Web-basierte Systeme – Übung

05: TypeScript und WebAssembly

Wintersemester 2025

Arne Vogel, Maxim Ritter von Onciul

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Übersicht

TypeScript

WebAssembly

Aufgabe 4

TypeScript

Dynamische vs. Statische Sprachen

Dynamische Sprachen

- Kein Buildsystem nötig
  - + Geringerer Entwicklungsaufwand
  - + Compilezeit entfällt
- Typinformationen zur Laufzeit
  - + Debug-Informationen
  - + Objekte erweiterbar
- Code dynamisch
  - + Kann zur Laufzeit nachgeladen werden
- ...

Statische Sprachen

Dynamische vs. Statische Sprachen

Dynamische Sprachen

- Kein Buildsystem nötig
  - \+ Geringerer Entwicklungsaufwand
  - \+ Compilezeit entfällt
- Typinformationen zur Laufzeit
  - \+ Debug-Informationen
  - \+ Objekte erweiterbar
- Code dynamisch
  - \+ Kann zur Laufzeit nachgeladen werden
- ...

Statische Sprachen

- Codeanalyse
  - \+ (Offensichtliche) Fehler vor Ausführung bemerkt
  - \+ Findet Fehler in nicht ausgeführtem Code
- Optimierungen
  - \+ Bessere Performance zur Laufzeit
- ...

Fehlerfindung in JavaScript

- Fehler in JavaScript erst zur Laufzeit bemerkbar
- In vielen Fällen lösen „unsinnige“ Operationen keinen Fehler aus!
- Beispiel: Undefinierte Variablen:

```
1  let myName = 'David';
2  // ...
3  name = 'Manuel';
```

- `name` nicht mit `let` deklariert
  - ⇒ Globale Variable `name` wird erzeugt

Fehlerfindung in JavaScript

- Beispiel: Fehlende/Überflüssige Funktionsparameter

```
1  function concat(first, second) {
2          console.log(first + ' ' + second);
3  }
4  // ...
5  concat('Hallo'); // Ausgabe: 'Hallo undefined'
```

- Jeder Parameter ist optional
  - Fehlende Parameter sind `undefined`
  - Überflüssige Parameter werden ignoriert

Fehlerfindung in JavaScript

- Beispiel: Rechenoperationen

```
1/0;       // Ergebnis: NaN
'test'/5; // Ergebnis: NaN
[] + {};  // Ergebnis: '[object Object]'
{} + [];  // Ergebnis: 0
```

- JavaScript hat unterschiedliche Regeln für Rechenoperationen

- Viele nicht intuitiv

Fehlerfindung in JavaScript

- Fehlerüberprüfung kann selbst implementiert werden
  - Check ob `undefined`, `NaN`, `0`, ...
  - Anzahl von Parametern und deren Typen

$\Rightarrow$ Extrem aufgeblähter Code

Strict Mode

- Besserer Ansatz: Strict Mode
  - Seit ECMAScript 5
  - Aktiviert mit `"use strict";` als erster Anweisung
  - Abwärtskompatibel
- Weist JS-Engine an bestimmte Operationen zu verbieten
  - Entweder Syntaxfehler beim Parsen oder Laufzeitfehler
- Beispiele:
  - Zuweisung nicht deklarierter Variablen
  - Nutzung des `with` Statements
    - Fehleranfällig und schlecht für Performance
  - Löschen von Variablen und Funktionen
  - Keywords als Variablennamen

TypeScript

- **TypeScript**[1] ist eine Programmiersprache auf Basis von JavaScript
  - Open Source: `https://github.com/Microsoft/TypeScript`
  - Entwickelt von Microsoft
  - Jeder JS Code ist auch TypeScript Code

- TypeScript kann nicht direkt vom Browser ausgeführt werden
  - TypeScript Code wird zu JavaScript **kompiliert**

- Größter Unterschied: Statische Typisierung

  $\Rightarrow$ Typfehler werden zur Compile-Zeit gefunden

TypeScript - Typen

- Typen werden über Annotationen festgelet

```
let myName: string = "Manuel";

function add(first: number, second: number): number {
  return first + second;
}
```

- Nicht jeder Typ muss angegeben werden

```
function add(first: number, second: number) {
  // Compiler erkennt, dass return-Wert Typ number hat
  return first + second;
}
```

TypeScript - Typen

- Basis-Typen:
  - `boolean`, `number`, `string`, `array`, `enum`
    - Array entweder als `number[]` oder `Array<number>`
  - `tuple`
    - Array fester Größe
  - `any`
    - Beliebiger Typ
    - $\Rightarrow$ Keine Typechecks durchgeführt
  - `void` und `never`
    - Für Funktionen ohne Rückgabewert
  - `undefined` und `null`
    - Genutzt um anzuzeigen, welche Variablen den Wert undefined/null annehmen dürfen
- Außerdem Interessant: `HTMLElement`
  - Interface für alle HTML Objekte

TypeScript - Unions

- Mit **Unions** ist es möglich mehr als einen Typen zu erlauben

```ts
let id : number | string;
id = 12345;   // OK
id = "12345"; // OK
id = true;    // Fehler
```

- Besonders mit `undefined` und `null` sinnvoll

```ts
let id : number | string | undefined; // undefined erlaubt
let name : string;                    // undefined nicht erlaubt
```

- Exakten Typ überprüfen: (JS)

```ts
if (typeof message === 'string') { /* ... */ }
```

- Vererbungsgewahr Typ überprüfen: (TS)

```ts
if (animal instanceof Dog) { /* ... */ }
```

TypeScript - Weitere Features

- Definition von Klassen
  - Ab ECMAScript 6 auch in JS
  - Einige zusätzliche Features
    - Vererbung
    - Public, protected und private Variablen
    - Kurzformen für Konstruktor, getter/setter
    - ...
- Interfaces
  - Kann (optionale) Felder und Funktionen enthalten

TypeScript - Build-Vorgang

- TypeScript ist kompilierte Sprache
  - TypeScript → Compiler → JavaScript
- Compiler entfernt allen TS-spezifischen Code
  - Ergebnis is äquivalenter JS Code
  - Kompilieren dient nur Überprüfung auf Fehler

```typescript
interface Person {
  name : string;
}
function printName(p: Person) {
  console.log(p.name);
}
let me: Person = {name: "Manuel"};
printName(me);
```

→

```javascript
function printName(p) {
  console.log(p.name);
}
let me = { name: "Manuel" };
printName(me);
```

TypeScript - Build-Vorgang

Compiler kann konfiguriert werden

- ECMAScript-Version des ausgegebenen Codes
  - Standard is ECMAScript 3
  - Erscheinungsjahr 1999

- Strenge der Fehlerchecks[2]
  - `noImplicitAny`
    - Fehler, wenn fehlender Typ nicht erkannt werden kann
  - `strictNullChecks`
    - Überprüft, ob Werte null/undefined sein dürfen
  - `noImplicitReturns`
    - Fehler, wenn nicht jeder Code-Pfad einen Rückgabewert hat

WebAssembly

Übersicht

TypeScript

WebAssembly

Aufgabe 4

WebAssembly - Überblick

- **WebAssembly (WASM)** ist ein Bytecode-Format für die Ausführung im Web

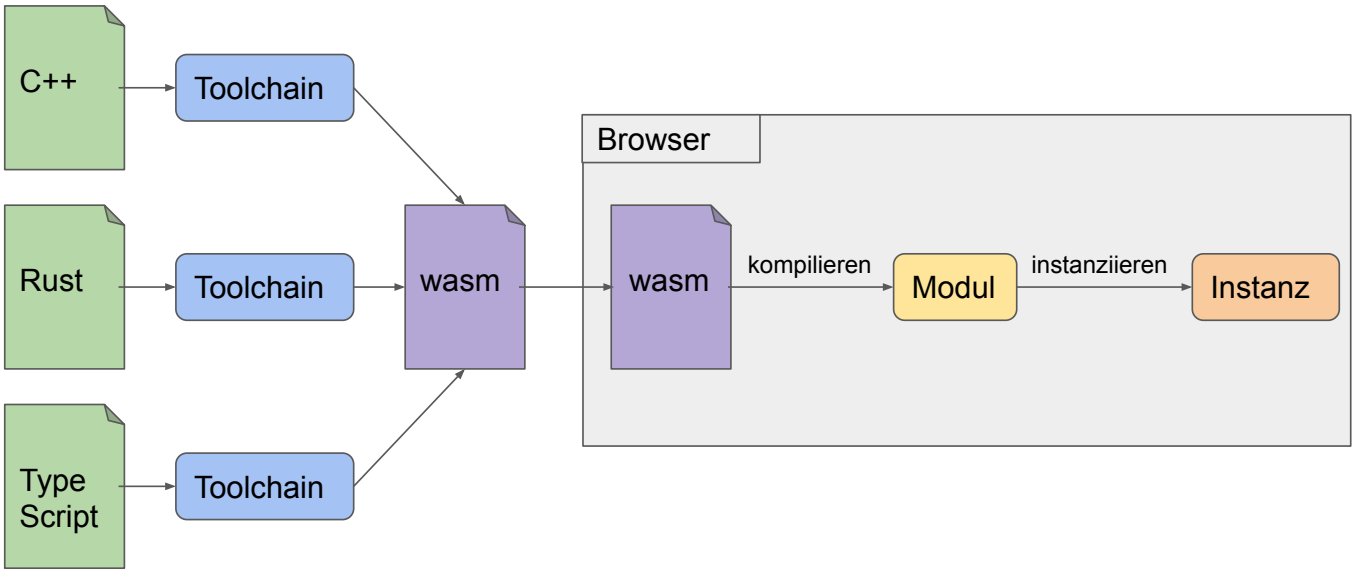
- Ziele:
  - Gute Performanz
    - Ahead-of-Time Optimierungen
    - Kompaktes Format
  - Sicherheit
    - Eingeschränkter Speicherzugriff
    - Kein Zugriff auf DOM, Cookies, ...

Außerhalb des Webs

- Mittlerweile wird WASM auch außerhalb des Browsers eingesetzt
- WebAssemly Funktionen in der Cloud
- Eigenständige WASM-Ausführungsumgebungen
  - WebAssembly als Docker-Alternative
  - Runtimes:

    aWsm, Chicory, EOSVM, Extism, Fizzy, GraalWASM, Happy New Moon With Report, inNative, Lucet, wasm3, Swam, WAKit, WasmEdge, Wasmer, Wasmi, Wasmo, Wasmtime, WasmVM, WAMR, Wizard Research Engine, TWVM, wazero
- Teile von Firefox werden aus WebAssembly kompiliert

Toolchains



- WASM wird nicht von Entwicklern direkt geschrieben, sondern aus anderen Sprachen generiert
- Nach kompilieren ist bekannt, was gebraucht wird
  - Funktionen, Speicher, ...
  - Wenn alles erfüllt, kann instanziiert werden

Toolchains

- Mittlerweile viele Ausgangssprachen unterstützt
  - Unterschiedlich ausgereift
- Überblick hier: `https://github.com/appcypher/awesome-wasm-langs`

Produktiv eingesetzt: C, C++, Rust, Go

Produktiv einsetzbar:
.Net, AssemblyScript, Brainfuck, C#, Clean, Cyber, COBOL, Dart, F#, Forth, Grain, LabVIEW, Lobster, Lox, Lua, Nelua, Never, Rego, TypeScript, WebAssembly, Zig

Instabil, nutzbar:
Ada, C4wa, Crystal, D, Eclair, Eel, Elixir, Janet, Java, JavaScript, KCL, KotlinWasm, Lisp, Lys, Pascal, Perl, PHP, Poetry, Python, Prolog, R, Ring, Ruby, Scheme, Scopes, Swift, Tcl, V, Virgil, Wa, Wonkey

In Arbeit:
Ballerina, BASIC, Co, Faust, Forest, Haskell, Julia, Kou, MoonBit, Nerd, Nim, Ocaml, Plorth, Roc, Theta, Wase, xcc

Veraltet, ungewartet:
Astro, Idris, Speedy.js, Turboscript, Wah, Walt, Wam, Wracket

Toolchains - wasm-pack

- **wasm-pack**[3] ist eine Toolchain zum Generieren von WASM aus Rust
- Intern wird **wasm-bindgen** genutzt
- wasm-pack erzeugt zusätzlichen Code zur besseren Integration

Toolchains - wasm-bindgen

- wasm-bindgen verwendet Annotationen um zu signalisieren, dass Funktion von außen aufgerunfen werden kann

```rust
// Export a `greet` function from Rust to JavaScript, that alerts
// hello message.
#[wasm_bindgen] // <---- Annotation
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

- Aus Code + Annotationen generiert wasm-bindgen WASM und JavaScript

Aufgabe 4

Übersicht

TypeScript

WebAssembly

Aufgabe 4

Aufgabe 4

- Der Chat-Client aus Aufgabe 3 soll um Funktionalität erweitert werden
  - Rechtschreibprüfung
  - Wortvorhersage

- Statt JavaScript sollen diesmal TypeScript und WebAssembly eingesetzt werden

Aufgabe 4.1 - Vorbereitung

- Installation der Toolchains für TypeScript und WebAssembly
- Anpassen von Webpack für neuen Build-Vorgang
  - Bestehender Code aus Aufgabe 3 soll weiterverwendet werden

Aufgabe 4.2 - Rechtschreibprüfung

- Falsch geschriebene Worte werden markiert und Verbesserungsvorschläge gegeben
  - Code für Fehlerfindung und Vorschläge wird gestellt
  - Wörter nicht im Wörterbuch sollen hinzugefügt werden können
- Selbst erstellter Code soll in TypeScript geschrieben werden
  - Soll mit `noImplicitAny` kompilieren
  - Gegebene Hilfsfunktionen um Typen erweitern

Aufgabe 4.3 - Wortvorhersage

- Aus letztem geschriebenen Wort soll nächstes vorgeschlagen werden
- Dazu Rust-Library zu WASM kompiliert und einbinden
  - Library muss um Annotationen erweitert werden
  - Vortrainiertes Modell wird gestellt
- Generierter JavaScript Code soll nachvollzogen werden

Anmerkung
Es gibt nicht für jedes Wort einen Vorschlag!