**Interfaces (Schnittstellen)**

**Interfaces** kann man als **Baupläne** für die **Teil- oder Ganzstruktur** einer Klasse bezeichnen.

public class VipReservation: Reservation, IWithSurcharge, ISpecialServices

{

public bool HasWelcomeDrink { get; private set; } = true;

public decimal? Surcharge { get; set; }

public IEnumerable<string>? SpecialServices { get; set; }

}

**Was passiert hier?**

* **VipReservation** ist eine **spezialisierte Version** von **Reservation** (VipReservation **leitet** von der Basisklasse **Reservation** ab, sie **erbt** davon.).
* Sie **implementiert** aber auchzwei Interfaces:
  + **IWithSurcharge** → definiert, dass ein Zuschlag berechnet werden kann.
  + **ISpecialServices** → definiert, dass spezielle Zusatzleistungen möglich sind.

Dadurch erhalten alle Klassen, die diese Interfaces implementieren, die Möglichkeit, **Zuschläge** und/oder **Zusatzleistungen** anzubieten – unabhängig davon, wie sie intern aufgebaut sind.

**Vorteile von Interfaces**

* **Lose Kopplung:** Interfaces ermöglichen die Erweiterung von Klassen **ohne Abhängigkeit vom konkreten Typ.**
* **Modularität:** Du kannst Features wie **Zusatzleistungen**, **Zuschläge**, **Rabatte**, **Service-Level** usw. als Interfaces definieren und bei Bedarf kombinieren.

|  |  |
| --- | --- |
| Konzept | Beispiel |
| Mehrfach-Interface-Implementierung | IWithSurcharge, ISpecialServices |
| Property-basierte Interfaces | keine Methoden, nur get/set-Properties |
| Erweiterbarkeit ohne Vererbung | andere Klassen können die Interfaces auch nutzen |

**Zusammengefasst**

VipReservation wird durch diese Interfaces **flexibler** und **klarer strukturiert**: sie **"verpflichtet"** sich, **Zuschläge** und **Extraservices** bereitzustellen, **ohne** dass diese Logik in der Basisklasse Reservation verankert sein muss.

**Warum sollte ich statt konkreter Klasse ein Interface als Variablen-Typ verwenden?**

Wenn Sie statt einer konkreten Klasse ein **Interface** als Variablen-Typ verwenden, bekommen Sie mehrere Vorteile:

**1. Lose Kopplung**

* **Problem**: Wenn der Variablen-Typ eine konkrete Klasse ist, hängt Ihr Code direkt von dieser Klasse ab.
* **Lösung**: Mit einem Interface hängt Ihr Code nur von der „Fähigkeit“ ab, nicht von der konkreten Umsetzung.  
  Dadurch können Sie Implementierungen leicht austauschen.

// Abhängigkeit von einer konkreten Klasse

PdfExporter exporter = new PdfExporter();

// Abhängigkeit von einem Interface

IExporter exporter = new PdfExporter();

**2. Austauschbarkeit & Erweiterbarkeit**

* Wenn Sie morgen statt PdfExporter einen WordExporter brauchen, müssen Sie bei der Interface-Variante **nur** die Instanziierung ändern, nicht den restlichen Code.
* Der Code, der exporter.Export(data) aufruft, bleibt identisch, solange beide Klassen IExporter implementieren.

**3. Testbarkeit (Unit-Tests, Mocking)**

* Wenn Sie nur ein Interface erwarten, können Sie in Tests eine **Fake- oder Mock-Implementierung** einschleusen.
* Das wäre nicht möglich, wenn der Code immer eine konkrete Klasse erzwingen würde.

public class ReportService

{

private readonly IExporter exporter;

public ReportService(IExporter exporter)

{

this.exporter = exporter;

}

public void CreateReport(string data)

{

exporter.Export(data);

}

}

// Im Test

IExporter fakeExporter = new FakeExporter();

var service = new ReportService(fakeExporter);

**4. SOLID-Prinzipien (insbesondere DIP)**

* Das **Dependency Inversion Principle** sagt: „High-Level-Module sollten nicht von Low-Level-Modulen abhängen, sondern beide von Abstraktionen.“
* Interfaces sind genau diese Abstraktionen.  
  Sie helfen, großen Projekten eine flexible Architektur zu geben.

**5. Flexiblere APIs**

* Eine Methode, die ein Interface als Parameter annimmt, ist sofort universeller einsetzbar.

void PrintAll(IEnumerable<string> daten)

{

foreach (var d in daten)

Console.WriteLine(d);

}

* Hier können Sie List<string>, string[], HashSet<string> usw. übergeben, weil sie alle IEnumerable<string> implementieren.