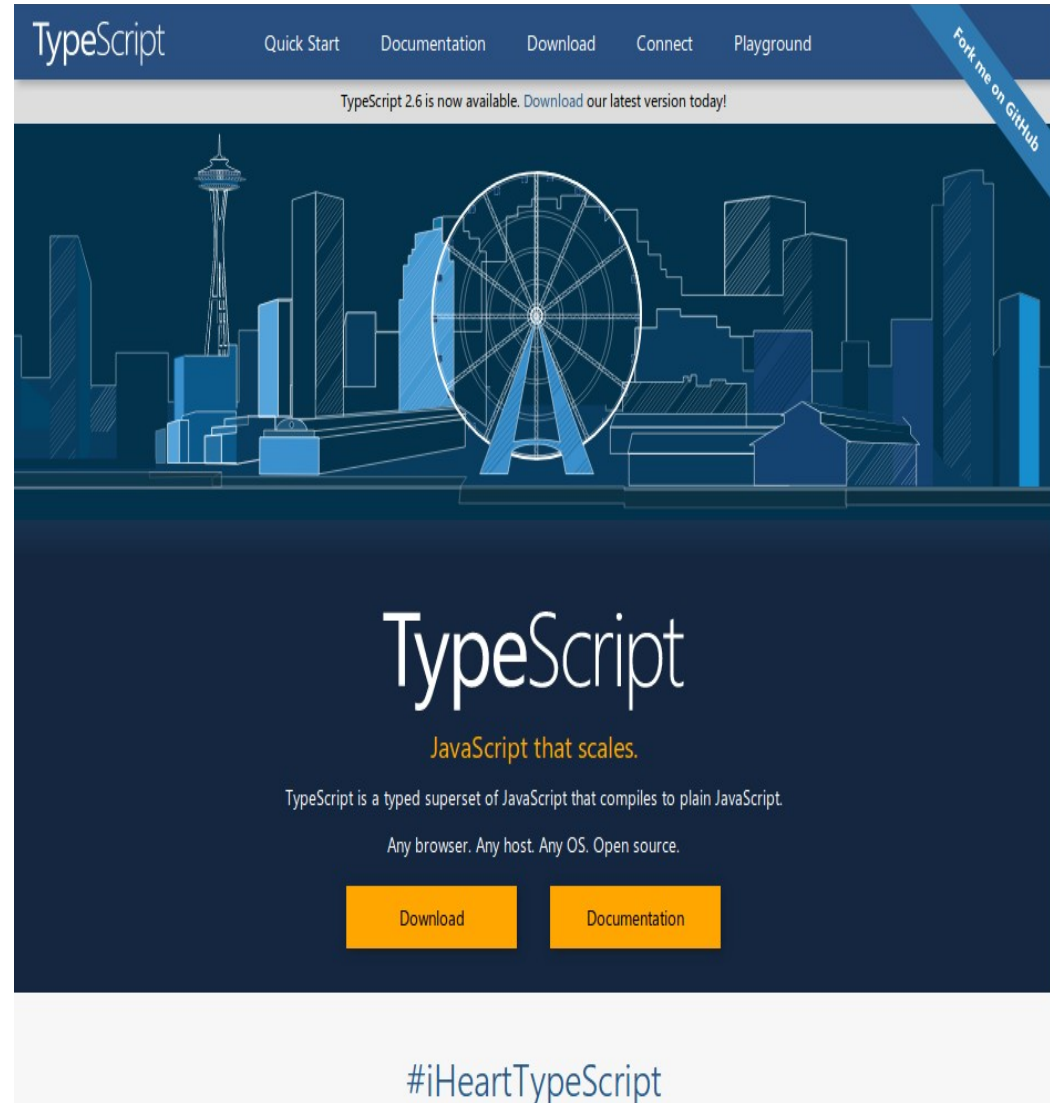
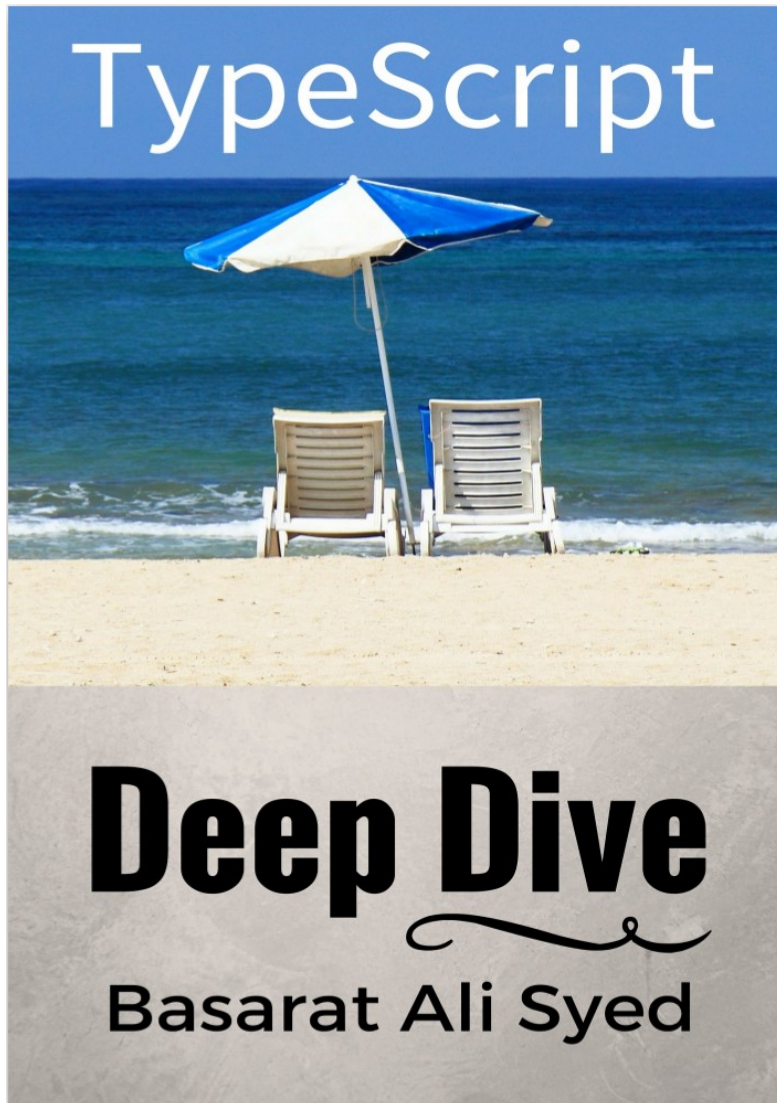




integrata
cegos

Typescript

Programmierung



- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
 - LPGL Lizenzmodell
- Dies ist ein Programmier-Seminar
 - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
 - Musterbeispiele werden zur Verfügung gestellt
 - Diese können am Ende des Seminars als ZIP-Datei kopiert werden
 - USB-Stick oder ähnliches
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung
- Konventionen
 - Befehle werden in `Courier-Schriftart` dargestellt
 - Dateinamen werden in *`Courier-Schriftart`* dargestellt
 - Links werden in `unterstrichener Courier-Schriftart` dargestellt

© Javacream

Javacream

Dr. Rainer Sawitzki

Alois-Gilg-Weg 6

81373 München

Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.

Einführung	1
Aufsetzen eines TypeScript-Projekts	2
Programmierung	3
Benutzerdefinierte Datentypen	4
Einige Details	5

1

TYPESCRIPT EINFÜHRUNG

1.1

TYPESCRIPT IM JAVASCRIPT-UMFELD

- Es gibt kein anerkanntes "JavaScript-Konsortium", das eine allgemeine Spezifikation definiert
- Es existieren viele Bibliotheken und Werkzeuge, die größtenteils von der Open Source-Community vertrieben werden
- Die Einsatzmöglichkeiten von JavaScript sind äußerst vielseitig
 - Im Browser
 - Auf dem Server
 - Als Abfragesprache für NoSQL-Datenbanken
 - Als Skript-Sprache für Produkte
 - In Embedded Systems

- JavaScript-Entwicklungsumgebungen bieten im Vergleich zu anderen Sprachen wie Java wenig Komfort
 - Code-Assists
 - Automatische Fehlererkennung
- Notwendig sind deshalb andere, kreative Ansätze
 - Linter
 - Testgetriebene Entwicklung
 - Benutzung von typisierten Sprachen, die JavaScript generieren

- Der Build-Prozess ist im JavaScript-Umfeld durch die Verbreitung auch relativ kleiner Frameworks und Produkte aufwändig
- Der Build-Prozess muss Abhängigkeiten
 - deklarieren
 - auflösen
 - laden und
 - zum Betrieb ausliefern
- Notwendig ist damit der Aufbau einer Build-Umgebung
 - Packaging Manager
 - Software-Repository
 - Dependency Management

- Das Testen ist durch die Verstrickung von JavaScript mit HTML, CSS und Browser nicht trivial
 - Unit-Tests, die ausschließlich JavaScript-Sequenzen testen, sind eher selten
 - Standalone JavaScript-Interpreter müssen hierfür benutzt werden
- Browser-Tests sind aufwändig und erschweren die Test-Automatisierung drastisch
 - Unterschiedliche Browser-Implementierungen der verschiedenen Hersteller müssen berücksichtigt werden
 - Tests müssen durch einen Tester mit einem UI-Recorder aufgezeichnet werden
 - Headless Browser ohne User Interface ermöglichen wenigstens eine rudimentäre Testautomatisierung

Alles nicht so einfach...

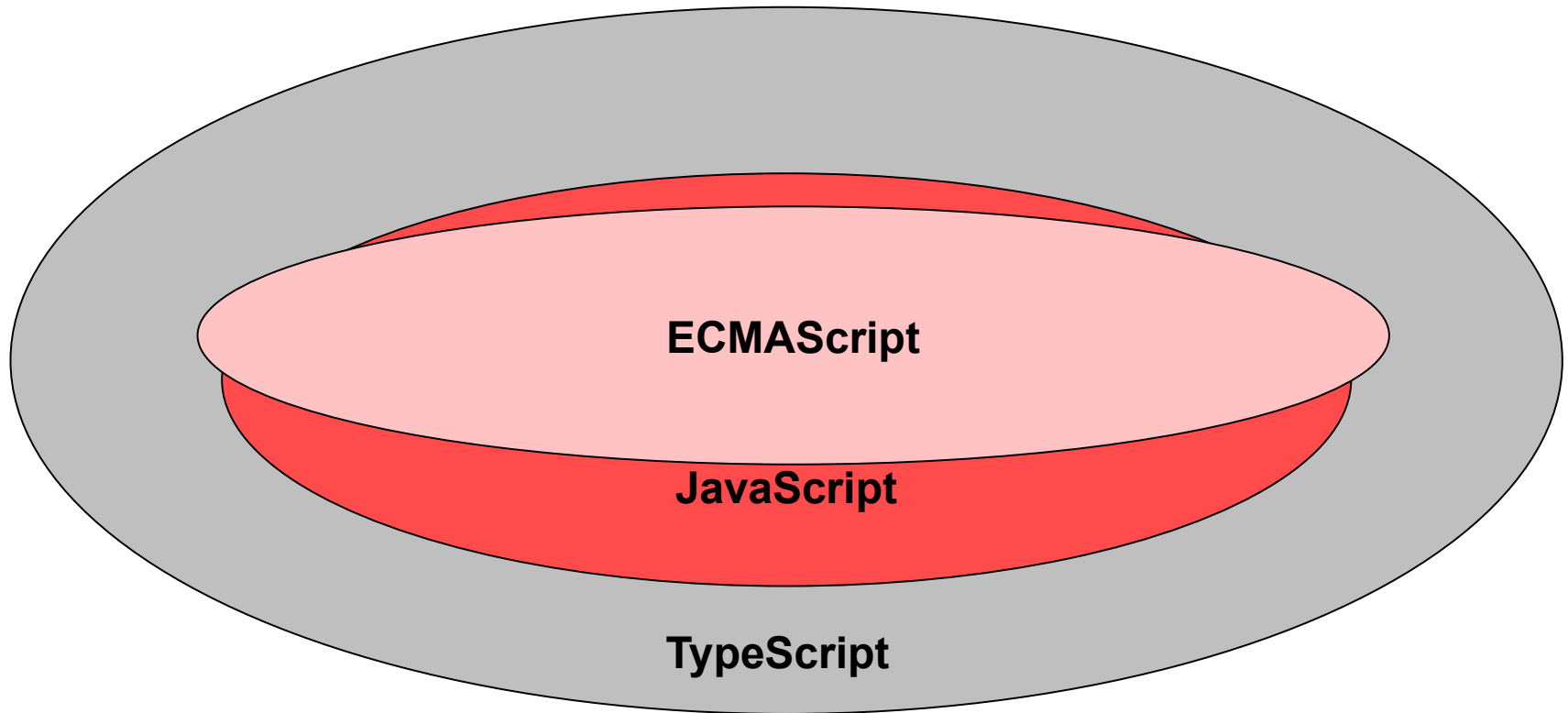


1.2

BEGRIFFE UND EINORDNUNG

- Alle Browser
- Google Chrome V8
- Node
 - basiert auf V8
- Java-Implementierungen
 - Rhino
 - Nashorn

- ECMAScript
 - Eine von der „European Computer Manufacturers Association“ spezifizierte Script-Sprache
 - Enthält elementare Syntax und Sprachkonstrukte
 - JavaScript ist ein Superset von ECMAScript
 - Vorsicht: Nicht alle JavaScript-Engines unterstützen den neuesten Stand von ECMAScript!
 - Siehe <http://en.wikipedia.org/wiki/ECMAScript>
- TypeScript
 - Eine von Microsoft entwickelte typisierte und Klassen-orientierte Programmiersprache
 - Ein Superset von JavaScript
- Andere Sprachen:
 - Coffescript
 - Go
 - ...



- Transpiler
 - Erzeugen aus Script-Sprachen andere Scripte
 - TypeScript wird nach JavaScript transpiliert
- Software-Repositories und –Registries
 - Enthalten Produkte, Bibliotheken, ...
 - Identifikation über einen eindeutigen Namen sowie eine Versionsnummer
 - Zugriff über Netzwerk, primär Internet
- Dependency Management
 - Jede Software enthält eine Deklaration der von ihr benötigten Abhängigkeiten
 - Transitive Dependencies treten auf, wenn eine Dependency selbst wiederum Dependencies deklariert
- Packaging Manager
 - Lokale Installation von Software aus einem Software-Repository
 - Auflösung aller notwendigen Dependencies
 - auch transitiv

2

AUFSETZEN EINES PROJEKTS

2.1

NPM

- Primär ein Packaging Manager
- `npm` ist Bestandteil der `node`-Installation
 - `npm -v`
- Die offizielle `npm` Registry liegt im Internet
 - <https://docs.npmjs.com/misc/registry>
 - Im Wesentlichen eine CouchDB
 - Laden der Software durch RESTful Aufrufe
 - Die `npm`-Registry ist aktuell die größte Sammlung von Software
- Unternehmens-interne oder private Registries können angemietet werden

- npm wird über die Kommandozeile angesprochen
 - eine grafische Oberfläche wird als separates Modul zur Verfügung gestellt
- Hilfesystem
 - `npm -h`
 - `npm <command> -h`
 - <https://docs.npmjs.com/>

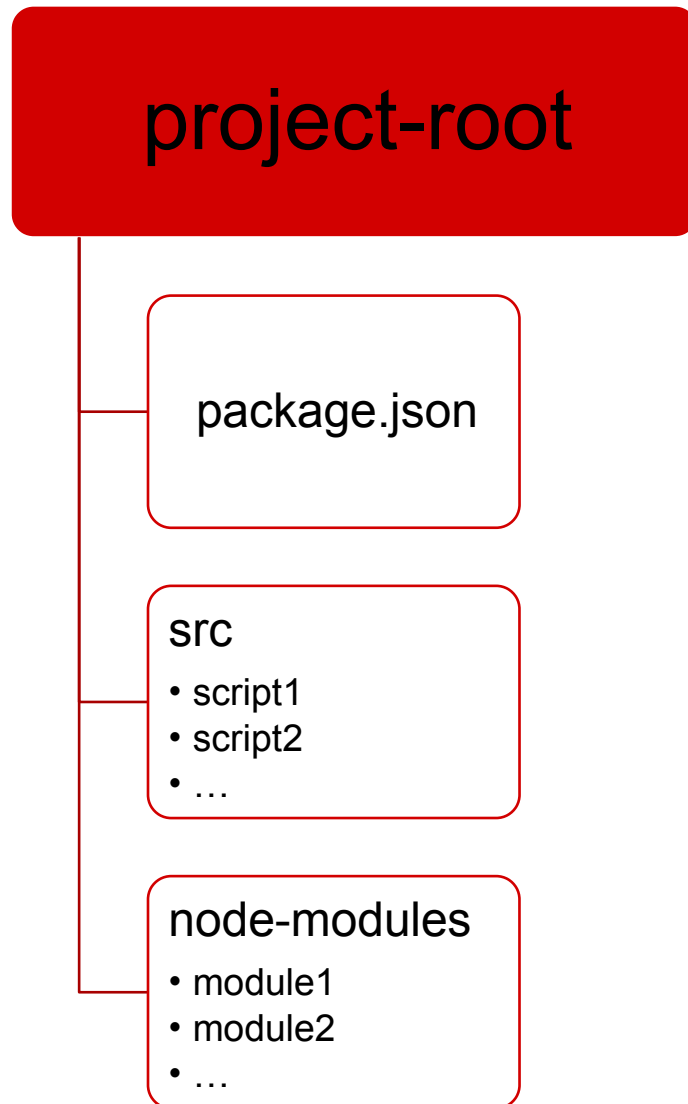
2.2

NODE-MODULES

- Jede via `npm` geladene Bibliothek wird als Node-Module konzipiert
- Jedes Modul besitzt
 - Eine Informationsdatei, die `package.json`, die das Projekt zusätzlich beschreibt
 - Abhängige Bibliotheken im Unterverzeichnis `node_modules`
 - Diese sind selbst ebenfalls Node-Module
 - Einen Entry-Point, in dem der Module-Entwickler das Fachobjekt seines Moduls erzeugt und exportiert
 - Dazu wird dem `module`-Objekt die Eigenschaft `exports` gesetzt
 - Zur Benutzung eines Moduls innerhalb eines Scripts dient der Node-Befehl `require`
 - Der Rückgabewert von `require` ist das vom Modul erzeugte und exportierte Fachobjekt

- Enthält die Projektinformation im JSON-Format
- Die Datei enthält
 - Den Projektnamen
 - Die aktuelle Versionsnummer
 - Meta-Informationen wie Autor, Schlüsselwörter, Lizenz
 - Dependencies
 - Ein `scripts`-Objekt mit ausführbaren Befehlen
 - Diese können mit `npm run <script>` ausgeführt werden

- Jedes `npm`-basierte Projekt ist ein neues Node-Module
- Initialisierung mit `npm init`
 - Dabei werden interaktiv die Informationen abgefragt, die zur Erstellung der initialen `package.json` benötigt werden



Beispiel: Ein einfaches Projekt

```
{  
  "name": "npm-sample",  
  "version": "1.0.0",  
  "description": "a simple training project",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [  
    "training"  
  ],  
  "author": "Javacream",  
  "license": "ISC"  
}
```

- **Datei *index.js***

```
module.exports = {  
  log: function() {  
    console.log('Hello')  
  }  
}
```

- **In der REPL**

```
var training = require('./index.js')  
training.log()
```

- Abhängigkeiten werden mit `npm install` von einer npm-Registry geladen
 - Ohne weitere Konfiguration wird dazu die Standard-Registry benutzt
 - Damit ist eine Internet-Verbindung notwendig
 - Es können aber auch Unternehmens-interne Repository-Server benutzt werden
 - z.B. Nexus
- Rechner-Registry
 - Die Abhängigkeiten werden auf dem Rechner abgelegt
 - Ab jetzt ist damit keine Internet-Verbindung mehr nötig
 - Orte:
 - lokale Ablage in einem Unterverzeichnis namens `node_modules`
 - Empfohlenes Standard-Verfahren zur Installation von Dependencies für eigene Software-Projekte
 - globale Ablage
 - Empfohlenes Standard-Verfahren zur Installation von allgemein verwendbaren Werkzeugen

2.3

EIN TYPESCRIPT-PROJEKT

- TypeScript
 - `npm install typescript --save-dev`
- Lite-Server
 - `npm install lite-server --save-dev`
- Parallelisierung von npm-Kommandos
 - `npm install concurrently --save-dev`

- Initialisierung mit `tsc --init`
 - Erzeugt die Datei `tsconfig.json`
 - Darin werden alle möglichen Konfigurationen angelegt
 - wobei die allermeisten auskommentiert sind
- Für die folgenden Beispiele wird insbesondere der strict-Mode aktiviert


```
"scripts": {  
  "serve": "lite-server",  
  "compile": "tsc --outDir ./dist -p .",  
  "compile-watch": "tsc -w --outDir ./dist -p .",  
  "test": "echo \"Error: no test specified\" && exit  
1",  
  "start": "concurrently \"npm run compile-  
watch\" \"npm run serve\""  
}
```

- Starten mit `npm start`
 - Startet den TypeScript-Transpiler
 - Startet den Lite Server mit Browser-Sync
 - Startet den Default-Browser und stellt die `index.html`-Seite dar
- Parallelisierung
 - Sämtliche Dateien mit der Endung `.ts` im Ordner `src` werden automatisch nach `dist` transpiliert
 - Änderungen der `.ts`-Dateien werden automatisch erkannt
 - Der Server aktualisiert den Browser mit den geänderten Informationen

3

GRUNDLAGEN DER PROGRAMMIERUNG

3.1

EINORDNUNG VON TYPESCRIPT

- Jegliche JavaScript-Anweisung ist valides TypeScript

```
var message = "Hello World!"  
function printout(s) {  
    console.log("Hello World!")  
}  
printout(message)
```

- Syntaktische Fehler werden jedoch vom TypeScript-Compiler erkannt

- Die OOP-Konzepte von ECMAScript sind prinzipiell den Konstrukten in TypeScript sehr ähnlich
 - aber nicht identisch
 - Eine ECMA-Klasse mit Attributen ist keine valide TypeScript-Klasse

3.2

OPERATOREN

- TypeScript unterstützt die aus JavaScript bekannten Operatoren
 - Mathematisch
 - Logisch
- Ebenfalls unterstützt wird der Punkt-Operator zum Zugriff auf Eigenschaften eines Objekts
- Schleifen
 - `for`
 - `while`
- Abfragen
 - `if-else`
 - `switch`

3.3

TYPISIERUNG

- Ein Typ definiert einen Satz von Eigenschaften und Funktionen
- Jede Variable hat einen Typen, der sich nach der Deklaration nicht mehr ändern kann
 - Der Compiler prüft dies
 - Statische Typisierung
- Diese Einschränkung ist für Programmierer häufig vorteilhaft
 - Moderne Entwicklungsumgebungen prüfen die Typisierung bereits während der Eingabe
 - Ein Satz typischer Programmierfehler wird damit bereits frühzeitig erkannt
 - Ebenso beschränkt die Typisierung die möglichen Aufrufe auf einer Variablen, so dass die Entwicklungsumgebung Vorschläge unterbreiten kann
 - Code Assists vermindern damit die Tipparbeit gewaltig

- `let | const<name>`
 - `const name`
 - `let state`

- `boolean`
 - Ein logischer Wert, also `true` oder `false`
- `number`
 - Eine Ganz- oder Kommazahl
- `string`
 - Eine Zeichenkette

- Explizite Typisierung

```
let name : string
let state : boolean
```

- Type Inference

- Hier wird der Typ durch die Zuweisung eines Wertes definiert

```
let name = "Hello"
let state = true
```

- Contextual Type

- Auch bei Zuweisungen versucht TypeScript, untypisierte Deklarationen zu erkennen

```
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.button); //<- Error,
};
```

- Umwandlung des `any`-Typen in einen speziellen Typen

```
const value :any  
let message:string = <string>value  
let message2 : string = value as string
```

- `array`
 - Eine Liste
- `tuple`
 - Eine feste Menge von anderen Basis-Typen
- `enum`
 - Eine feste Menge von Werten

- `null`
 - Eine Eigenschaft ist nicht gesetzt
- `undefined`
 - Eine Eigenschaft oder Funktion ist nicht vorhanden
 - Damit unterschiedlich zu `null`
- `any`
 - Eine untypisierte Variable, die jeden Wert zugewiesen bekommen kann
 - Damit ist bei Bedarf auch eine untypisierte Programmierung auch in TypeScript möglich
- `void`
 - Eine Funktion, die keinen expliziten `return`-Wert aufweist
- `never`
 - Der Rückgabotyp einer Funktion, die kein implizites oder explizites `return`-Statement aufweist
 - Endlose Ausführung oder garantiertes Werfen einer `Exception`

- Namespaces gruppieren Deklarationen
 - Diese sind nur innerhalb des Namespaces direkt ansprechbar
 - Damit wird die Wahrscheinlichkeit von Namenskollisionen vermieden
- Deklarationen werden mit Hilfe des Schlüsselworts `export` anderen Namespaces zur Verfügung gestellt
- Aus einem anderen Namespace müssen die Variablen mit dem Namespace angesprochen werden
 - "Qualifizierte Namen"

```
namespace Namespace1{  
    export let message = "Hello from namespace1"  
}
```

```
namespace Namespace2{  
    console.log(Namespace1.message);  
}
```

- Ein Modul exportiert Deklarationen auf Top-Level-Ebene
- Exportierte Deklarationen können importiert werden
- Zur Unterstützung von Modulen unterstützt der TypeScript-Compiler unterschiedliche Optionen:
 - ES
 - commonjs

4

BENUTZERDEFINIIERTE DATENTYPEN

4.1

INTERFACES

- Ein TypeScript-Interface definiert eine Signatur bestehend aus Eigenschaften
 - Einfache Attribute
 - Funktionen
- Eigenschaften können Optional sein
 - An den Namen der Eigenschaft wird ein `?` ergänzt
- Unveränderbare Eigenschaften werden mit `readonly` deklariert
- Das Interface wird als Typ benutzt
 - Das hierfür benutzte Objekt muss der Struktur des Interfaces entsprechen
 - Dies prüft der Compiler

```
interface Person{
    lastname: string
    readonly firstname: string
    address?: string
    formattedName():string
}

let p:Person = {
    lastname: "Sawitzki",
    firstname: "Rainer",
    formattedName: function(){
        return this.firstname + " " + this.lastname
    }
}
```

- Interfaces können in einer Vererbungshierarchie benutzt werden
 - Schlüsselwort `extends`
- Das Sub-Interface erbt die Struktur des Super-Interfaces


```
interface Worker extends Person{
    company: string
    work(): string
}

let worker:Worker = {
    company: "Integrata",
    lastname: "Sawitzki", firstname: "Rainer",
    formattedName: function(){
        return this.firstname + " " + this.lastname
    },
    work: function(){
        return "working at " + this.company
    }
}
```

4.2

KLASSEN

- Klassen definieren wie Interfaces eine Struktur
 - Die Attribute einer Klasse
- Im Gegensatz zu Interfaces können Klassen aber auch Funktionen implementieren
 - Die Methoden einer Klasse
- Instanzen einer Klasse werden jedoch durch einen Konstruktor-Aufruf erzeugt
 - Dazu dient der `new`-Operator
 - Der Konstruktor selbst ist eine spezielle Methode ohne Rückgabotyp
 - `constructor(params)`

```
class SimplePerson{
  name:string
  height:number
  constructor(name:string, height:number){
    this.name = name
    this.height = height
  }
  sayHello():string{
    return "Hello, my name is " + this.name
  }
}
```

```
let simplePerson = new SimplePerson("Mustermann", 188)
console.log(simplePerson.sayHello())
```

- Methoden können überschrieben werden
 - Eine Subklasse implementiert die selbe Signatur einer Methode wie die Superklasse
 - Die Aufrufe von überschriebenen Methoden werden zur Laufzeit ausgewertet
 - Polymorphie
 - Der Zugriff auf eine Methode der Superklassen-Hierarchie ist mit der Referenz `super` möglich

- TypeScript unterstützt für Attribute und Methoden das Prinzip der Kapselung
 - `public`
 - `protected`
 - `private`

- `readonly`-Attribute sind möglich
- Verkürzter Konstruktor durch "Parameter properties"
 - `constructor(readonly attr:type)` deklariert und setzt ein Attribut

- getter- und setter-Methoden
 - Diese definieren ein "Pseudo-Attribut"
 - Beim lesenden oder schreibenden Zugriff werden die korrespondierenden Methoden aufgerufen

Beispiel: getter und setter

```
class PersonWithGetterAndSetter {
    private _name: string;

    get name(): string {
        console.log("reading name")
        return this._name;
    }

    set name(newName: string) {
        console.log("setting name")
        this._name = newName;
    }
}

let p = new PersonWithGetterAndSetter ();
p.name = "Bob Smith";
console.log(p.name);
```

- Auch Klassen unterstützen das Konzept der Vererbung
- Methoden einer Klasse können auch abstrakt sein
 - Analog zu Definition einer Interface-Funktion
 - Eine Klasse, die mit `new` instanziiert werden soll darf keine abstrakten Methoden enthalten
- Ein Interface kann von einer Klasse erben
 - Allerdings darf die Klasse keine nicht-abstrakten Methoden enthalten
- Eine Klasse kann eine Schnittstelle implementieren
 - Schlüsselwort `implements`

- Mit Hilfe des Schlüsselworts `static` können Attribute und Methoden einer Klasse zugeordnet werden
 - Der Zugriff erfolgt über den Klassennamen

4.3

FUNKTIONEN

- `(param: type, param2: type2) => returnType`
- Damit sind Funktionen in das Typsystem von TypeScript integriert

- **Optionale Parameter**
 - Deklaration eines optionalen Parameters mit ?
 - `param1:type, param2?:type) => returnType`
- **Default-Werte**
 - `param1:type, param2:type = defaultValue) => returnType`
- **Rest-Parameter**
 - `(param: type, ...restParams: type[])`
- **Überladene Methoden**
 - Die Parameterliste wird bei der Auflösung der Signatur berücksichtigt

4.4

ENUMERATIONEN

```
enum Direction {  
    Up = 1,  
    Down,  
    Left,  
    Right,  
}
```


5

EIN PAAR DETAILS

5.1

GENERISCHE DATENTYPEN

```
function identity<T>(arg: T): T {  
    return arg;  
}
```

5.2

INTERSECTION TYPES

- Eine Intersection verbindet zwei Typen

```
function extend<T, U>(first: T, second: U): T & U {  
    let result = <T & U>{};  
    for (let id in first) {  
        (<any>result)[id] = (<any>first)[id];  
    }  
    for (let id in second) {  
        if (!result.hasOwnProperty(id)) {  
            (<any>result)[id] = (<any>second)[id];  
        }  
    }  
    return result;  
}
```

5.3

DECLARATION MERGING

- Werden Typen mehrfach deklariert, so werden deren Eigenschaften gemerged

```
interface Box {  
    height: number;  
    width: number;  
}
```

```
interface Box {  
    scale: number;  
}
```

```
let box: Box = {height: 5, width: 6, scale: 10};
```