**Portfolio Modul 323**

M323 - Funktionale Programmierung mit Angular

Portfolio von jeremias Rieser



**Inhaltsverzeichnis**

**M323 - Funktionale Programmierung**

[1. Einführung 1](#_Toc153835692)

[2. Erwartungen und Standortbestimmung 1](#_Toc153835693)

[3. Wichtige Code-Anweisungen 1](#_Toc153835694)

[3.1 TypeScript Basics 1](#_Toc153835695)

[3.3 Angular 4](#_Toc153835696)

[4. Coding Conventions 11](#_Toc153835697)

[5. Reflexionen 11](#_Toc153835698)

[5.1 Reflexion zum Modul 11](#_Toc153835699)

[5.1.1 Vergleich zur Standortbestimmung 11](#_Toc153835700)

[5.1.2 Learnings aus dem Projekt 11](#_Toc153835701)

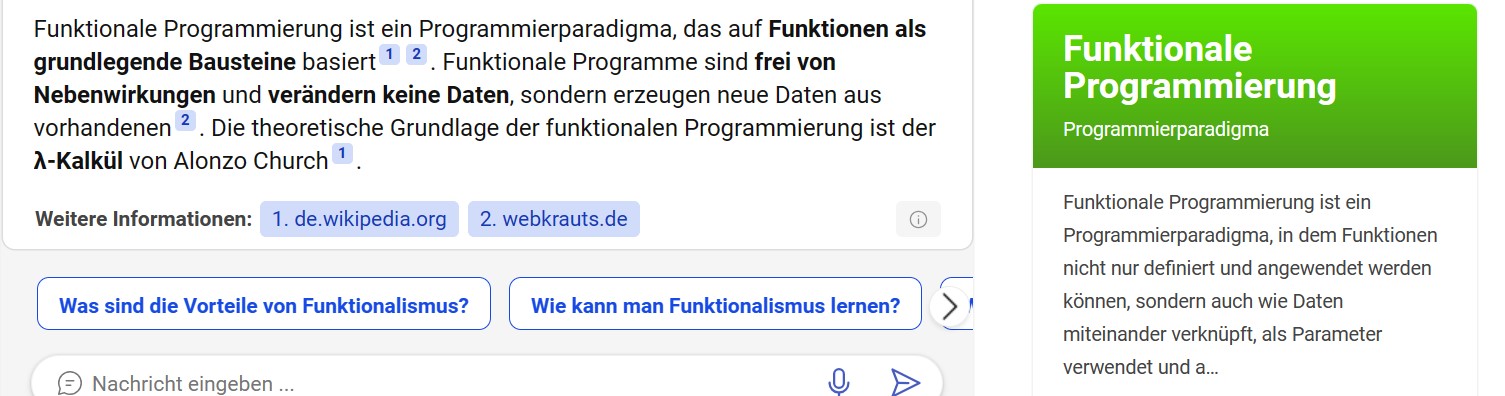
[6. Glossar 11](#_Toc153835702)

# 1. Einführung

Das Portfolio hält den Wissensfortschritt an den Eckpunkten das Moduls 323 fest und soll Ihnen helfen, Ihren Wissensfortschritt sichtbar zu machen. Zusätzlich dient dieses als Grundlage zur Reflexion sowie als Sprachspeicher, welchen Sie für die weiteren Module wiederverwenden können.

# 2. Erwartungen und Standortbestimmung

Was verstehe ich darunter?



# 3. Wichtige Code-Anweisungen

## 3.1 TypeScript Basics

Oft bringt TypeScript Features, die bei JS nachgepatched werden:

Data Types: string, number, boolean, void, null, undefined, any Gelernte Elemente (Vor Prüfung erweitern): Type Annotations: let name:typ; object Type: object alles ausser primitive Datentype wie string, number etc.

Array: let skills: string[]

Tuple: let skills: [string, number]; -> skills = [«esse», 5]

Any: Alles. Wenn nichts (kein Datentyp) geschrieben, dann normal any. Man kann auch Sachen aufrufen, die nicht existieren anyItem.eatSomething(); (kein Fehler, aber macht eben nichts)

Vererbung: constructor(eat:number){/\*call parent constructor\*/ super(eater)} (generell super.etwas holt aus der Vaterklasse) Interfaces:

interface HistoryEntry {a:number, b:number, c:number} let calcHistory: HistoryEntry[] = []; interface StringFormat {

(str: string, isUpper: boolean): string

}

let format: StringFormat;

format = function (str: string, isUpper: boolean) {

return isUpper ? str.toLocaleUpperCase() : str.toLocaleLowerCase();

};

Union-type: let a: string|number

Ein Union-Typ in TypeScript ermöglicht es, eine Variable, einen Parameter oder einen Rückgabewert so zu definieren, dass er mehrere verschiedene Typen annehmen kann. Das bedeutet, dass die Variable, der Parameter oder der Rückgabewert eines solchen Union-Typs eines von mehreren verschiedenen Typen sein kann.

Mehrmals nachgeschlagen: interfaces, Class und Tuple

DuckTyping: Interface nicht zwingend, wenn so oder so Eigenschaften in Klassen bereits existieren (oder so was in der Art)

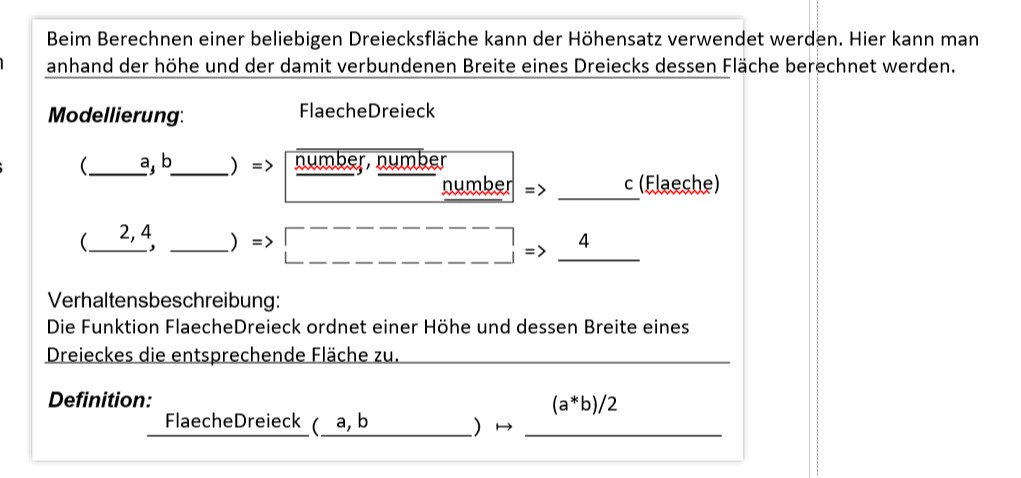
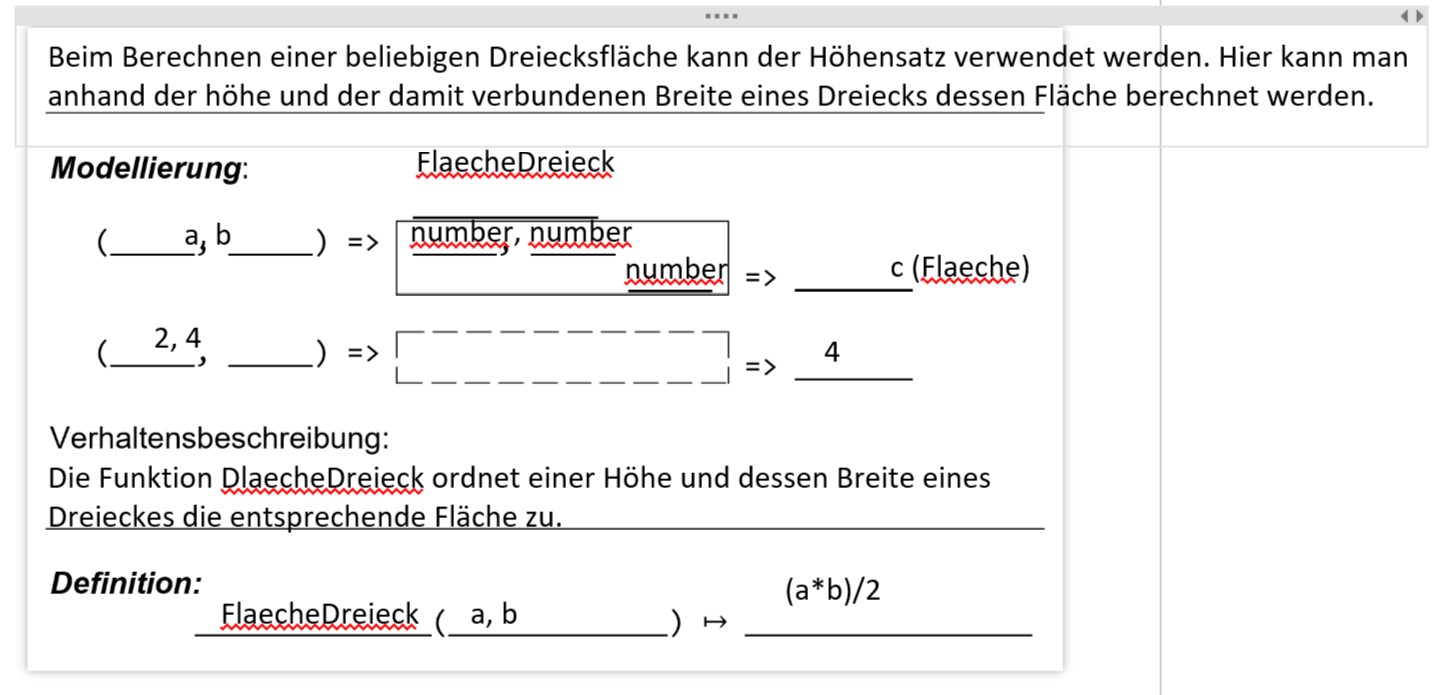
**3.2 Funktionale Programmierung**

Strukturiert: Kontrollstrukturen viel genutzt: if/else, Schleifen etc.

Prozedur: Durch Funktionen. Diese können dann von irgendwelchen Prozessen aufgerufen werden

OO: Halt Objektorientiert: Blueprints und dann Realisierung. Abstraktion der Realität

Deklaratives/Funktionales Programmieren benutzt keine Variablen (keine Seiteneffekte). Kaum Kontrollstrukturen (dadurch rekursiv). Möglichst viele Teilfunktionen für Wiederverwendung.



Beispiel Modellierung für Fläche berechnen

Vorgehensweise funktionale Programmierung:

1. Problem identifizieren
2. Aufteilung des Problems in Unterfunktionen
   1. Möglichst granular (genau) unterteilen

Relation zwischen den Funktionen herstellen

* 1. Statt statischer Formeln werden jetzt dynamische Zuordnungen (auch zwischen den Funktionen) benutzt.

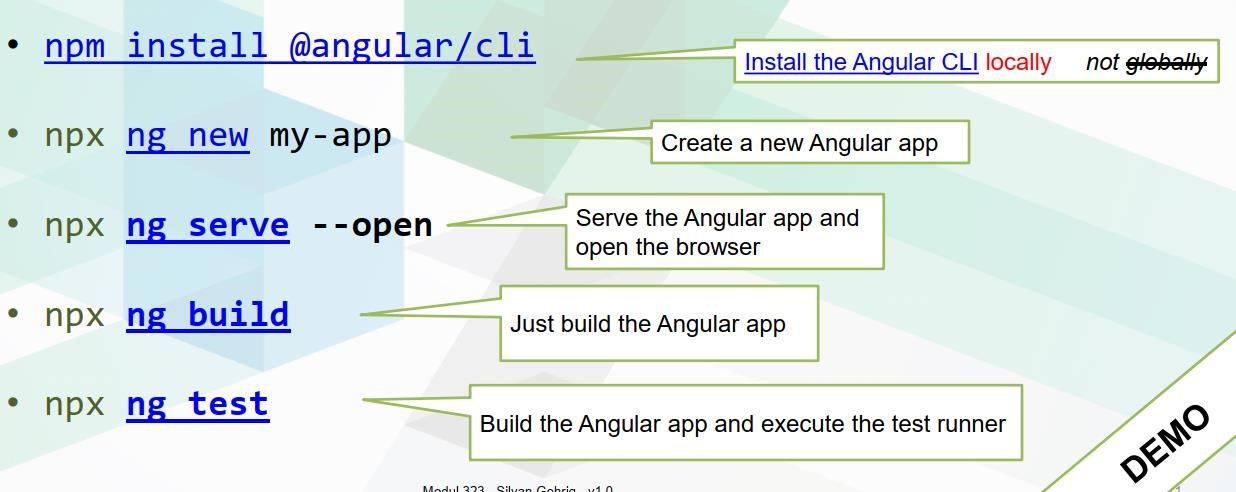
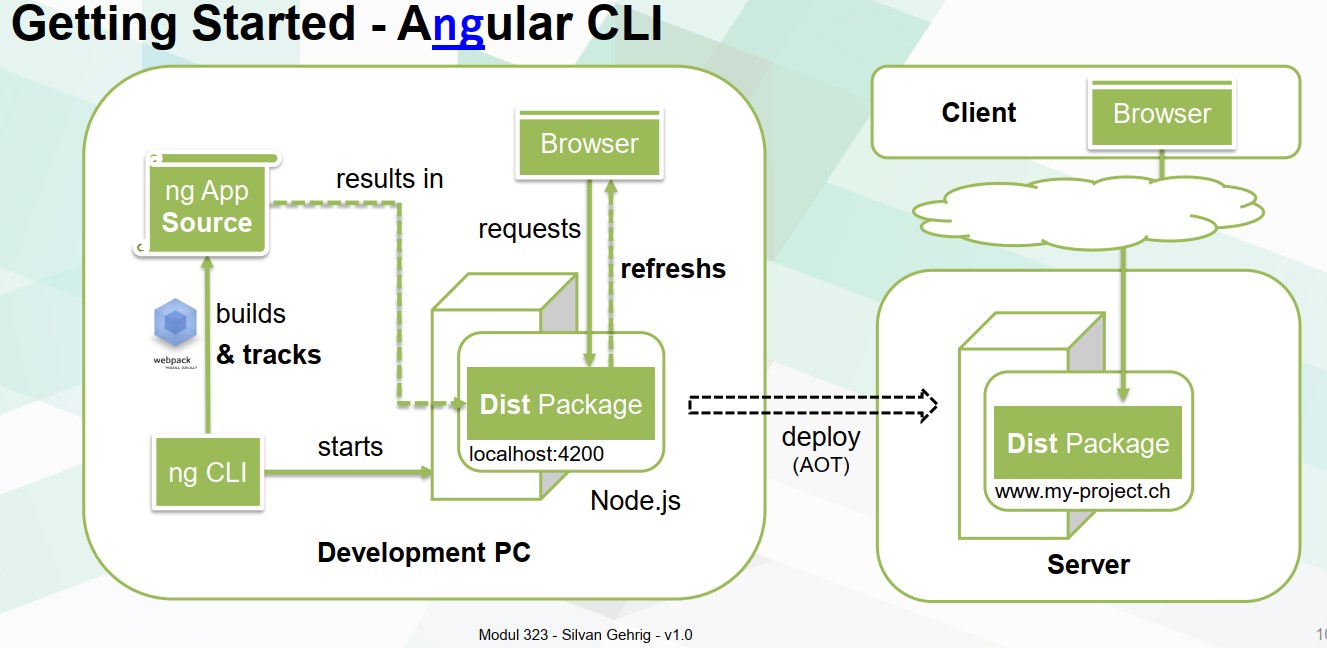
1. Funktionen mit Ein- und Ausgabe realisieren

Funktionale Aufteilung (decomposition) weiter verfeinern

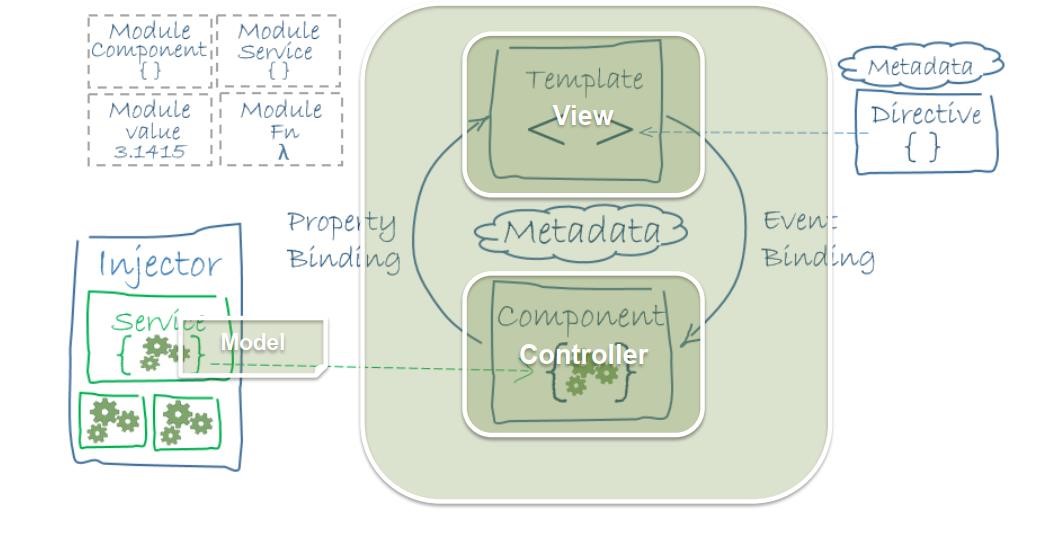
## 3.3 Angular

Command-Line Interface:

Hilft dabei, Angular Projekte konsistent und sauber zu errichten und zu erweitern. (MEHR HINSCHREIBEN)



Funktional + OO



Registration -> Request -> Resolve -> Fulfill Komponente:

Bestehen aus View (HTML), SCSS (Style) und Logik (TS)

Eine Komponente ist ein Teil der ganzen Applikation. Damit wird eine Trennung der App in einzelne Teile ermöglicht.

Welche Bindings: One-way in: [] oder {{}}

One-way call: (eventBindings)

* View -> Component: (click)=»clickHanlder($event)»
* Component -> View: {{…}} or […]

Two-way: [(ngModel)]="ainput"

Das mit Input und Output -> UNBEDINGT ERNEUT ANSCHAUEN

**@Input & @Output:**

**In TypeScript/Angular:** @Input und @Output sind Dekoratoren, die in Angular verwendet werden, um Daten zwischen Komponenten zu übergeben. @Input wird verwendet, um Daten in eine Komponente zu übergeben, während @Output verwendet wird, um Ereignisse aus einer Komponente zu emittieren.

# 4. Coding Conventions

Die Coding Conventions von Angular mit TypeScript wurden von Google (sowie namhaften [Autoren wie John Papa)](https://github.com/johnpapa/angular-styleguide/blob/master/a2/README.md) im [Angular coding style guide](https://angular.io/guide/styleguide) unter [angular.io](https://angular.io/guide/styleguide) beschrieben.

Weiterhin definiert Microsoft [Do’s and Don’ts,](https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html) welche beim Programmieren von TypeScript beachtet werden sollen. Diese sind auf [typescriptlang.org](https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html) verfügbar.

# 5. Reflexionen

In den Reflexionsaufgaben beurteilen Sie Ihre Arbeit und Blicken Sie nochmals auf das Modul zurück. Die Reflexionsfragen finden Sie in ausgewählten Aufträgen.

## 5.1 Reflexion zum Modul

Ja, ich bin mit dem Projekt zufrieden.

Frage 1: Ich konnte mich in Angular und Typescript verbessern. Ausserdem wurde ich mit der Funktionalen Programmierung sicherer.

Frage 2: Mein Programm basiert auf sehr vielen Funktionen, welche gemeinsam interagieren. Allerdings hätte ich die Funktionen besser aufteilen und nicht so viele in einen Component schreiben sollen.

Frage 3: Die Funktionale Programmierung ist auch in vielen anderen Programmiersprachen hilfreich und funktionieren ähnlich. Ich bin besonders stolz auf den Materialtable im Projekt. Die loadTrasaction Funktion funktioniert sehr gut.

### 5.1.1 Vergleich zur Standortbestimmung

a) Sehr realistisch. Ich konnte mit einigen Zusatzaufgaben beginnen. Es gibt nicht wirklich etwas, was ich noch vermisse.

b) Die Learning Group hat mir eher weniger geholfen. Ein gewisses Teammitglied hatte bis zum letzten Termin noch fast nicht begonnen. Meine Ziele konnte ich noch nicht ganz erreichen. Ich möchte das Projekt in Zukunft noch beenden, um es auf GitHub zu stellen.

c) Etwas mehr selbstständig zu arbeiten. Ich musste oft andere Fragen, um an schwierigen Stellen weiterzukommen. Ausserdem sollte ich in Zukunft die Theorie besser verstehen. Den Code sollte ich auch übersichtlicher gestalten.

### 5.1.2 Learnings aus dem Projekt

Tag 1:

[ Beantworten Sie die Fragen im Projektbeschrieb (Programm). ] Tag 2:

# 6.Projektdokumentation

## 6.1 Angular Services und Immutable state

2.1 a) Ich habe den localstorageservice erstellt. Er vereinfacht die Verwendung von dem jwt-Token, welches für faste jede Funktion gebraucht wurde, um den Account zu identifizieren. Er besitzt die Funktion load und save, wobei save nur beim Registrieren wirklich wichtig war. Um den Service in anderen Komponenten einzufügen, muss er allerdings jeweils zuerst in das File importiert werden. Dazu sollte man den Service mit einem Namen im Constructor mitgeben. Danach sollte man ihn verwenden können.

b) Services an sich zu nutzen ist einer der best practices, wenn man mit angular arbeitet, da man dadurch weniger Code in einem einzelnen File hat. Man kann in services ausserdem hot-observables einbauen, was bei meinem Service allerdings nicht der Fall ist.

2.2 a) Bei der Immutable state geht es um den Status einer Variable. Bei diesem Status kann die Variable nicht verändert werden. Dazu kann «const x = readonly<>» verwendet werden. Danach kann die Variable zwar gelesen, aber nicht bearbeitet werden. Der erste Vorteil davon ist, dass durch diese state Fehler im code verhindert werden können. Der zweite ist, dass man sich sicher sein kann, mit welchem Inhalt der Variabel man gerade arbeitet. Dass kann das Coden leichter machen. Ich habe keine Immutable states in meinem Code verwendet. In Angular ist das Arbeiten mit immutable state bevorzugt. Andere Frontends sind eher auf andere states ausgelegt.

b) Manchmal ist es dringend nötig, dass man keine Immutable state verwendet, da es sonst zu sehr viel Code führen würde.

## 6.2 Lamdas und currying

3.0 a) Jeder Parameter gibt eine closure Funktion zurück. Das bedeutet es gibt mehrere return werte, welche verwendet werden können.

b) Es ist insgesamt mehr Code, als wenn man die Funktion ohne currying schreiben würde.

c) Ich habe keine Curry Funktion verwendet.

# 7. Glossar

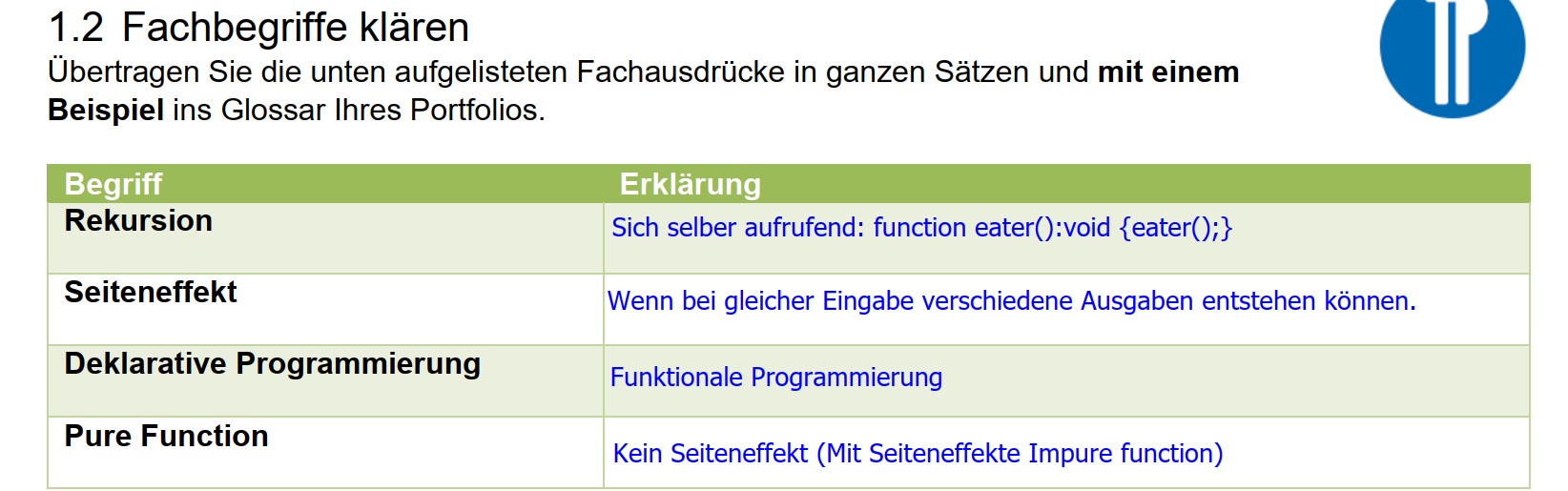
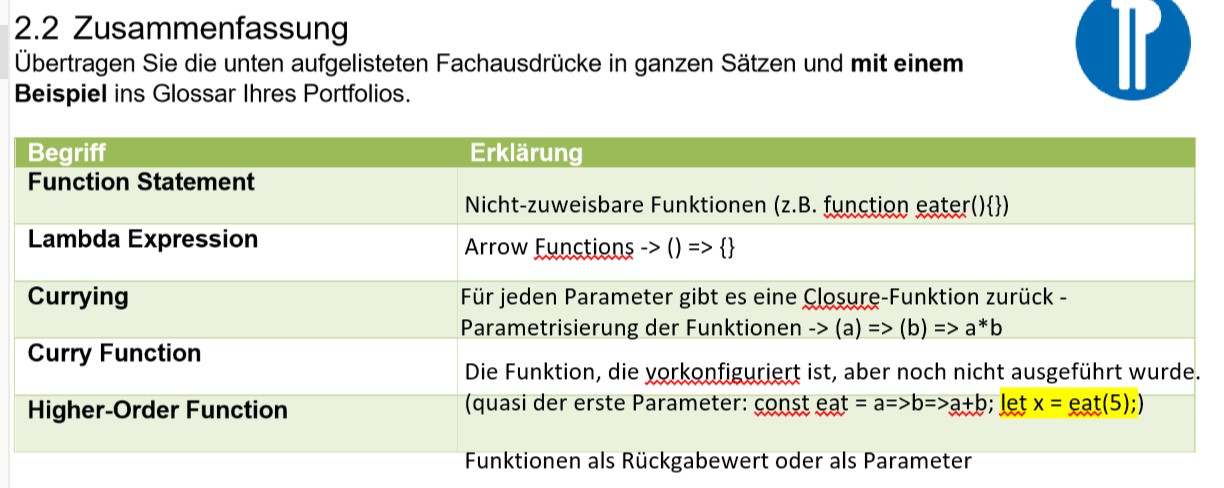
Sie werden Ihr Glossar im Verlauf dieses Moduls laufend ergänzen. Das Glossar dient als Sprachspeicher, damit Sie den effektiven und präzisen Umgang mit der Fachsprache zur Funktionalen Programmierung erlernen.

**Wichtig**: Über das gesamte Modul 323 werden Sie Fachbegriffe zum Thema funktionale Programmierung erlernen und hier im Glossar definieren. **Arbeiten sie regelmässig an Ihrem Glossar**.

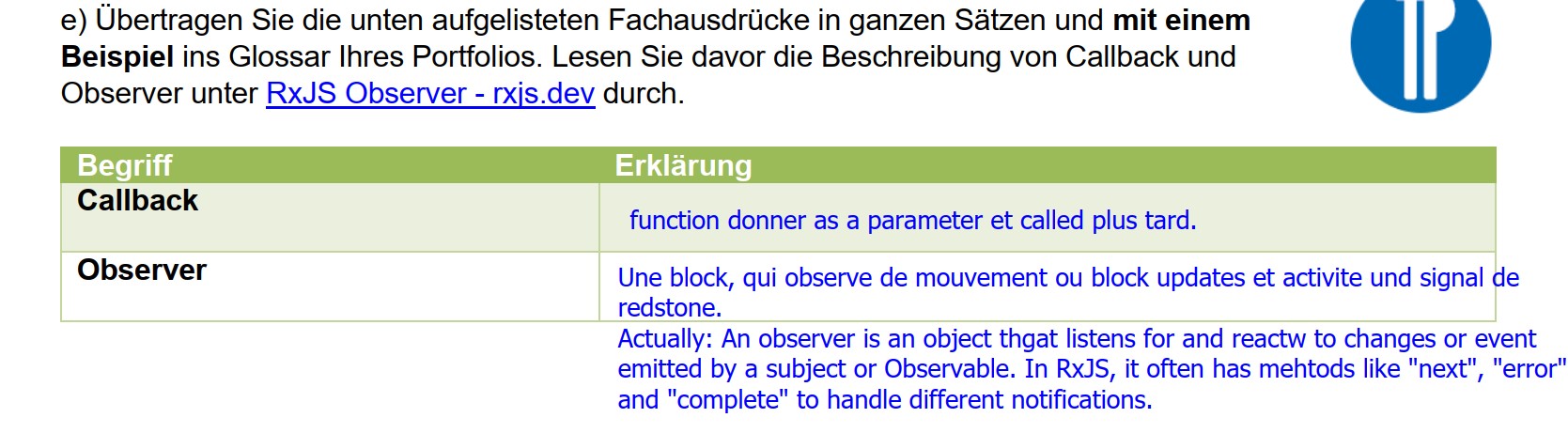
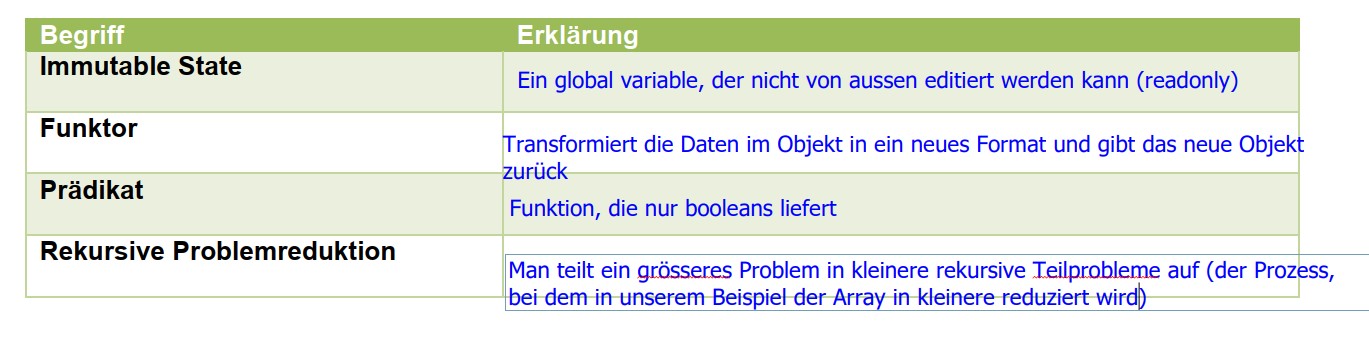
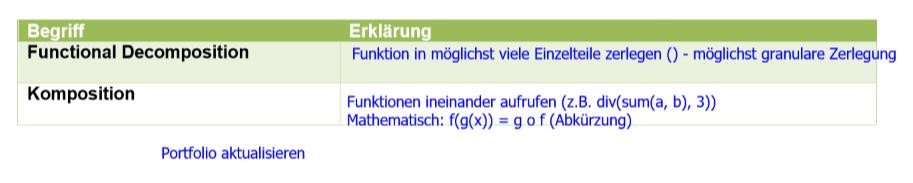
Beschreiben Sie die Fachausdrücke in ganzen Sätzen mit mindestens 100 Buchstaben (inklusive Leerzeichen) pro Ausdruck. Verwenden Sie, wenn immer möglich, Code-Beispiele für Ihre Erklärungen.

**Begriff Erklärung**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |



Alle Curries High Order functions



-----------

Flüchtigkeitsfehler lassen sich dank Type Annotations reduzieren.

Der Typescript Compiler erkennt nicht vorhandene Bezeichner.

Der Datentyp Tuple ist eine Spezialisierung von Arrays.

Ein nachgestelltes ? zeigt eine optionale Type Parameter an.

Der Datentyp any unterdrückt die Typenüberprüfung

In einer Rekursion ruft eine Funktion sich selbst auf.

Unterschiedliches Verhalten bei selber Parametrisierung nennt man Seiteneffekt.

Mittels Komposition werden Unterfunktionen in Funktionen aufgerufen.

Eigenschaften von Klassen können mit Typ Annotations versehen werden.

In der Konstruktor-Definition lassen sich Eigenschaften definieren.

Mittels Type Inference nähert der Compiler Typen an.

Innere Funktions können auf Variablen der äusseren Functions zugreifen.

Durch die Closure wird indirekt Zugriff auf den Variablenscope gewährt.

Mit Closures lässt sich das Information Hiding ohne Klassen umsetzen.

Curry beschreibt eine partiell parametrische Funktion.

Durch Currying können Funktionen flexibler eingesetzt werden.

Bindings erlauben die Konfiguration von Komponenten sowie HTML-Elementen.

Komponenten gliedern sich in View, Logik und Styles.

Angular basiert auf dem MVC-Pattern.

Hot Observables feuern garantiert mehrere Events.



Hot Observables haben ein Stream-artiges Verhalten.

Cold Observables werden nach dem ersten Event abgeschlossen.

**Basics:**

* TypeScript ist eine typifizierte Version von JavaScript – Bietet erweiterte Syntax für die Typisierung von • Variablen, Parameter und Rückgabewerte, Eigenschaften (Properties), Schnittstellen (Interfaces),
* TypeScript ist ein Superset von JavaScript (siehe Bild rechts) – Allerdings in der Ausführung rückwärtskompatibel • Es erfolgt also ein Type Erasure, ähnlich wie bei Java
* TypeScript erweitert JavaScript zu einer vollen objektorientierten Sprache – tsc-Compiler «transpiled» TypeScript zu JavaScript –
* TypeScript hat keine eigene Runtime (ausser ts-node, welcher per se eine Art TypeScript «precompiler» mitliefert)
* Der tsc-Compiler lässt sich stark parametrisieren:

A yellow square with white text

Description automatically generated

* Type-Fehler werden bereits zur Compile Time angezeigt. –

Flüchtigkeitsfehler werden so schon vor dem Ausführen sichtbar und korrigierbar. – Ohne klare Typifizierung ist funktionale Programmierung sehr schwierig umsetzbar.

* Nicht auffindbare Bezeichner (von Variablen, Funktionen, Klassen, …) führen ebenfalls zu Compile-Time Fehlern. • Häufig basieren neue Features des JavaScript Standards (ECMAScript-262, TC39 Gremium) auf den TypeScript Features. TypeScript brings the future JavaScript to today.

**Immutable State:**

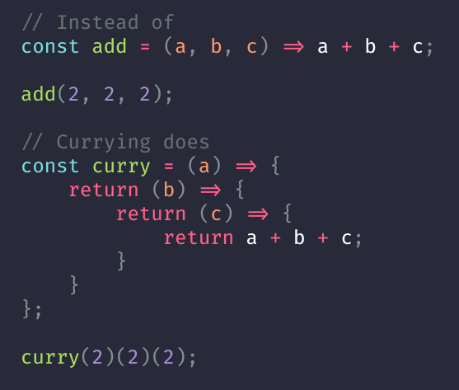
const readonly

**Callback:**

Ein Callback ist eine Funktion, die an eine andere Funktion übergeben wird und später von dieser anderen Funktion aufgerufen wird. Callbacks werden häufig in asynchronem Code verwendet, um mit dem Abschluss von Operationen zu interagieren, z. B. beim Laden von Daten.

**Currying:**

Currying ist eine Technik, eine Funktion mit mehreren Argumenten in eine Reihe von Funktionen aufzuteilen, die jeweils ein einzelnes Argument entgegennehmen. TypeScript unterstützt Currying durch Funktionsüberladungen und Pfeilfunktionen.



**Bindings:**

Bindungen in Angular beziehen sich darauf, wie Daten zwischen der Komponente und dem Template fließen. Bindungen können Property-Bindings für das Aktualisieren von Eigenschaften und Event-Bindings für das Reagieren auf Ereignisse umfassen.

**Event:**

**In TypeScript/Angular:** Ein Ereignis in Angular bezieht sich auf Benutzeraktionen oder Systemereignisse, auf die die Anwendung reagieren kann. In der Regel werden Ereignisse in Angular-Komponenten durch Event-Bindings behandelt, um auf Benutzerinteraktionen zu reagieren.

**MVC-Pattern:**

Das MVC (Model-View-Controller)-Muster ist ein Designmuster zur Strukturierung von Anwendungen. In Angular übernimmt der Komponenten-Controller die Rolle des Controllers, das Template repräsentiert die View, und Services oder Modelle repräsentieren das Modell.

**Annotations:**

Type Annotation Syntax

Typen werden immer nach einem : geschrieben:

const myVariable: string = "hello World";

function myFunction(param1: string, param2: string) : number /\* Typisierung des Rückgabewerts \*/ {

return 42;

}

TypeScript setzt auf die «Type Annotations» von UML

**Tuple:**

Der Datentyp Tuple ist eine Spezialisierung von Arrays.

A black background with blue text

Description automatically generated

**@Input & @Output:**

**In TypeScript/Angular:** @Input und @Output sind Dekoratoren, die in Angular verwendet werden, um Daten zwischen Komponenten zu übergeben. @Input wird verwendet, um Daten in eine Komponente zu übergeben, während @Output verwendet wird, um Ereignisse aus einer Komponente zu emittieren.

**Parametisierung:**

In TypeScript/Angular: Parametisierung bezieht sich auf die Übergabe von Parametern an Funktionen oder Methoden. In Angular kann dies die Übergabe von Parametern zwischen Komponenten oder Diensten bedeuten.

Bitte lassen Sie mich wissen, wenn Sie weitere Fragen zu bestimmten Begriffen haben oder weitere Erläuterungen wünschen.

**Closure:**

**In TypeScript/Angular:** Eine Closure tritt auf, wenn eine Funktion auf Variablen außerhalb ihres eigenen Gültigkeitsbereichs zugreifen kann, auch wenn die äußere Funktion bereits abgeschlossen ist.

function outerFunction() {

let outerVariable = 10;

function innerFunction() {

// Die innere Funktion hat Zugriff auf die äußere Variable

console.log(outerVariable);

}

// Rückgabe der inneren Funktion, die eine Closure bildet

return innerFunction;

}

**Variablesscope:**

**In TypeScript/Angular:** Der Variablenbereich bezieht sich auf den Gültigkeitsbereich einer Variable, also den Teil des Codes, in dem die Variable sichtbar und ansprechbar ist. In TypeScript gibt es den globalen Scope, Funktionsscope und Blockscope.

**Lambda:**

**In TypeScript/Angular:** Lambda-Funktionen, auch als Pfeilfunktionen bekannt, sind kürzere Schreibweisen für anonyme Funktionen. Sie werden häufig in TypeScript und angular verwendet, um Funktionen kompakter zu definieren, insbesondere bei Funktionen mit nur einer Anweisung.

export *const* multiply = (*a*:*number*, *b*:*number*): *number* *=>* *a* \* *b*;

export *const* divide = *function* divide(*a*:*number*, *b*:*number*): *number* {

    return (*b* == 0) ? -1 : *a* / *b*;

}

**Type Inference:**

**In TypeScript/Angular:** Die Typinferenz ist die Fähigkeit des Typsystems, den Typ einer Variable automatisch zu erkennen, basierend auf ihrem Wert. TypeScript bietet starke Typisierung und kann oft den Typ ableiten, ohne dass er explizit angegeben werden muss.

**Rekursion:**

**In TypeScript/Angular:** Rekursion bezieht sich auf die Technik, eine Funktion aufzurufen, die sich selbst aufruft. In TypeScript können rekursive Funktionen verwendet werden, um Probleme durch wiederholte Teilproblemlösungen zu lösen.

*const* fakultaet = (*a*: *number*): *number* *=>* {

   if(*a* === 1){

     return 1;

   }

   return multiply(*a*, fakultaet(*a* - 1));

}

**Seiteneffekt:**

**In TypeScript/Angular:** Ein Seiteneffekt tritt auf, wenn eine Funktion zusätzlich zu ihrer Rückgabewertänderung etwas außerhalb der Funktion bewirkt, wie das Ändern von Variablen außerhalb ihres Gültigkeitsbereichs. In funktionaler Programmierung werden Seiteneffekte oft vermieden.

**Deklarative/Funktionale Programmierung:**

**In TypeScript/Angular:** Deklarative Programmierung konzentriert sich darauf, "was" erreicht werden soll, anstatt "wie" es erreicht werden soll. Funktionale Programmierung ist ein Paradigma, das sich auf die Verwendung von Funktionen als grundlegende Bausteine konzentriert.

**Imperative Programmierung:**

**In TypeScript/Angular:** Im Gegensatz zur deklarativen Programmierung konzentriert sich die imperative Programmierung darauf, Schritte und Anweisungen festzulegen, wie eine Aufgabe erreicht werden soll.

**Typisierung:**

**In TypeScript/Angular:** Die Typisierung bezieht sich darauf, dass Variablen, Parameter und Rückgabewerte in TypeScript mit einem bestimmten Datentyp versehen sind. Dies hilft, Fehler frühzeitig zu erkennen und die Codequalität zu verbessern. :string, :number, :bools

A screenshot of a computer

Description automatically generated

**Best Practices:**

Best Practices in Angular helfen dabei, qualitativ hochwertigen und wartbaren Code zu schreiben.

/\*\*/

**In TypeScript/Angular:** "Best Practices" sind bewährte Methoden und Herangehensweisen, um effizienten, wartbaren und fehlerfreien Code zu schreiben. In Angular gibt es viele Best Practices, einschließlich der Verwendung von Modulen, Services, asynchronem Programmieren, und anderen.

**Function stream:**

/\*\*/

**In TypeScript/Angular:** "Function Stream" ist keine gängige Bezeichnung im Angular-Kontext. Es könnte sich auf die Verwendung von Observables beziehen, wenn Funktionen (Methoden) als Streams von Ereignissen betrachtet werden. Diese Streams können dann in der Anwendung auf verschiedene Weisen verarbeitet werden.

**Hot Observables:**

**In TypeScript/Angular:** Ein "Hot Observable" ist ein Observable, das unabhängig von den Abonnenten Daten produziert, sobald es erstellt wird. Alle Abonnenten teilen sich denselben Datenstrom. In Angular können Services oft Hot Observables bereitstellen, die Daten für verschiedene Komponenten bereitstellen.

**Cold Observables:**

**In TypeScript/Angular:** Ein "Cold Observable" ist ein Observable, das erst dann Daten produziert, wenn jemand sich dafür interessiert (es abonniert). Jeder Abonnent erhält seinen eigenen Datenstrom. Ein Beispiel dafür ist eine HTTP-Anforderung in Angular, die eine Cold Observable erstellt.

A close up of a sign

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated