TypeScript Basis Handleiding

Hier leer je de basisprincipes van TypeScript, met een focus op types, interfaces, generics.

Wat is TypeScript?

TypeScript is een **superset** van JavaScript die **statische types** toevoegt. Het helpt je om fouten vroegtijdig te vinden en maakt je code beter onderhoudbaar.

Onderdelen van TypeScript

TypeScript bestaat uit verschillende onderdelen die samen een krachtige ontwikkelomgeving vormen:

1. **Typesysteem**: Geeft statische typecontrole en helpt bij het vinden van fouten tijdens het schrijven van code.
2. **ES6+ Ondersteuning**: Ondersteunt moderne JavaScript-functies zoals modules, async/await, en arrow functions.
3. **Compiler (tsc)**: Zet TypeScript-code om naar standaard JavaScript, zodat het in elke omgeving werkt.
4. **Configuratiebestand (tsconfig.json)**: Biedt controle over hoe TypeScript wordt gecompileerd.
5. **Type-definities**: Met ingebouwde en externe type-definities via `@types` kunnen bibliotheken zoals React en Node.js probleemloos worden gebruikt.
6. **Integratie met editors**: TypeScript werkt goed samen met IDE's zoals VS Code, waardoor functies zoals IntelliSense beschikbaar zijn. TypeScript is een superset van JavaScript die statische types toevoegt. Het helpt je om fouten vroegtijdig te vinden en maakt je code beter onderhoudbaar.

Types in TypeScript

TypeScript biedt verschillende soorten types om verschillende situaties te ondersteunen:

Primitive Types

1. **string**: Voor tekstwaarden.

```
let name: string = "John";
```

2. **number**: Voor numerieke waarden.

```
let age: number = 25;
```

3. **boolean**: Voor true/false waarden.

```
let isAdmin: boolean = true;
```

4. **null** en **undefined**: Voor respectievelijk geen waarde of een niet-geïnitialiseerde waarde.

```
let empty: null = null;
let notAssigned: undefined = undefined;
```

Complex Types

1. **array**: Voor lijsten van waarden.

```
const numbers: number[] = [1, 2, 3];
```

2. **tuple**: Voor vaste lijsten met specifieke typen op bepaalde posities.

```
let tuple: [string, number] = ["hello", 42];
```

3. **object**: Voor complexe structuren.

```
const person: { name: string; age: number } = { name: "Alice", age: 30
};
```

Special Types

1. **any**: Hiermee kun je elke waarde toewijzen (wordt afgeraden).

```
let random: any = 42;
random = "text";
```

2. **unknown**: Een veiliger alternatief voor `any`.

```
let input: unknown = "hello";
if (typeof input === "string") {
  console.log(input.toUpperCase());
}
```

3. **void**: Geeft aan dat een functie geen waarde retourneert.

```
const logMessage = (message: string): void => {
  console.log(message);
```

```
};
```

4. **never**: Voor functies die nooit een waarde teruggeven (bijvoorbeeld omdat ze een fout gooien).

```
const throwError = (message: string): never => {
  throw new Error(message);
};
```

Geavanceerde Types

1. **Union Types**: Voor waarden die meerdere typen kunnen zijn.

```
let id: string | number;
id = 123;
id = "ABC";
```

2. **Intersection Types**: Combineer meerdere typen.

```
type Employee = { name: string } & { id: number };
let employee: Employee = { name: "John", id: 1 };
```

3. **Literal Types**: Beperk een waarde tot specifieke opties.

```
type Direction = "left" | "right";
let move: Direction = "left";
```

Type Aliases

Maak een alias voor een complex type.

```
type Point = { x: number; y: number };
let point: Point = { x: 10, y: 20 };
```

Interfaces

Beschrijf objectstructuren.

```
interface User {
  name: string;
  age: number;
```

```
}
let user: User = { name: "Alice", age: 25 };
```

Optionele en Readonly Eigenschappen

In TypeScript kun je eigenschappen optioneel maken of aangeven dat ze alleen-lezen zijn.

Optionele Eigenschappen

Een optionele eigenschap wordt aangegeven met een vraagteken (`?`). Dit betekent dat de eigenschap niet verplicht is.

```
interface User {
  name: string;
  age?: number; // Optioneel
}

const user1: User = { name: "Alice" };
const user2: User = { name: "Bob", age: 30 };
```

Readonly Eigenschappen

Readonly eigenschappen kunnen niet worden gewijzigd na initialisatie.

```
interface User {
  readonly id: number;
  name: string;
}

const user: User = { id: 1, name: "Alice" };
// user.id = 2; // Error: Cannot assign to 'id' because it is a read-only
property
```

Generics

Generics maken het mogelijk om typen flexibel en herbruikbaar te maken zonder de typeveiligheid te verliezen. Ze worden gebruikt bij functies, klassen en interfaces.

Generics in Functies

Een generieke functie kan werken met meerdere typen zonder vooraf vast te leggen welk type het is.

```
const identity = <T>(value: T): T => {
  return value;
};
```

```
const number = identity<number>(42);
const text = identity<string>("Hello");
```

Generics in Interfaces

Interfaces kunnen ook generiek zijn, waardoor ze met verschillende typen kunnen werken.

```
interface Box<T> {
  content: T;
}

const stringBox: Box<string> = { content: "Hello" };
const numberBox: Box<number> = { content: 42 };
```

Generics zijn krachtig en bieden een manier om typen flexibel en uitbreidbaar te maken terwijl je typeveiligheid behoudt.

Meer informatie: Officiële TypeScript documentatie.