NEWS · OCTOBER 29, 2025

TypeScript für Delphi-Entwickler: Die Grundlagen verstehen (Teil 2 von 5)

By Dr. Holger Flick

Willkommen zurück

In Teil 1 haben wir besprochen, warum der Übergang von Delphi zu Next.js geschäftlich Sinn macht. Jetzt erkunden wir TypeScript selbst und warum es sich überraschend vertraut für Delphi-Entwickler anfühlt.

Denken Sie daran: Diese Serie handelt nicht davon, Ihnen beizubringen, TypeScript zu programmieren. Es geht darum, Ihnen zu zeigen, dass die Konzepte, die Sie in Delphi gemeistert haben, direkt übertragbar sind, und dass die Lernkurve kürzer ist, als Sie befürchten mögen.

Das Type System: Delphi-DNA in JavaScript

Basic Types - Die Grundlagen

In Delphi deklarieren Sie Typen explizit:

```
var
  Age: Integer;
  Name: string;
  Price: Double;
  IsActive: Boolean;
```

TypeScript ist nahezu identisch:

```
let age: number;
let name: string;
let price: number;
let isActive: boolean;
```

Hinweis: TypeScript verwendet `number` für alle numerischen Types (Integer, Double, Single, etc.). JavaScript unterscheidet intern nicht zwischen Integer- und Floating-Point-Zahlen, aber TypeScripts Type Checking stellt Type Safety sicher.

Type Inference - Der intelligente Compiler

Erinnern Sie sich, wie modernes Delphi Types ableiten kann?

```
var
  Count := 42;  // Inferred as Integer
  Message := 'Hello';  // Inferred as string
```

TypeScript macht das wunderbar:

```
let count = 42;        // Inferred as number
let message = 'Hello'; // Inferred as string
```

Anders brachte diese Weisheit mit. Der Compiler ist intelligent genug, um Types herauszufinden, aber Sie können immer noch explizit sein, wenn Sie Klarheit wollen.

Interfaces - Ihr Alter Freund

Interfaces in Delphi definieren Verträge:

```
type
  ICustomer = interface
    function GetId: Integer;
    function GetName: string;
    procedure SetName(const Value: string);
    property Id: Integer read GetId;
    property Name: string read GetName write SetName;
  end;
```

TypeScript Interfaces sind sauberer und gleichermaßen mächtig:

```
interface Customer {
  id: number;
  name: string;
  email: string;
  readonly createdAt: Date;  // readonly wie "read" in Delphi
  phoneNumber?: string;       // ? bedeutet optional
}
```

Verwendung des Interface:

```
function saveCustomer(customer: Customer): void {
  console.log(`Saving ${customer.name}`);
  // customer.createdAt = new Date(); // ERROR! readonly property
}

const newCustomer: Customer = {
  id: 1,
  name: "John Smith",
  email: "john@example.com",
  createdAt: new Date()
  // phoneNumber is optional, so we can omit it
};

saveCustomer(newCustomer); // Type-safe!
```

Während das Delphi Interface eine implementierte Klasse erfordern würde, stellen TypeScript Interfaces sicher, dass Objekte der erwarteten Form entsprechen, ohne eine explizite Implementierung zu benötigen. Ich lasse die Class-Implementierung hier der Kürze halber weg.

Optionale und Nullable Types

In Delphi könnten Sie verwenden:

```
var
  OptionalValue: Integer;
  HasValue: Boolean;
```

TypeScript hat eingebaute Nullable Types:

```
let optionalValue: number | null = null;  // Kann number oder null sein
let undefinedValue: number | undefined;   // Kann number oder undefined sein
let maybeString: string | null | undefined;

// Type Guard (wie Assigned() Check in Delphi)
if (optionalValue !== null) {
  console.log(optionalValue * 2); // Sicher zu verwenden
}
```

Der `?` Operator macht das noch sauberer:

```
interface Address {
  street: string;
  city: string;
  zipCode?: string;  // Optional property
}

function displayAddress(address: Address): void {
  console.log(address.street);
  console.log(address.zipCode?.toUpperCase()); // Safe Navigation
}
```

Das ist wie Delphis `Assigned()` Check, aber in die Sprach-Syntax eingebaut.

Klassen - Object-Oriented Programming lebt weiter

Sie kennen Klassen. TypeScript hat sie auch:

Delphi Class:

```
type
  TCustomer = class
  private
    FId: Integer;
    FName: string;
    FBalance: Double;
  public
    constructor Create(AId: Integer; const AName: string);
    destructor Destroy; override;
    procedure AddToBalance(Amount: Double);
    property Id: Integer read FId;
    property Name: string read FName write FName;
    property Balance: Double read FBalance;
  end;

constructor TCustomer.Create(AId: Integer; const AName: string);
begin
  inherited Create;
  FId := AId;
  FName := AName;
  FBalance := 0.0;
end;

procedure TCustomer.AddToBalance(Amount: Double);
begin
  FBalance := FBalance + Amount;
end;
```

TypeScript Class:

```
class Customer {
  private id: number;
  private balance: number;
  public name: string;

  constructor(id: number, name: string) {
    this.id = id;
    this.name = name;
    this.balance = 0;
  }

  addToBalance(amount: number): void {
    this.balance += amount;
  }

  getBalance(): number {
    return this.balance;
  }

  // Getter Property (wie Delphi Property read)
  get customerId(): number {
    return this.id;
  }
}

// Verwendung
const customer = new Customer(1, "Jane Doe");
customer.addToBalance(1000);
console.log(customer.getBalance()); // 1000
console.log(customer.customerId);   // 1 (using getter)
```

Moderne TypeScript Kurzschreibweise

TypeScript hat eine Abkürzung, die Delphi-Entwickler schätzen werden:

```
class Product {
  constructor(
    private id: number,
    public name: string,
    public price: number,
    readonly category: string
  ) {
    // Constructor Body kann leer sein!
    // Properties werden automatisch erstellt und zugewiesen
  }

  displayInfo(): void {
    console.log(`${this.name}: $${this.price}`);
  }
}

const product = new Product(1, "Widget", 29.99, "Hardware");
product.displayInfo(); // Widget: $29.99
```

Das ist unglaublich prägnant und behält dabei Type Safety bei.

Generics - Type-Safe Collections

Delphi Generics:

```
type
  TList<T> = class
  private
    FItems: array of T;
  public
    procedure Add(Item: T);
    function Get(Index: Integer): T;
  end;

var
  CustomerList: TList<TCustomer>;
  NumberList: TList<Integer>;
```

TypeScript Generics:

```
// Generic Function
function firstElement<T>(arr: T[]): T | undefined {
  return arr[0];
}

const firstNumber = firstElement([1, 2, 3]);    // Type: number | undefined
const firstName = firstElement(["a", "b"]);     // Type: string | undefined

// Generic Class
class DataStore<T> {
  private items: T[] = [];

  add(item: T): void {
    this.items.push(item);
  }

  get(index: number): T | undefined {
    return this.items[index];
  }

  getAll(): T[] {
    return [...this.items]; // Gibt eine Kopie zurück
  }
}

// Verwendung
const customerStore = new DataStore<Customer>();
customerStore.add(newCustomer);

const numberStore = new DataStore<number>();
numberStore.add(42);
```

Der Compiler stellt Type Safety durchgängig sicher. Versuchen Sie, einen String zu `numberStore` hinzuzufügen, und TypeScript wird Sie zur Compile Time stoppen.

Enums - Named Constants Richtig Gemacht

Delphi Enums:

```
type
  TOrderStatus = (osNew, osPending, osShipped, osDelivered, osCancelled);

var
  Status: TOrderStatus;

begin
  Status := osShipped;
end;
```

TypeScript Enums:

```
enum OrderStatus {
  New = "NEW",
  Pending = "PENDING",
  Shipped = "SHIPPED",
  Delivered = "DELIVERED",
  Cancelled = "CANCELLED"
}

let status: OrderStatus = OrderStatus.Shipped;

function updateOrder(orderId: number, status: OrderStatus): void {
  console.log(`Order ${orderId} is now ${status}`);
}

updateOrder(123, OrderStatus.Delivered);
// updateOrder(123, "DELIVERED"); // ERROR! Type Safety durchgesetzt
```

Const Enums (Performance-Optimierung)

```
const enum Direction {
  Up,
  Down,
  Left,
  Right
}

let move = Direction.Up; // Kompiliert zu: let move = 0 (kein Runtime Overhead)
```

Type Unions und Intersections - Mächtiger als Delphi

TypeScript kann Dinge, die Delphi nicht einfach kann:

Union Types (OR-Logik)

```
type Status = "active" | "inactive" | "pending";
type ID = number | string;

function getUserStatus(id: ID): Status {
  // id kann number ODER string sein
  return "active";
}

getUserStatus(123);        // OK
getUserStatus("user-456"); // OK
getUserStatus(true);       // ERROR!
```

Intersection Types (UND-Logik)

```
interface Timestamped {
  createdAt: Date;
  updatedAt: Date;
}

interface Identifiable {
  id: number;
}

type Entity = Timestamped & Identifiable;

const user: Entity = {
  id: 1,
  createdAt: new Date(),
  updatedAt: new Date()
  // Muss ALLE Properties von beiden Interfaces haben
};
```

Functions - First-Class Citizens

In Delphi deklarieren Sie Funktionen so:

```
function CalculateTotal(Price: Double; Quantity: Integer): Double;
begin
  Result := Price * Quantity;
end;
```

TypeScript Functions sind flexibler:

```
// Traditionelle Function
function calculateTotal(price: number, quantity: number): number {
  return price * quantity;
}

// Arrow Function (Lambda Expression)
const calculateTotal = (price: number, quantity: number): number => {
  return price * quantity;
};

// Prägnante Arrow Function (impliziter Return)
const calculateTotal = (price: number, quantity: number): number =>
  price * quantity;

// Optionale Parameter
function greet(name: string, title?: string): string {
  return title ? `${title} ${name}` : name;
}

greet("Smith");           // "Smith"
greet("Smith", "Dr.");    // "Dr. Smith"

// Default Parameter
function createUser(name: string, role: string = "user"): void {
  console.log(`Creating ${role}: ${name}`);
}

createUser("John");              // "Creating user: John"
createUser("Jane", "admin");     // "Creating admin: Jane"
```

Type Aliases für Functions

```
type MathOperation = (a: number, b: number) => number;

const add: MathOperation = (a, b) => a + b;
const multiply: MathOperation = (a, b) => a * b;

function calculate(op: MathOperation, x: number, y: number): number {
  return op(x, y);
}

console.log(calculate(add, 5, 3));      // 8
console.log(calculate(multiply, 5, 3)); // 15
```

Das ist ähnlich zu Delphis Procedural Types, aber eleganter.

Arrays und Collections: Ein anderer Ansatz

In Delphi arbeiten Sie mit Arrays so:

```
var
  Numbers: TArray<Integer>;
begin
  SetLength(Numbers, 3);
  Numbers[0] := 1;
  Numbers[1] := 2;
  Numbers[2] := 3;
end;
```

TypeScript Arrays sind einfacher und mächtiger:

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

Aber hier wird es interessant. TypeScript umarmt einen Functional Programming Stil, der sich anfangs ungewohnt anfühlen könnte:

```
// Anstatt einer for-Schleife verwenden Sie oft map, filter, reduce
const doubled = numbers.map(n => n * 2);        // [2, 4, 6]
const evens = numbers.filter(n => n % 2 === 0); // [2]
const sum = numbers.reduce((acc, n) => acc + n, 0); // 6
```

Dieser Stil ist prägnanter, sobald Sie sich daran gewöhnt haben. Denken Sie daran als deklarative statt imperative Programmierung. Anstatt dem Computer zu sagen, *wie* er die Schleife durchlaufen soll, sagen Sie ihm, *was* Sie mit jedem Element machen wollen.

Was das alles für Sie bedeutet

Sie haben jetzt die Kernkonzepte von TypeScript gesehen und wie sie sich auf Delphi abbilden:

- **Typen sind vertraut**: `Integer` wird zu `number`, `string` bleibt `string`, `Boolean` wird zu `boolean`
- **Interfaces funktionieren gleich**: Verträge für Ihre Datenstrukturen definieren
- **Klassen existieren immer noch**: Object-Oriented Programming ist nicht verschwunden
- **Generics sind da**: Type-safe Collections und Functions

- **Enums bieten Named Constants**: Genau wie in Delphi

Die Syntax ist anders, aber die Konzepte sind identisch. Das ist kein Zufall—Anders Hejlsberg entwarf beide Sprachen mit derselben Philosophie: Entwicklern mächtige Tools geben, die Fehler früh abfangen.

Der Functional Shift

Die größte Denkweise-Änderung ist nicht die Syntax — es ist der Programmierstil. TypeScript (und modernes JavaScript) umarmt funktionale Programmiermuster mehr als Delphi:

- **Immutability**: Bevorzugung von Werten, die sich nicht ändern
- **Pure Functions**: Functions ohne Side Effects
- **Array Methods**: `.map()`, `.filter()`, `.reduce()` statt Schleifen
- **Arrow Functions**: Prägnant und klar

Das ist nicht falsch—es ist anders. Viele Delphi-Entwickler finden, dass sie nach der Anpassung diesen Stil für bestimmte Problemtypen tatsächlich bevorzugen. Für andere bedeutet TypeScripts Unterstützung für Klassen, dass sie bei objektorientierten Ansätzen bleiben können.

Das Schöne ist, Sie haben Auswahlmöglichkeiten.

Warum das für Ihre Migration wichtig ist

Zu verstehen, dass TypeScript keine fremde Sprache ist—es ist ein Dialekt dessen, was Sie bereits kennen—ist entscheidend für die Planung Ihrer Migrationsstrategie.

Wenn Sie bewerten, ob Sie eine Delphi-Anwendung ins Web verschieben sollen, ist eine der größten Fragen: "Kann mein Team das lernen?" Die Antwort, wie Sie gesehen haben, ist ja. Die Syntax ist anders, aber die Konzepte sind dieselben.

Die wirklichen Fragen sind:

- Wie viel Ihrer Business Logic kann direkt übersetzt werden?
- Was muss für Web Architektur überdacht werden?
- Wie lange wird der Übergang dauern?
- Was können Sie in Phasen versus alles auf einmal machen?

Das sind strategische Fragen, keine technischen. Und sie sind für jede Anwendung unterschiedlich.

Wenn Sie bereit für Hilfe sind

Wenn Sie das lesen und über Ihre spezifischen Delphi-Anwendungen nachdenken, fragen Sie sich wahrscheinlich:

- "Könnten wir unser Order Management System migrieren?"
- "Was ist mit unserer Lagerverwaltungs-Anwendung?"
- "Wir haben 15 Jahre Business Logic—kann die erhalten werden?"
- "Wie lange würde das wirklich dauern?"

Das sind genau die Fragen, bei denen ich Delphi-Shops helfe. Der erste Schritt ist nicht Code schreiben—es ist zu verstehen, was Sie haben, was Ihre Ziele sind, und wie ein realistischer Migrationspfad aussieht.

Ich biete Migration Consulting speziell für Delphi-Anwendungen:

- Bewertung Ihrer aktuellen Delphi Codebase
- Identifikation dessen, was einfach übersetzt werden kann und was Überarbeitung braucht
- Architecture Planning für Web Deployment
- Team Training in TypeScript und modernem Web Development
- Hands-on Migration Assistance
- Post-Migration Support

Das Ziel ist nicht, Ihr Team zu ersetzen—es ist, ihnen das Wissen und Framework zu geben, um die Entwicklung nach der initialen Migration fortzusetzen.

Was als Nächstes kommt

Sie verstehen jetzt TypeScripts Kernfeatures und wie sie sich auf Ihr Delphi-Wissen abbilden. In Teil 3 werden wir React Components erkunden—denken Sie an sie als die VCL für Web-Anwendungen. Sie werden sehen, wie das Erstellen von User Interfaces in React konzeptionell ähnlich zum Erstellen von Forms in Delphi ist.

Die Komponenten (Components) werden anders aussehen, aber das Denken ist dasselbe: kleine, wiederverwendbare Teile zu größeren Anwendungen zusammenfügen.

Denken Sie über die Migration Ihrer Delphi-Anwendung nach? Lassen Sie uns ein Gespräch über Ihre spezifische Situation führen. Auch wenn Sie nur Optionen erkunden, kann ich Ihnen helfen zu verstehen, was involviert ist und was für Ihr Geschäft Sinn macht. Kontaktieren Sie mich!

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —