening klasse. Dit beschermt de interne staat van het object tegen onrechtmatige wijzigingen.
2. **Publieke Methodes:**
   - Er zijn methodes zoals StortGeld, NeemGeldOp, en HaalBalansOp die specifieke manieren bieden om met de balans te werken. Deze methodes controleren of de operaties geldig zijn (bijvoorbeeld of het stortbedrag positief is, of er voldoende saldo is om op te nemen).
3. **Constructor:**
   - De constructor zorgt ervoor dat de beginbalans niet negatief kan zijn bij het aanmaken van een Bankrekening object.

Door gebruik te maken van encapsulatie wordt de integriteit van het Bankrekening object gewaarborgd en wordt de manier waarop de balans wordt gewijzigd gecontroleerd. Hierdoor is het makkelijker om fouten te voorkomen en wordt de code robuuster en beter onderhoudbaar.

(ChatGPT 3.5 – met kleine aanpassingen)

Wat is het verschil tussen abstractie en inkapseling ?

Vergelijking

Abstractie:

- **Doel:** Vereenvoudigt complexiteit door alleen de essentiële kenmerken en gedragingen te benadrukken.
- **Focus:** Wat een object doet.
- **Toepassing:** Wordt bereikt door abstracte klassen, interfaces en algemene klassen en methoden.

Inkapseling:

- **Doel:** Verbergt de interne toestand en beschermt de data-integriteit door gecontroleerde toegang te bieden.
- **Focus:** Hoe de toegang tot de interne toestand wordt beheerd.
- **Toepassing:** Wordt bereikt door toegangsmodificatoren en het gebruik van getters en setters.

Samenvatting

Abstractie en inkapseling zijn beide cruciaal voor het ontwerpen van robuuste, onderhoudbare en flexibele objectgeoriënteerde systemen. Abstractie helpt bij het beheren van complexiteit door de essentiële kenmerken en gedragingen te isoleren, terwijl inkapseling de interne details verbergt en de toegang tot de data controleert om de integriteit van het systeem te waarborgen. Samen dragen deze concepten bij aan de modulariteit, herbruikbaarheid en veiligheid van software.

(ChatGPT 3.5 – 3/7/2024)

Extra info : https://www.guru99.com/nl/difference-between-abstraction-and-encapsulation.html

Overerving

Definitie

Overerving, ook bekend als inheritance, is een fundamenteel concept in objectgeoriënteerde programmering (OOP). Het stelt een nieuwe klasse (subklasse of afgeleide klasse) in staat om eigenschappen (attributen) en methoden van een bestaande klasse (superklasse of basisklasse) over te nemen. Overerving bevordert hergebruik van code, maakt het mogelijk om hiërarchieën van klassen te creëren en ondersteunt polymorfisme.

Wat is Overerving?

Overerving is het mechanisme waarmee een klasse eigenschappen en methoden van een andere klasse erft. Dit betekent dat de subklasse toegang heeft tot de publieke en beschermde leden (attributen en methoden) van de superklasse. Overerving maakt het mogelijk om algemene functionaliteit in een superklasse te definiëren en deze functionaliteit te specialiseren of uit te breiden in subklassen.

Voordelen van Overerving

1. **Herbruikbaarheid van Code:**
   Overerving bevordert het hergebruik van bestaande code. Door algemene functionaliteit in een superklasse te plaatsen, kunnen subklassen deze functionaliteit hergebruiken en uitbreiden.

2. **Modulariteit en Onderhoudbaarheid:**
   Door gedeelde functionaliteit op een centrale plaats (superklasse) te definiëren, wordt het gemakkelijker om wijzigingen door te voeren. Wijzigingen in de superklasse worden automatisch doorgevoerd in alle subklassen.

3. **Hiërarchische Structuur:**
   Overerving maakt het mogelijk om een hiërarchische structuur van klassen te creëren, waarbij meer specifieke klassen (subklassen) worden afgeleid van meer algemene klassen (superklassen).

4. **Polymorfisme:**
   Overerving ondersteunt polymorfisme, waardoor objecten van verschillende subklassen op uniforme wijze kunnen worden behandeld via een gemeenschappelijke superklasse.

(ChatGPT 3.5 – 3/7/2024)

What Is Inheritance?

Inheritance is a fundamental concept in OOPS, i.e., object-oriented programming that allows a subclass to inherit properties and methods from a superclass without re-implementing them. Think of it as a child inheriting genetic traits and skills from their parents.

How Inheritance Simplifies Code

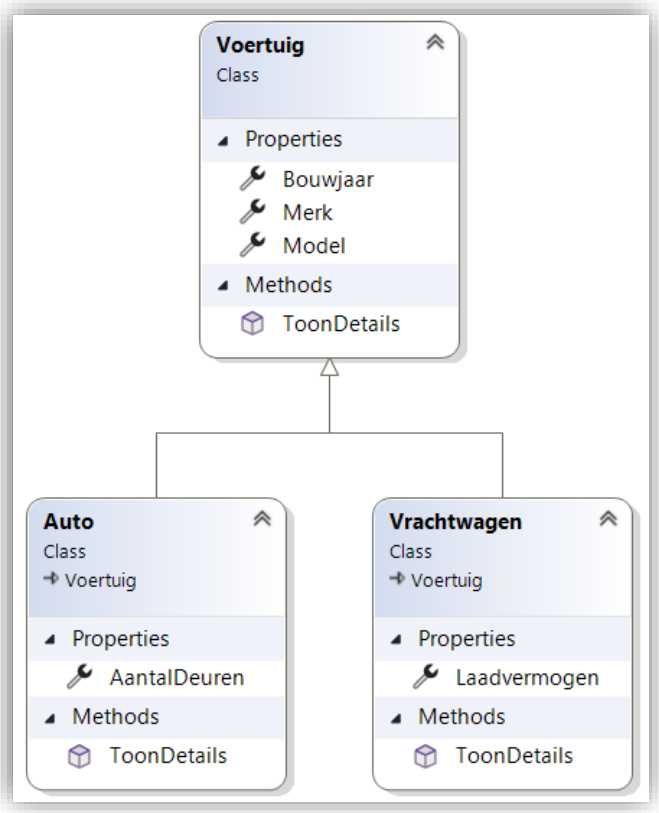
1. **Code Reusability:** You don't have to reinvent the wheel. If you have a class for "Vehicle," you can create subclasses like "Car" and "Bicycle" that inherit the standard features like "move" and "stop." There is no need to rewrite these functions for each type of vehicle.

2. **Modularity:** Inheritance promotes a modular approach. Changes or updates to the base class automatically apply to all its subclasses. For example, enhancing the "Vehicle" class with a "fuelEfficiency" attribute will benefit all its subclasses.

(https://www.almabetter.com/bytes/articles/four-pillars-of-oops)

Voorbeeld - Voertuigen

We beginnen met een basisklasse Voertuig en maken afgeleide klassen zoals Auto en Vrachtwagen.



```
public class Voertuig
{
    // Eigenschappen van de basisklasse
    public string Merk { get; set; }
    public string Model { get; set; }
    public int Bouwjaar { get; set; }

    // Methode van de basisklasse
    public virtual void ToonDetails()
    {
        Console.WriteLine($"Merk: {Merk}, Model: {Model}, Bouwjaar: {Bouwjaar}");
    }
}
```

```
public class Auto : Voertuig
{
    // Extra eigenschap specifiek voor de afgeleide klasse
    public int AantalDeuren { get; set; }

    // Methode van de afgeleide klasse
    public override void ToonDetails()
    {
        base.ToonDetails(); // Aanroep van de basisklasse methode
        Console.WriteLine($"Aantal Deuren: {AantalDeuren}");
    }
}
```

```
public class Vrachtwagen : Voertuig
{
    // Extra eigenschap specifiek voor de afgeleide klasse
    public int Laadvermogen { get; set; }

    // Methode van de afgeleide klasse
    public override void ToonDetails()
    {
        base.ToonDetails(); // Aanroep van de basisklasse methode
        Console.WriteLine($"Laadvermogen: {Laadvermogen} kg");
    }
}
```

Uitleg

1. **Basisklasse Voertuig**: Deze klasse bevat gemeenschappelijke eigenschappen zoals Merk, Model, en Bouwjaar, en een methode ToonDetails om de details weer te geven. De methode ToonDetails is gemarkeerd als virtual zodat deze kan worden overschreven in afgeleide klassen.

2. **Afgeleide klasse Auto**: Deze klasse erft van Voertuig en voegt een extra eigenschap AantalDeuren toe. De methode ToonDetails is overschreven (override) om extra details te tonen.
3. **Afgeleide klasse Vrachtwagen**: Deze klasse erft van Voertuig en voegt een extra eigenschap Laadvermogen toe. De methode ToonDetails is ook overschreven om extra details te tonen.
4. **Gebruik in het programma**: We maken objecten van de klassen Auto en Vrachtwagen, stellen de eigenschappen in, en roepen de methode ToonDetails aan om de details van elk voertuig weer te geven.

Dit voorbeeld toont hoe overerving helpt om gemeenschappelijke functionaliteit in een basisklasse te plaatsen en deze uit te breiden of aan te passen in afgeleide klassen, wat resulteert in een overzichtelijke en onderhoudbare code.

(ChatGPT 3.5 – met kleine aanpassingen)

Polymorfisme

Definitie

Polymorfisme is een fundamenteel concept in objectgeoriënteerde programmering (OOP) dat de mogelijkheid biedt om objecten van verschillende klassen op een uniforme manier te behandelen. Het maakt het mogelijk om één interface te definiëren en om dezelfde interface te gebruiken voor verschillende onderliggende vormen (data types). Polymorfisme bevordert flexibiliteit en herbruikbaarheid van code.

Typen Polymorfisme

1. **Compile-time Polymorfisme (Statisch Polymorfisme):**
   Dit type polymorfisme wordt bepaald tijdens het compileren van de code. Het wordt meestal bereikt door method overloading (meerdere methoden met dezelfde naam maar verschillende parameterlijsten) en operator overloading (het definiëren van nieuwe functionaliteit voor bestaande operatoren).

2. **Run-time Polymorfisme (Dynamisch Polymorfisme):**
   Dit type polymorfisme wordt bepaald tijdens de uitvoering van de code. Het wordt meestal bereikt door method overriding, waarbij een subklasse een methode van de superklasse overschrijft met een nieuwe implementatie. Interfaces spelen ook een grote rol in run-time polymorfisme.

Voordelen van Polymorfisme

1. **Flexibiliteit en Onderhoudbaarheid:**
   Polymorfisme maakt het mogelijk om een algemene interface te definiëren en specifieke implementaties te verschaffen, wat de flexibiliteit en onderhoudbaarheid van de code vergroot.

2. **Herbruikbaarheid van Code:**
   Door polymorfisme kunnen we dezelfde code gebruiken om met verschillende soorten objecten te werken, waardoor codeherhaling wordt verminderd.

3. **Eenvoudiger Onderhoud:**
   Wijzigingen in de superklasse of interfaces vereisen minder wijzigingen in de subklassen, omdat de polymorfe methoden automatisch de nieuwe implementaties volgen.

4. **Dynamisch Gedrag:**
   Polymorfisme maakt het mogelijk om het gedrag van objecten dynamisch te bepalen op basis van hun daadwerkelijke types, wat leidt tot flexibelere en krachtigere softwareontwerpen.

(ChatGPT 3.5 – 3/7/2024)

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this polymorphism occurs, the object's declared type is no longer identical to its run-time type.
- Base classes may define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. In your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

(https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism)

(https://mohamed-hendawy.medium.com/polymorphism-in-object-oriented-programming-with-examples-in-c-7329499cc706)

Voorbeeld - voertuigen

Polymorfisme is een ander fundamenteel concept in object georiënteerd programmeren (OOP), dat de mogelijkheid biedt om dezelfde methode op verschillende manieren te gebruiken, afhankelijk van het object waarop de methode wordt aangeroepen. In C# wordt polymorfisme vaak bereikt door methoden te overschrijven (override) en door het gebruik van basisklasse- en interface-referenties.

We zullen ons vorige voorbeeld uitbreiden om polymorfisme te illustreren.

```
public class Program
{
    public static void Main()
    {
        // Een lijst van voertuigen aanmaken
        List<Voertuig> voertuigen = new List<Voertuig>
        {
            new Auto
            {
                Merk = "Toyota",
                Model = "Corolla",
                Bouwjaar = 2020,
                AantalDeuren = 4
            },
            new Vrachtwagen
            {
                Merk = "Volvo",
                Model = "FH16",
                Bouwjaar = 2018,
                Laadvermogen = 20000
            }
        };

        // Details van elk voertuig weergeven
        foreach (Voertuig voertuig in voertuigen)
        {
            voertuig.ToonDetails();
            Console.WriteLine(); // Voeg een lege regel toe voor leesbaarheid
        }
    }
}
```

Uitleg

1. **Basisklasse Voertuig**: Deze klasse bevat gemeenschappelijke eigenschappen zoals Merk, Model, en Bouwjaar, en een methode ToonDetails om de details weer te geven. Deze methode is gemarkeerd als virtual, zodat deze kan worden overschreven in afgeleide klassen.

2. **Afgeleide klasse Auto**: Deze klasse erft van Voertuig en voegt een extra eigenschap AantalDeuren toe. De methode ToonDetails is overschreven (override) om extra details te tonen.
3. **Afgeleide klasse Vrachtwagen**: Deze klasse erft van Voertuig en voegt een extra eigenschap Laadvermogen toe. De methode ToonDetails is ook overschreven om extra details te tonen.
4. **Gebruik van polymorfisme in het programma**:
    - We maken een lijst van voertuigen (List<Voertuig>) die zowel objecten van de klasse Auto als Vrachtwagen bevat.
    - We itereren over deze lijst en roepen de methode ToonDetails aan voor elk voertuig. Dankzij polymorfisme zal de correcte ToonDetails methode worden aangeroepen, afhankelijk van het daadwerkelijke type object (Auto of Vrachtwagen).

Dit laat zien hoe polymorfisme werkt: we gebruiken een basisklasse-referentie (Voertuig) om methoden aan te roepen die zijn overschreven in afgeleide klassen (Auto en Vrachtwagen). Hierdoor kan dezelfde methode-oproep (ToonDetails) verschillende uitvoeringen hebben, afhankelijk van het objecttype waarop de methode wordt aangeroepen.

(ChatGPT 3.5 – met kleine aanpassingen)

Samenvatting

	DESCRIPTION	BENIFITS
ABSTRACTION	• Focuses on creating simplified models of complex systems by hiding unnecessary details. • Creates abstract classes and interfaces defining common characteristics and behaviors.	• Code modularity • Separates interface from implementation • Promotes code flexibility and maintainability
INHERITANCE	• Allows creation of new classes based on existing classes. • Derived classes inherit properties and behaviors of the superclass.	• Code reuse • Supports hierarchy • Promotes code extensibility
POLIMORPHISM	• Enables objects of different classes to be treated as objects of a common superclass. • Same method can have different implementations based on object type.	• Code flexibility • Dynamic method binding • Enhances code reusability
ENCAPSULATION	• Combines data and methods into a single unit called a class. • Hides internal details of an object and provides controlled access to properties.	• Data privacy • Code organization • Modularity • Data integrity • Protection against unauthorized access

https://medium.com/@kalanamalshan98/oop-concepts-mastering-basics-with-real-life-examples-for-easy-understanding-part-1-da5b8fc21036