NEWS · OCTOBER 29, 2025

TypeScript for Delphi Developers: Understanding the Fundamentals (Part 2 of 5)

By Dr. Holger Flick

> **Update (July 2026): the full series recap — with proof**
> This series now has a companion post that condenses all five parts into one page and pairs them with real-world 2026 case studies — including a production Delphi multi-tier app rebuilt on Next.js in about four hours: Delphi to Next.js: The Complete Series, and the Proof It Works.

Welcome Back

In Part 1, we discussed why the transition from Delphi to Next.js makes business sense. Now, let's explore TypeScript itself and why it feels surprisingly familiar to Delphi developers.

Remember: this series isn't about teaching you to code TypeScript. It's about showing you that the concepts you've mastered in Delphi translate directly, and that the learning curve is shorter than you might fear.

The Type System: Delphi DNA in JavaScript

Basic Types - The Fundamentals

In Delphi, you declare types explicitly:

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

TypeScript is nearly identical:

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

Note: TypeScript uses `number` for all numeric types (Integer, Double, Single, etc.). JavaScript doesn't distinguish between integer and floating-point numbers internally, but TypeScript's type checking ensures type safety.

Type Inference - The Smart Compiler

Remember how modern Delphi can infer types?

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

TypeScript does this beautifully:

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

Anders brought this wisdom forward. The compiler is smart enough to figure out types, but you can still be explicit when you want clarity.

Interfaces - Your Old Friend

Interfaces in Delphi define contracts:

```
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 are cleaner and equally powerful:

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

Using the 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!
```

As the Delphi interface would require a class to be implemented, TypeScript interfaces ensure that objects conform to the expected shape without needing explicit implementation. I obmit the class implementation here for brevity.

Optional and Nullable Types

In Delphi, you might use:

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

TypeScript has built-in nullable types:

```
let optionalValue: number | null = null;  // Can be number or null
let undefinedValue: number | undefined;   // Can be number or undefined
let maybeString: string | null | undefined;

// Type guard (like checking Assigned() in Delphi)
if (optionalValue !== null) {
  console.log(optionalValue * 2); // Safe to use
}
```

The `?` operator makes this even cleaner:

```
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
}
```

This is like Delphi's `Assigned()` check, but built into the language syntax.

Classes - Object-Oriented Programming Lives On

You know classes. TypeScript has them too:

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 (like Delphi property read)
  get customerId(): number {
    return this.id;
  }
}

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

Modern TypeScript Shorthand

TypeScript has a shortcut that Delphi developers will appreciate:

```
class Product {
  constructor(
    private id: number,
    public name: string,
    public price: number,
    readonly category: string
  ) {
    // Constructor body can be empty!
    // Properties are automatically created and assigned
  }

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

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

This is incredibly concise while maintaining type safety.

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]; // Returns a copy
  }
}

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

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

The compiler ensures type safety throughout. Try to add a string to `numberStore`, and TypeScript will stop you at compile time.

Enums - Named Constants Done Right

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 enforced
```

Const Enums (Performance Optimization)

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

let move = Direction.Up; // Compiled to: let move = 0 (no runtime overhead)
```

Type Unions and Intersections - More Powerful Than Delphi

TypeScript can do things Delphi can't easily do:

Union Types (OR logic)

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

function getUserStatus(id: ID): Status {
  // id can be number OR string
  return "active";
}

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

Intersection Types (AND logic)

```
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()
  // Must have ALL properties from both interfaces
};
```

Functions - First-Class Citizens

In Delphi, you declare functions:

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

TypeScript functions are more flexible:

```
// Traditional 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;
};

// Concise arrow function (implicit return)
const calculateTotal = (price: number, quantity: number): number =>
  price * quantity;

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

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

// Default parameters
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 for 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
```

This is similar to Delphi's procedural types but more elegant.

Arrays and Collections: A Different Approach

In Delphi, you work with arrays like this:

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

TypeScript arrays are simpler and more powerful:

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

But here's where it gets interesting. TypeScript embraces a functional programming style that might feel unfamiliar at first:

```
// Instead of a for loop, you often use 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
```

This style is more concise once you get used to it. Think of it as declarative rather than imperative programming. Instead of telling the computer *how* to loop, you tell it *what* you want to do with each item.

What This Means for You

You've now seen the core concepts of TypeScript and how they map to Delphi:

- **Types are familiar**: Integer becomes number, string stays string, Boolean becomes boolean
- **Interfaces work the same way**: Defining contracts for your data structures
- **Classes still exist**: Object-oriented programming didn't disappear
- **Generics are there**: Type-safe collections and functions

- **Enums provide named constants**: Just like in Delphi

The syntax is different, but the concepts are identical. That's not an accident—Anders Hejlsberg designed both languages with the same philosophy: give developers powerful tools that catch errors early.

The Functional Shift

The biggest mindset change isn't the syntax—it's the programming style. TypeScript (and modern JavaScript) embraces functional programming patterns more than Delphi does:

- **Immutability**: Prefer values that don't change
- **Pure functions**: Functions without side effects
- **Array methods**: `.map()`, `.filter()`, `.reduce()` instead of loops
- **Arrow functions**: Concise and clear

This isn't wrong—it's different. Many Delphi developers find that once they adjust, they actually prefer this style for certain types of problems. For others, TypeScript's support for classes means you can stick with object-oriented approaches.

The beauty is you have choices.

Why This Matters for Your Migration

Understanding that TypeScript isn't a foreign language—it's a dialect of what you already know—is crucial for planning your migration strategy.

When you're evaluating whether to move a Delphi application to the web, one of the biggest questions is: "Can my team learn this?" The answer, as you've seen, is yes. The syntax is different, but the concepts are the same.

The real questions are:

- How much of your business logic can be directly translated?
- What needs to be rethought for web architecture?
- How long will the transition take?
- What can you do in phases versus all at once?

These are strategic questions, not technical ones. And they're different for every application.

When You're Ready for Help

If you're reading this and thinking about your specific Delphi applications, you're probably wondering:

- "Could we migrate our order management system?"
- "What about our inventory application?"
- "We have 15 years of business logic—can that be preserved?"
- "How long would this really take?"

These are exactly the questions I help Delphi shops answer. The first step isn't writing code—it's understanding what you have, what your goals are, and what a realistic migration path looks like.

I offer migration consulting specifically for Delphi applications:

- Assessment of your current Delphi codebase
- Identification of what translates easily and what needs rework
- Architecture planning for web deployment
- Team training in TypeScript and modern web development
- Hands-on migration assistance
- Post-migration support

The goal isn't to replace your team—it's to give them the knowledge and framework to continue development after the initial migration.

What's Next

You now understand TypeScript's core features and how they map to your Delphi knowledge. In Part 3, we'll explore React components—think of them as the VCL for web applications. You'll see how building user interfaces in React is conceptually similar to building forms in Delphi.

The components will look different, but the thinking is the same: compose small, reusable pieces into larger applications.

Thinking about migrating your Delphi application? Let's have a conversation about your specific situation.

Even if you're just exploring options, I can help you understand what's involved and what makes sense for your business. Contact me!

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 —