



Werde Teil unseres Teams!

Spannende Projekte erwarten Dich als  
Linux & OpenSource Spezialist

Mehr auf der Innenseite!

jobs@b1-systems.de



# DEVELOPER

Herbst 2022

# Programmiersprachen – Next Generation

## TypeScript

Wartungsarmer Code dank Typsystem  
Den TypeScript-Compiler richtig nutzen  
Infrastructure as Code mit TypeScript

## Kotlin

Einstieg in Kotlin: Klassischer Ansatz – neu gedacht  
Objektorientierte und funktionale Konzepte vereint  
Native Apps entwickeln mit Kotlin Multiplatform Mobile

## Rust

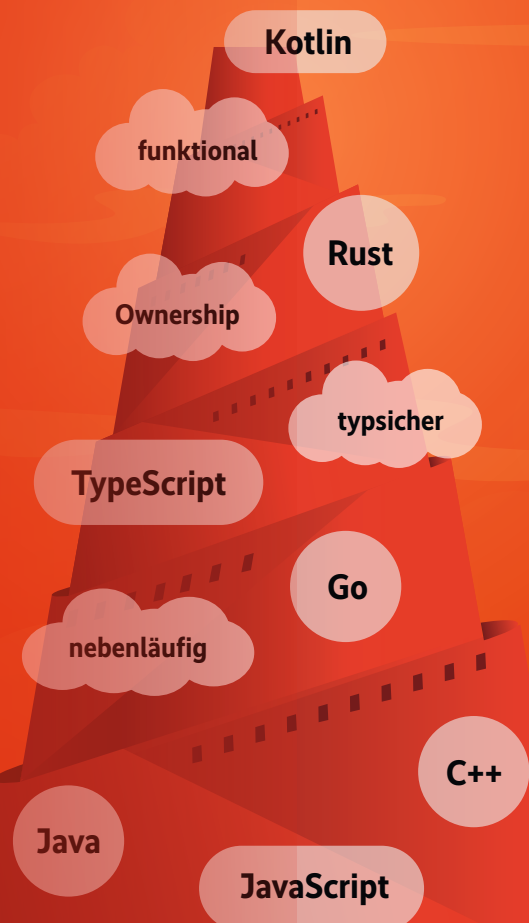
Sicheres Speichermanagement dank Ownership  
Effiziente asynchrone Programmierung  
Unverzichtbares Werkzeug: Rust-Makros

## Go

Flexibilität gewinnen mit Interfaces  
Mehr Sicherheit für die Software Supply Chain  
Concurrency und Generics

## Sprachenvielfalt

Quantenprogrammierung mit Q#  
Robuster und lesbarer Code mit Konzepten in C++20  
Java 17: Mit klarer Syntax und neuen Datentypen gegen die Konkurrenz



14,90 €

Österreich 16,40 €  
Schweiz 27,90 CHF  
Luxemburg 17,10 €

www.ix.de

Quereinsteiger (w|m|d), Praktikanten (w|m|d) &  
Werkstudenten (w|m|d) willkommen



Wir suchen

## Junior & Senior Open Source Admins/Consultants

(w|m|d)

**Das sind Deine Stärken:**

- eine schnelle Auffassungsgabe
- analytisches Denken
- Leidenschaft für Linux/Open Source
- Selbstorganisation & Kommunikation

**Das bringst Du mit:**

- (Erste) Erfahrungen mit Linux
- Kenntnisse in einem oder mehreren der folgenden Bereiche wünschenswert:
  - Cloud
  - Container
  - System- und Konfigurationsmanagement
  - High Availability
  - Monitoring
  - Continuous Integration/Delivery

**Das erwartet Dich bei uns:**

- ca. 90% Homeoffice
- vielfältige & abwechslungsreiche Einsätze
- familiäres Arbeitsklima & flache Hierarchien
- Vereinbarkeit von Job & Familie

**Mehr erfahren & bewerben:**

**jobs@b1-systems.de**

Formlose Bewerbung genügt



**B1 Systems GmbH - Ihr Linux-Partner**

Linux/Open Source Consulting, Training, Managed Service & Support

ROCKOLDING · KÖLN · BERLIN · DRESDEN · JENA

[www.b1-systems.de](http://www.b1-systems.de) · [info@b1-systems.de](mailto:info@b1-systems.de)

# Von Generation zu Generation

**A**m Anfang war der Code und der Code war in Assembler geschrieben, wenn man die Veteranen der Programmierung vor dem Computerzeitalter von Ada Lovelace bis zu Konrad Zuse außen vor lässt. Schnell kamen die ersten höheren Sprachen wie FORTRAN und COBOL auf, später BASIC und C. Und immer wieder findet ein Generationswechsel statt, sei es durch neue Konzepte wie die objektorientierte Programmierung unter anderem in C++ oder ein neues Umfeld wie das Internet, das die Tür für JavaScript öffnete.

Die Welt der Softwareentwicklung ist im steten Wandel. Die heute dominierenden Sprachen Java, Python, JavaScript und C# wurden vor dreißig Jahren noch nicht an den Unis unterrichtet. Kaum ein Developer wird über das gesamte Berufsleben bei denselben Sprachen bleiben. Und das ist gut so. Neue Patterns bringen frische Konzepte und öffnen die Tür für neue Sprachen. Die prozedurale Programmierung und später die Objektorientierung befreiten uns vom Spaghetti-Code, die Java Virtual Machine vereinfachte die plattformübergreifende Programmierung und der Garbage Collector verhinderte typische Speicherfehler.

Wer auf eine neue Sprache wechseln will, muss den richtigen Zeitpunkt abpassen, den Sweetspot im Hype-Zyklus, wenn das Tal der Enttäuschung durchschritten und das Plateau der Produktivität oder zumindest der Pfad der Erleuchtung erreicht ist.

Vier Sprachen haben aus unserer Sicht gute Chancen, die nächste Generation zu bilden: TypeScript, Kotlin, Rust und Go. Sie sind reif genug und haben unterschiedliche geistige Vorgänger, die sie um eigene Konzepte erweitern: TypeScript bringt Typsicherheit zu JavaScript, Kotlin vermischt funktionale Konzepte mit objektorientierter Programmierung auf der JVM. Gegenüber C bringt das Ownership-Konzept von Rust Speichersicherheit ohne den Overhead eines Garbage Collector, und Go zielt mit Blick auf Cloud-Computing und

Anwendungen im Cluster auf nebenläufige Programmierung. Und das ist nur die Oberfläche der Neuerungen der Next Generation.

Anders als beim Turmbau zu Babel ist die Sprachenvielfalt in der Softwareentwicklung kein Nachteil. Projekte lassen sich sprachübergreifend umsetzen und die meisten Programmiersprachen bieten Konzepte für das geschmeidige Zusammenspiel. Damit kommt es nicht zu der babylonischen Sprachverwirrung, die seinerzeit verhinderte, dass die übermütigen Menschen den Turm bis in den Himmel bauten. Die Spitze des Programmiersprachenturms ist noch nicht erreicht. Go, Rust, Kotlin und TypeScript entstanden in den Jahren 2009 bis 2012 und sind gut ein Jahrzehnt gereift. Welche Sprachen die nächste Generation der 2030er-Jahre bilden, wird sich zeigen.

RAINALD MENGE-SONNENTAG





## TypeScript

Die Programmiersprache TypeScript zeichnet sich durch ein mächtiges statisches Typsystem aus, das hilft, Programmierfehler zu vermeiden. Dank verschiedener Operatoren und Konstrukte bietet das JavaScript-Superset aber auch die Flexibilität, Verknüpfungen und Abhängigkeiten zwischen Typen zu erstellen. TypeScript lässt sich nahtlos in JavaScript-Projekte einbinden und ist mittlerweile aus der Welt der Frontend-Frameworks nicht mehr wegzudenken.

ab Seite 7

## Kotlin

Kotlin verbindet funktionale Konzepte mit objektorientierter Programmierung auf der JVM. Als Alternative zu Java verspricht Kotlin höhere Effizienz: Die Programmiersprache punktet mit klarer Struktur und guter Lesbarkeit. Kotlin lässt sich auch jenseits der JVM nutzen: Das SDK Kotlin Multiplatform Mobile eröffnet die Möglichkeit, native Anwendungen plattformübergreifend zu entwickeln und dabei einmal erstellte Businesslogik wiederzuverwenden.

ab Seite 45



## TypeScript

Typsicher und komfortabel	8
Wartungsarmer Code mit dem Typsystem	16
JavaScript in typsicher: TypeScript im Web-Framework	22
Programmieren statt Konfigurieren: Infrastructure as Code	26
Design bis API: TypeScript's Compiler verstehen und einsetzen	32
Tiefer Blick in das Typsystem	39

## Kotlin

Einstieg in Kotlin: Klassischer Ansatz – neu gedacht	46
Effizienter entwickeln mit Kotlin	52
Eine Sprache vereint zwei Welten: funktional und objektorientiert	58
Native Apps entwickeln mit Kotlin Multiplatform Mobile	64
Jetpack Compose: ein Blick auf Androids UI-Technik	69

## Rust

Memory Management: Speichermanagement in Rust	74
Blick auf die asynchrone Programmierung	78
Tokio als asynchrone Laufzeitumgebung	83
Makros in Rust: Einführung in ein unverzichtbares Werkzeug	88

## Go

Entwickeln für verteilte Systeme	94
Mit Go sicher in die Cloud	99
Interfaces: reine Typ-Sache	104
Go Generics – Das karge Leben ist vorbei mit generischen Typen	108
Concurrency – Nebenläufigkeit leicht gemacht	115
Kryptografie in Go	120

## Sprachenvielfalt

Einstieg in Microsofts Quantensprache Q#	126
Java 17: LTS-Release rundet wichtige Sprachfeatures ab	132
C++20-Konzepte (Teil 1): Robusterer generischer Code mit Konzepten	136
C++20-Konzepte (Teil 2): Neue Wege mit Konzepten	140
Vite.js: Rasantes JavaScript-Build-Tool	146
App-Entwicklung mit Flutter 3	152

## Sonstiges

Editorial	3
Impressum	151



## Rust

Mit dem Ownership-Konzept zielt Rust auf mehr Speichersicherheit ab und verzichtet dabei auf den Overhead eines Garbage Collector, legt die Hürde für Neueinsteiger aber recht hoch. Das gilt gleichermaßen für die asynchrone Programmierung, für die eine eigene Laufzeitumgebung wie Tokio notwendig ist. Ein mächtiges Werkzeug in Rust sind Makros, die weit über die aus C/C++ bekannten einfachen Textersetzungen hinaus gehen. Sie sind nicht nur weniger fehleranfällig, sondern eröffnen auch die Möglichkeit, Domain-specific Languages in den Rust-Code zu integrieren.

ab Seite 73

## Go

Zu den Stärken von Go zählt Concurrency. In Gestalt der Goroutinen ist nebenläufige Programmierung in der Sprache elegant und effizient umgesetzt. Sie vermeiden die gefürchteten Data Races, daher eignet sich Go besonders für verteilte Anwendungen. Den Interfaces verdankt Go zudem die Flexibilität dynamisch typisierter Sprachen und mittels der seit Go 1.18 endlich verfügbaren Generics lassen sich nun auch generische Funktionen, Strukturen sowie Channels erstellen. Mit Blick auf die Sicherheit der Software Supply Chain bietet Go eine Reihe von Konzepten, die helfen, Angriffe zu vermeiden.

ab Seite 93



## Sprachenvielfalt

Für jede Aufgabenstellung findet sich eine geeignete Programmiersprache. In der Quantenprogrammierung lässt sich Microsofts Q# sowohl eigenständig als auch im Zusammenspiel mit Python und .NET-Sprachen verwenden. Etablierte Platzhirsche wie Java und C++ investieren kontinuierlich in neue Funktionen und Konzepte, um sich gegen die Konkurrenz zu behaupten. Rasante Build-Tools wie Vite verleihen TypeScript im JavaScript-Ökosystem noch mehr Fahrt und das Gespann aus Flutter und Dart empfiehlt sich als universelles Cross-Plattform-Werkzeug.

ab Seite 125



# // heise devSec()

Die Konferenz für  
sichere Software- und  
Webentwicklung

**4. – 6. Oktober 2022  
in Karlsruhe**

## Vor-Ort-Konferenz ... wieder unter Menschen

In der Softwareentwicklung muss Sicherheit von Anfang an mitgedacht werden. Die heise devSec hilft Ihnen dabei.

### Aus dem Programm:

- ☉ Sicherheitsrisiko Single-Page-Anwendungen
- ☉ Automatisierte Sicherheitstests mit Open Source
- ☉ Mit SBOMs die Software Supply Chain absichern
- ☉ C/C++ Compiler-Checks richtig einsetzen
- ☉ Sichere C#-Anwendungen entwickeln
- ☉ Post-Quantum Cryptography auf dem Sprung in die Praxis

Podiumsdiskussion mit Experten zur **Software Supply Chain Security**

**Jetzt anmelden: [www.heise-devsec.de](http://www.heise-devsec.de)**

**Jetzt  
schnell  
Ihr Ticket  
sichern!**

Goldspensoren

**CyberRes**  
A Micro Focus Line of Business

**SECURAI**

**WIBU  
SYSTEMS**

Veranstalter

@ heise Security @ heise Developer

**dpunkt.verlag**



## TypeScript

Die Programmiersprache TypeScript zeichnet sich durch ein mächtiges statisches Typsystem aus, das Entwicklerinnen und Entwicklern hilft, Programmierfehler zu vermeiden. Dank verschiedener Operatoren und Konstrukte bietet das JavaScript-Superset aber auch die Flexibilität, Verknüpfungen und Abhängigkeiten zwischen Typen zu erstellen. Die komplexe, über die API aber leicht zugängliche Umsetzung der Operatoren und Konstrukte spiegelt sich auch im TypeScript-Compiler wider.

TypeScript lässt sich nahtlos in JavaScript-Projekte einbinden und ist mittlerweile aus der Welt der Frontend-Frameworks nicht mehr wegzudenken. Sogar als Alternative zu Konfigurationssprachen wie YAML für Infrastructure as Code eignet sich TypeScript.

Typsicher und komfortabel mit TypeScript	8
Wartungsarmer Code mit dem TypeScript-Typsystem	16
JavaScript in typsicher: TypeScript	22
Programmieren statt Konfigurieren: Infrastruktur als TypeScript-Code	26
Design bis API: TypeScript's Compiler verstehen und einsetzen	32
Tiefer Blick in das Typsystem von TypeScript	39



# Typsicher und komfortabel mit TypeScript

Von Nils Hartmann

TypeScript erweitert JavaScript um ein statisches Typsystem. Die Programmiersprache lässt sich nahtlos in JavaScript-Projekten verwenden.

■ Durch die rasante Verbreitung immer komplexerer Anwendungen im Browser hat die Sprache JavaScript in den letzten Jahren einen wahren Boom erlebt. Dass sie kein statisches Typsystem besitzt, kann allerdings zu Laufzeitfehlern und unwartbarem Code führen. Diese Lücke schließt die Sprache TypeScript, die auf JavaScript aufbaut und ein mächtiges und flexibles statisches Typsystem zur Verfügung stellt.

Microsoft hat die Sprache entwickelt und setzt sie unter anderem für Office365 und VS Code ein. Der Erfinder von TypeScript ist Anders Hejlsberg, der auch als Vater von C# und Turbo Pascal gilt. Der Sourcecode der Programmiersprache steht unter der Apache-Lizenz.

## Der Spatz in der Hand

In einem Interview im Jahr 2018 hat Rod Johnson, der Erfinder des Spring-Frameworks für Java gesagt, TypeScript sei für ihn die „zurzeit wichtigste Sprache“ mit einem „massiven Wachstum“ (das Zitat und die weiteren Links zum Artikel finden sich unter [ix.de/zcry](http://ix.de/zcry)). Laut Johnson sei es zwar möglich, eine gänzlich neue „ideale“ Programmiersprache zu entwickeln, die „50 mal besser“ als bisherige sei, aber man müsse sich fragen, wie einfach deren Einführung ist. Auf der anderen Seite könne man auch eine Sprache erschaffen, die nur zweimal besser als eine bestehende Sprache ist, dafür aber einfach zu adaptieren. Dieser Kandidat sei TypeScript.

Damit spielt Johnson auf ein zentrales Prinzip von TypeScript an. Demnach sollte die Sprache mit und in bestehenden JavaScript-Projekten jederzeit funktionieren. Im optimalen Fall solle es reichen, das Projekt mit TypeScript zu ergänzen

und Schritt für Schritt die Typsicherheit hinzuzufügen. TypeScript soll der Migration nicht im Weg stehen und im Zweifel sogar Code mit Typfehlern akzeptieren, damit während einer Migrationsphase immer gewährleistet ist, dass sich die Anwendung weiterhin ausführen lässt.

Die Strategie scheint aufgegangen zu sein: Die großen Webframeworks Angular, Vue und Svelte sind mittlerweile in TypeScript implementiert, und auch React bringt TypeScript-Support mit. Viele populäre Bibliotheken wie Jest und Redux sind auf TypeScript portiert und alle verbreiteten Editoren und IDEs kennen die Sprache.

Damit IDEs und Editoren übergreifend und einfach TypeScript-Support anbieten können, bringt die Programmiersprache einen lokalen Language Server für das Language Server Protocol (LSP) mit. Er arbeitet unabhängig vom Werkzeug und parst den Code. Unter anderem untersucht er, ob der Code in Ordnung ist oder welche Fehler er enthält. Außerdem kann er Refactorings durchführen. Eine IDE, die TypeScript-Support

### IX-TRACT

- ▶ TypeScript ist ein von Microsoft entwickeltes Superset von JavaScript.
- ▶ Die Programmiersprache bietet ein mächtiges statisches Typsystem, das Programmierfehler verhindern kann.
- ▶ Das Tooling lässt keine Wünsche offen, sodass der produktive Einsatz gewährleistet ist.



## Listing 1: Dynamische Typen in JavaScript

```
let person = "Susi";
console.log(typeof person); // Ausgabe: "string"
console.log(person.toUpperCase()); // Ausgabe: SUSI

person = 32;
console.log(typeof person); // Ausgabe: "number"
console.log(person + 1); // Ausgabe: 33

person = function() { return "Kate" };
console.log(typeof person); // Ausgabe: "function"
console.log(person()); // Ausgabe: Kate

person = undefined;
console.log(typeof person); // Ausgabe: "undefined"
```

anbieten möchte, muss somit lediglich den Language Server einbinden und das Ergebnis darstellen. Dadurch verhalten sich nahezu alle IDEs im Zusammenspiel mit TypeScript identisch, inklusive Fehlermeldungen und Refactoring-Funktionen. Außerdem können sie zum Release einer neuen TypeScript-Version unmittelbar die neuen Features anbieten.

## Dynamisch: Das Typsystem von JavaScript

TypeScript ist eine Obermenge von JavaScript. Jeder gültige JavaScript-Code ist somit gültiger TypeScript-Code, aber die Programmiersprache fügt neue Syntax-Konstrukte hinzu, insbesondere für Typannotation.

Listing 1 zeigt exemplarisch die dynamische Natur des JavaScript-Typsysteams. Es deklariert eine Variable, die zunächst zur Laufzeit den Typ `string` annimmt. Durch das Zuweisen einer Zahl ist sie später vom Typ `number`, danach wird sie zu einer Funktion und schließlich `undefined`. Der Typ einer Variablen lässt sich zur Laufzeit mit dem `typeof`-Operator ermitteln.

Das Beispiel zeigt syntaktisch einwandfreien JavaScript-Code, der zur Laufzeit funktioniert, da eine Variable ihren Typ ständig dynamisch anpassen kann: JavaScript hat ein dynamisches Typsystem. Die Flexibilität hat ihren Preis, denn beim Betrachten eines Codeausschnittes lässt sich der Typ einer Variable zur Laufzeit schwer bestimmen. Zwar helfen einige Werkzeuge dabei, die aber ebenfalls je nach Code an ihre Grenzen stoßen.

Die Funktion `sayHello` in Listing 2 erwartet genau einen Parameter, aber ohne in die Implementation zu schauen, lässt sich nicht ohne Weiteres erkennen, von welchem Typ. Da in JavaScript keine Prüfung der Typen stattfindet, ist der Code der Funktion syntaktisch korrekt und lässt sich ausführen. Allerdings würde beim zweiten Aufruf der Funktion im Listing ein Laufzeitfehler auftreten, weil sie sich zwar grundsätzlich mit einer Zahl aufrufen lässt, aber für den Datentyp `number` keine `toUpperCase`-Funktion definiert ist.

Die Wahrscheinlichkeit, dass das dynamische Typsystem zu Fehlern führt, wächst mit der Größe der Anwendung. Die Analyse von Code ist aufwendig, und das automatische Refactoring ist oft gar nicht oder nur risikobehaftet möglich.

## Statische Typen

Viele Programmiersprachen verwenden ein statisches Typsystem. Dabei erhält eine Variable beim Anlegen einen Typ, der entweder manuell festgesetzt ist oder den die Sprache ermittelt. Einmal festgelegt, ändert sich der Typ nie mehr. Die Typinformation können Tools wie Compiler beim Entwickeln oder Bauen der Anwendung überprüfen. TypeScript verfolgt diesen Ansatz. Der in Listing 1 gezeigte Code würde in TypeScript einen Compile-Fehler verursachen, obwohl er keine explizite Typzuweisung enthält. TypeScript kann häufig den Typ einer Variablen beim Anlegen über Typinferenz ermitteln. Für den Beispielfall kann TypeScript für die Variable `person` den Typ `string` herleiten und wirft in den folgenden Zuweisungen auf eine

## Listing 2: Typfehler in JavaScript

```
function sayHello(person) {
  return "Hello, " + person.toUpperCase();
}

sayHello("Klaus"); // OK
sayHello(7); // Fehler in sayHello:
// TypeError: person.toUpperCase
// is not a function
```

```
let person = "Susi";
let person: string;
console.log(person.toUpperCase());
```

Ein Tooltip im Editor zeigt den hergeleiteten TypeScript-Typ für `person` an (Abb. 1).

`number` beziehungsweise `function` jeweils Compile-Fehler aus. Da in TypeScript `undefined` und `null` zwei eigene Typen sind, würde die letzte Zeile in Listing 1 ebenfalls einen Compile-Fehler verursachen. Die IDE kann beim Entwickeln den hergeleiteten Typ anzeigen (siehe Abbildung 1). Wenn die automatische Herleitung eines Typs nicht funktioniert, lässt sich der Typ explizit hinter dem Namen der Variable angeben (Listing 3).

Das manuelle Setzen eines Typs kann sinnvoll sein, wenn beispielsweise die Variable `person` nicht nur den Typ `string`, sondern weitere Typen wie `null` oder `undefined` annehmen darf. Dafür existiert der sogenannte Union Type, der mehrere Typen zusammenfasst. Der Code in Listing 4 erlaubt es, der `person`-Variable neben einer Zeichenkette die Typen `null` oder `undefined` zuzuweisen. Die Union-Type-Definition lässt sich folgendermaßen lesen: `person` ist `string` oder `null` oder `undefined`.

TypeScript kann nicht nur den Typ von Variablen herleiten, sondern auch den Rückgabety von Funktionen. Die Funktion in Listing 5 liefert über `return` einen String zurück. Damit kennt TypeScript den Rückgabety. Der Typ der Variable `g` wird bei der Zuweisung ebenfalls zu `string`, auf dem der Aufruf von `toUpperCase` erlaubt ist.

Die Typen der Funktionsparameter kann TypeScript nicht herleiten. Der Code muss für jeden Parameter eine Typannotation aufweisen. Die Syntax ist identisch mit der Typannotation an Variablen (Listing 6).

## Typsicheres Arbeiten mit eigenen Objekten

Anders als in Java und C# spielen Klassen in JavaScript eine untergeordnete Rolle. In der Regel arbeiten Entwicklerinnen und Entwickler stattdessen einfach mit Objekten, die zur Laufzeit nur den allgemeinen Typ `object` haben. TypeScript erlaubt, die Struktur eines Objektes zu beschreiben, um Anforderungen daran auszudrücken. Dazu dient entweder `type` oder

## Listing 3: Explizite Typangabe in TypeScript

```
let person: string = "Susi";
console.log(person.toUpperCase()); // Ausgabe: SUSI
person = undefined; // Fehler: Type 'undefined' is not
                    // assignable to type 'string'
```

## Listing 4: Union Type in TypeScript

```
let person: string | null | undefined = "Susi";

console.log(person.toUpperCase()); // Ausgabe: SUSI
person = undefined; // OK
person = null; // OK
person = "Klaus"; // OK
```

### Listing 5: Typinferenz bei Funktionen

```
function greet() {
    return "Hello"
}

let g = greet();
g.toUpperCase(); // OK g ist ein String
```

### Listing 6: Typinferenz für Funktionen

```
function greet(person: string) {
    return "Hello, " + person.toUpperCase()
}

greet("Susi"); // OK
greet(null); // Fehler: Argument of type 'null'
              // is not assignable to parameter of type 'string'
```

### Listing 7: Individuelle Objektdefinitionen

```
type Person = {
    firstname: string;
    lastname: string;
    age?: number;
}

let klaus: Person = {
    firstname: "Klaus",
    lastname: "Müller",
    age: 32
} // OK

let susi = {
    firstname: "Susi",
    lastname: "Meier"
};

let p: Person = susi; // OK
```

interface als Schlüsselwort. Die Unterschiede zwischen diesen beiden Konzepten sind marginal und anfangs zu vernachlässigen. Die type- beziehungsweise interface-Definition legt die Properties eines Objektes und deren Typen fest. Außerdem lässt sich angeben, ob eine Property in dem Objekt optional und/oder readonly ist.

Listing 7 zeigt die Definition eines Person-Objekts, das aus `firstname`, `lastname` und `age` besteht, wobei letztere Property durch das Fragezeichen als optional gekennzeichnet ist. Anschließend erzeugt der Code eine Variable, die ein Objekt des Person-Typs annehmen soll.

Der zweite Teil weist einer Variable `susi` ein Objekt zu, das strukturell dem Person-Typ entspricht. Daher ist es in der letzten Zeile erlaubt, die Variable `susi` an die Variable `p` zu übergeben, die explizit vom Typ `Person` ist. An dieser Stelle vergleicht TypeScript den für `susi` ermittelten mit dem für `p` erwarteten Typ. Da der abgeleitete Typ von `susi` strukturell `Person` entspricht, erlaubt TypeScript die Zuweisung. Dasselbe würde passieren, wenn es zwei Objekttypen unterschiedlichen Namens gäbe, die zueinander kompatibel sind. Damit ließe sich einer Variablen, die explizit vom Typ `Person` ist, ein Wert zuweisen, der beispielsweise von einem kompatiblen Typ `Employee` ist. Dieses sogenannte Structural Typing heißt umgangssprachlich Duck Typing: Wenn etwas aussieht wie eine Ente (Duck) und sich verhält wie eine Ente, ist es vermutlich eine Ente – unabhängig davon, ob es ein Schild mit der Aufschrift „Ente“ trägt oder nicht. Java und C# verwenden hingegen das sogenannte Nominal Typing, bei dem zwei Typen immer unterschiedlich sind, wenn ihre qualifizierten Namen voneinander abweichen.

## Typprüfung

In JavaScript lassen sich Funktionen anders als in objektorientierten Sprachen wie Java oder C# nicht überladen: Eine Funktion gibt es nur einmal. Das ist ein Grund, warum Funktionen mehrere Typen für ein Argument akzeptieren und zur Laufzeit prüfen, welchen Typen sie übergeben bekommen haben, um jeweils den korrekten Code auszuführen. Listing 8 zeigt die erweiterte `greet`-Funktion, die im Vergleich zu Listing 5 zusätzlich zum `string` ein `Person`-Objekt entgegennimmt.

Die Funktion prüft zur Laufzeit mit dem JavaScript-Operator `typeof`, welchen der beiden Typen sie übergeben bekommen hat und führt den passenden Code aus. Das Beispiel zeigt ein weiteres Feature von TypeScript: Die Sprache interpretiert den Code, der zur Laufzeit die Typprüfungen beispielsweise mit `typeof` durchführt und zieht daraus bereits zur Entwicklungszeit Rückschlüsse auf die Typen. Daher ändert sich der Typ des `person`-Parameters in der `greet`-Funktion. In der ersten Zeile ist der Typ `string` oder `Person`. Vor dem `if`-Block könnten daher faktisch keine Operationen auf der Variablen aufgerufen werden, da `string` und das `Person`-Objekt bis auf einige Default-Methoden keine Gemeinsamkeiten haben. Insbesondere hat das Objekt keine `toUpperCase`-Methode, die sich aufrufen lässt, und der `String` bietet keine `lastName`-Property, die für den Gruß im Falle des Objekts zum Einsatz kommt. Die Folge wäre somit ein Compile-Fehler.

Innerhalb des `if`-Blocks leitet TypeScript den Typ jedoch aus der `typeof`-Prüfung als `string` ab. Daher lässt sich die `toUpperCase`-Funktion verwenden. Da der Programmablauf

### Listing 8: Union Types als Funktionsparameter

```
function greet(person: string | Person) {
    // person ist hier string | Person

    console.log(person.toUpperCase());
    // Fehler: Property 'toUpperCase' does not exist
    // on type 'string | Person'.

    if (typeof person === "string") {
        // person ist hier string
        return "Hello, " + person.toUpperCase();
    }

    // Person ist hier Person
    return "Hello, " + person.lastname.toUpperCase()
}
```

### Listing 9: Type Narrowing

```
function greet(person: string | Person | null) {
    // person ist hier string | Person | null

    if (person === null) {
        // person ist hier null
        return "";
    }

    if (typeof person === "string") {
        // person ist hier string
        return "Hello, " + person.toUpperCase();
    }

    // person ist hier Person
    return "Hello, " + person.lastname.toUpperCase()
}
```

heise +

ct

iX

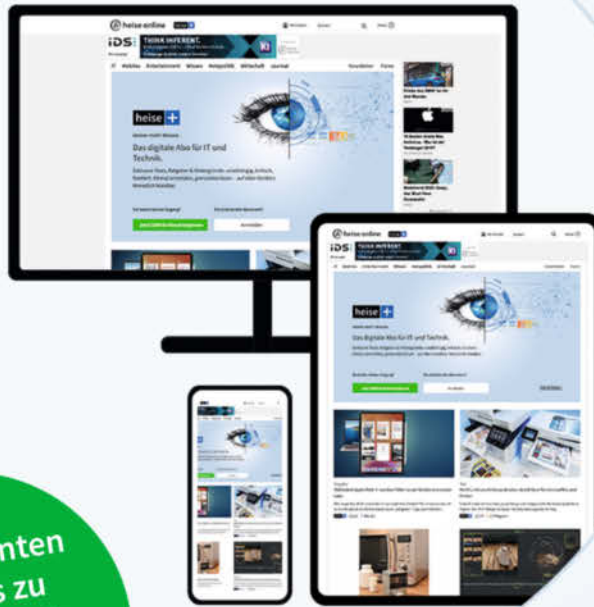
Mac&i

Make:

MIT  
Technology  
Review  
Das Magazin für Innovation von Heise

ct *Fotografie*

iX-Abonnenten  
lesen bis zu  
**75%**  
günstiger



## Das digitale Abo für IT und Technik.

**Exklusives Angebot für iX-Abonnenten:**  
Sonderrabatt für Magazinabonnenten

- ✓ Zugriff auf alle Artikel von heise+
- ✓ Jeden Freitag exklusiver Newsletter der Chefredaktion
- ✓ Alle Heise-Magazine online lesen: c't, iX, Technology Review, Mac & i, Make und c't Fotografie
- ✓ 1. Monat gratis lesen – danach jederzeit kündbar

Sie möchten dieses Exklusiv-Angebot nutzen? Jetzt bestellen unter:

[heise.de/plus-testen](https://heise.de/plus-testen)

✉ [leserservice@heise.de](mailto:leserservice@heise.de) ☎ 0541 80009 120

Ein Angebot von: Heise Medien GmbH & Co. KG • Karl-Wiechert-Allee 10 • 30625 Hannover

### Listing 10: Tagged Union Types

```

type VerifyIbanAction = {
  name: "VerifyIban";
  iban: string;
}

type VerifyAgeAction = {
  name: "VerifyAge";
  age: number;
}

function handleAction(action: VerifyIbanAction
  | VerifyAgeAction) {
  switch (action.name) {
    case "VerifyIban":
      // action ist hier VerifyIbanAction
      return verifyIban(action.iban);
    case "VerifyAge":
      // action ist hier VerifyAgeAction
      return verifyAge(action.age); }
}

function verifyIban(iban: string) { /* ... */ }

function verifyAge(age: number) { /* ... */ }

```

### Listing 11: Der never-Typ

```

function handleAction(action: VerifyIbanAction
  | VerifyAgeAction) {
  switch (action.name) { // ... wie gesehen ...
  }

  // action ist hier never
  handleInvalidAction(action);
}

function handleInvalidAction(action: never) {
  // Implementierung ausgelassen
}

```

die Funktion innerhalb des if-Blocks über return verlässt, kann TypeScript für den Bereich danach davon ausgehen, dass der Typ Person sein muss. Die Vorgehensweise heißt Type Narrowing, also etwa Typverengung: TypeScript schränkt eine Menge von Typen, die in einem Union-Typ zusammengefasst sind, durch unterschiedliche Prüfungen ein.

Die if-Prüfung mit typeof wird in TypeScript als Type Guard bezeichnet, weil sie als eine Art Wächter dient, der nur bestimmte Typen durchlässt. Zu den weiteren Type Guards zählt unter anderem die Prüfung auf null. Listing 9 erweitert die greet-Funktion, sodass sie zusätzlich null akzeptiert. Ohne

```

type VerifyIbanAction = {--
};

type VerifyAgeAction = {--
};

function handleAction(action: VerifyIbanAction |
  VerifyAgeAction) {
  switch (action.name) {
    case "VerifyAge":
      // ...
    case "VerifyIban":
      // ...
  }
  handleInvalidAction(action);
}

```



TypeScript kennt die Ausprägungen der name-Property (Abb. 2).

### Listing 12: Utility-Typen

```

// wie oben gesehen
type Person = {
  firstname: string;
  lastname: string;
  age?: number;
};

function patch(person: Readonly<Partial<Person>>) {
  person.firstname = "Klaus";
  // Fehler: Cannot assign to 'firstname'
  // because it is a read-only property.

  // Alle Eigenschaften aus Person sind hier optional,
  // deswegen führt folgender Aufruf zu einem Fehler,
  // obwohl lastname im Person-Type nicht als optional
  // gekennzeichnet ist
  person.lastname.toUpperCase();
  // Fehler: Object is possibly 'undefined'

  // weitere Implementierung ausgelassen
}

patch({
  firstname: "Klaus", // OK
  age: 32 // OK
}) // OK, auch ohne lastname, weil alle
// Eigenschaften optional gemacht wurden

patch({
  // Alle Eigenschaften im Objekt sind optional,
  // aber ihre ursprünglichen Typen bleiben
  // ansonsten erhalten
  lastname: null
  // Fehler: Type 'null' is not assignable
  // to type 'string | undefined'
})

```

die zusätzliche Prüfung gäbe es einen Fehler, da der Typ vor dem letzten return-Statement Person oder null wäre.

## Objekte unterscheiden

Eine Sonderform bilden die sogenannten Tagged Union Types. Dabei bestehen alle einzelnen Typen aus Objekten, die sich über den Wert einer Property unterscheiden. Listing 10 zeigt zwei action-Objekte, die als Gemeinsamkeit eine name-Property haben. Deren Wert ist für die beiden Objekte festgelegt, sodass nur einer der beiden Strings erlaubt ist (siehe Abbildung 2). Da die Property in beiden Funktionen vorhanden ist, kann vor dem Verwenden wie in der handleAction-Funktion eine Prüfung darauf erfolgen. Obwohl sie erst zur Laufzeit erfolgt, kann TypeScript daraus Rückschlüsse auf den Typ in den jeweiligen case-Zweigen schließen und die korrekte Verwendung überprüfen.

## Sag niemals nie

Spannend ist, was mit dem Typ von action passiert, wenn der Code unvorhergesehen hinter das switch-Statement gelangt. Die beiden Treffer auf die Property verlassen jeweils die Funktion. Gemäß der Typdefinition dürfte das Programm niemals den Code hinter dem switch-Statement erreichen. Allerdings führt TypeScript die Typüberprüfung lediglich zur Build-Zeit durch, was in den meisten Fällen ausreicht. TypeScript kann das Verwenden der Typen beziehungsweise die Aufrufe der

## Listing 13: Umsetzung eines Listener mit Generics

```
// Beispiel 1: keyof-Operator
// PersonKeys kann nur ein Key-Name aus dem Person-Objekt sein
type PersonKeys = keyof Person;
let lastname: PersonKeys = "lastname"; // OK
let city: PersonKeys = "city"; // Fehler: Type '"city"' is not assignable to type 'keyof Person'

// Beispiel 2: generische Funktion mit keyof-Operator
function addListener<O extends object>(o: O, propertyName: keyof O) {
  // Implementierung ausgelassen
}

const susi2 = {
  firstname: "Susi",
  lastname: "Meier",
};

addListener(susi2, "firstname"); // OK

addListener(susi2, "age");
// Fehler: Argument of type '"age"' is not assignable
// to parameter of type '"firstname" | "lastname"'
```

handleAction-Funktion in der ganzen Anwendung überprüfen, sodass der korrekte Aufruf der Funktion wahrscheinlich ist. Wenn eine Anwendung die übergebenen Objekte jedoch nicht statisch erzeugt, sondern beispielsweise die Antwort auf eine Anfrage über HTTP von einer REST API verwendet, kann es durchaus vorkommen, dass die Objekte nicht der geforderten Struktur entsprechen. In dem Fall würde der Programmfluss keinen der beiden case-Blöcke durchlaufen und die Funktion somit nicht planmäßig über return verlassen. Da dieser Fall aus TypeScript-Sicht niemals auftreten kann, heißt der Typ, den action an der Stelle hat, folgerichtig never. Es handelt sich dabei um einen eigenen Typ, der aussagt, dass er eigentlich niemals zum Einsatz kommt. Eine Funktion, die never als Rückgabebetyp angibt, zeigt an, dass sie potenziell nicht zurückkehrt, weil sie beispielsweise einen Fehler wirft. Listing 11 erweitert die handleAction-Funktion um eine Fehlerbehandlung. Sollte die Anwendung ein Objekt erhalten, das die switch-Anweisung nicht behandelt, ruft sie die Fehlerbehandlungsfunktion auf. Interessant daran ist, dass auch das Argument der Funktion vom Typ never ist. Eine Ergänzung des Union Type in der handleAction-Funktion führt zu einem Fehler in der Fehlerbehandlungsfunktion, da der Typ an der Stelle nicht mehr never ist, sondern der unbehandelte neue Typ. Damit ist zur Entwicklungszeit sichergestellt, dass der Code alle Ausprägungen des Union Type behandelt. Zur Laufzeit erfolgt ebenfalls eine Fehlerbehandlung, falls die Funktion unerwartet ein unbekanntes Objekt als Parameter erhält.

## Hilfreich: Utility-Typen

TypeScript bringt einige generische Typen mit, die aus einem Typ, der ein Objekt beschreibt, einen anderen Typ erzeugen.

```
function addListener<O extends object>(o: O, propertyName: keyof O) {--
}

const syd = {
  firstname: "Syd",
  lastname: "Barrett",
};

addListener(syd, "");
  ▾ firstname      firstname
  ▾ lastname
```

### Autovervollständigung für Ausprägungen eines mit keyof erzeugten Union Type (Abb. 3)

Der neue kann beispielsweise dieselben Properties wie der Ausgangstyp haben, die aber beispielsweise alle als optional oder readonly gekennzeichnet sind. Listing 12 zeigt zwei Utility-Typen. Die Funktion patch soll ein Objekt vom Typ Person an einen Server schicken. Der Aufrufer soll allerdings in der Lage sein, nur die Teile des Person-Objekts zu übergeben, die er auf dem Server speichert. Der Typ des Parameters person muss somit eine Untermenge von Person sein. Damit Entwicklerinnen und Entwickler den neuen Typ nicht manuell definieren müssen, können sie mit Partial automatisch einen Typ erzeugen, der alle Properties des ursprünglichen Typs enthält, die aber alle als optional deklariert sind. Der Utility-Typ Readonly sorgt dafür, dass die patch-Funktion das übergebene Objekt darüber hinaus nicht verändern kann. Er deklariert alle Properties als readonly. Damit führt der schreibende Zugriff auf die Felder zu einem Compile-Fehler.

## Flexibel mit Generics

Generics kennt TypeScript ebenfalls. Die in Listing 13 gezeigte addListener-Funktion soll einem beliebigen Objekt einen Listener für eine ebenso beliebige Property hinzufügen. Dazu erwartet sie einerseits einen Parameter mit dem Objekt, der in generischer Form als Typ Argument in spitzen Klammern beschrieben ist. Der zweite Parameter erwartet den Namen einer Property aus diesem Objekt. Dazu dient der keyof-Operator von TypeScript, der einen Union Type (siehe Abbildung 3) zurückgibt, dessen Ausprägungen aus den Keys eines Objekts bestehen. Ein Beispiel dafür ist ebenfalls in Listing 13 zu sehen: Der Typ PersonKeys ist ein Union Type aus den Strings firstname, lastname und age, sodass eine Variable des Typs nur einen dieser drei Werte annehmen kann.

Die addListener-Funktion verwendet keyof, um sicherzustellen, dass der zweite Parameter ein String ist, dessen Wert einem der Keys entspricht, die in dem übergebenen Objekt enthalten sind. Wenn der Aufrufer ein Objekt übergibt und einen String, der nicht einem der Key-Namen entspricht, gibt es einen Compile-Fehler, und eine IDE kann Autovervollständigung für Aufrufe bieten.

Mit Generics und Utility-Typen lässt sich eine Vielzahl typischer JavaScript-Programmiermuster abbilden und typischer beschreiben. Sofern die Utility-Typen nicht ausreichend sind, existiert eine Art Meta-Sprache auf Ebene des Typsystems. Mit ihr lassen sich individuelle abgeleitete Typen erstellen, die in TypeScript Mapped Types heißen.

Listing 14 zeigt eine validate-Funktion, die ein beliebiges Objekt validieren soll. Sie soll ein Objekt zurückliefern, das genauso aussieht wie das übergebene Objekt, mit dem Unterschied, dass alle Felder zwar denselben Namen, aber den Typ boolean haben. Dessen Wert drückt aus, ob das Feld erfolgreich validiert werden konnte. Der ValidatedObject-Type beschreibt ein solches Objekt. Er iteriert über alle Properties in einem bestehenden Objekt, das als Typargument übergeben wird, und setzt in dem neuen Objekt den Typ aller Felder des Original-Objektes auf den Typ boolean.

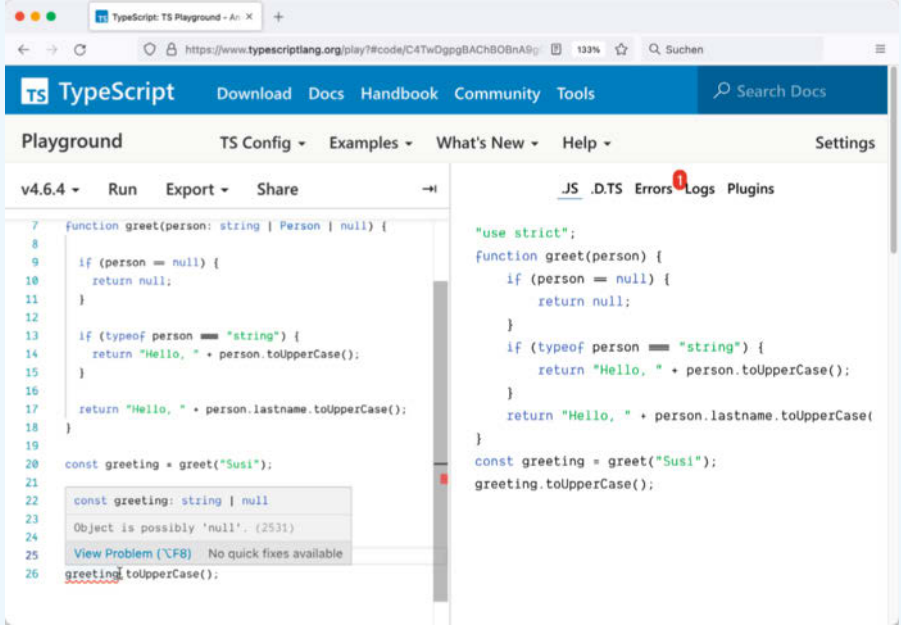
### Einbindung in das Java-Script-Ökosystem

Wer eine Anwendung mit TypeScript baut, muss potenziell Libraries verwenden, die in JavaScript implementiert sind. In solchen Fällen kommt einem eine wesentliche Designentscheidung von TypeScript zugute: Die Programmiersprache soll nicht nur mit TypeScript-, sondern auch mit JavaScript-Code zusammenarbeiten. Das bezieht sich ebenso auf eigenen JavaScript-Bestandscode wie auf JavaScript-Libraries.

Um die Bibliotheken typsicher mit TypeScript zu verwenden, ohne sie auf TypeScript umzustellen, lassen sich Typdeklarationen erstellen: TypeScript-Beschreibungen der API einer Bibliothek. Dazu gehören in erster Linie ihre exportierten Funktionen, Objekte und Klassen. Wenn eine Bibliothek die Deklarationen nicht bereitstellt, lassen sie sich extern pflegen und in einem eigenen Repository zur Verfügung stellen. Das Projekt Definitely Typed (siehe ix.de/zcry) bietet eine umfangreiche Auswahl von Typdeklarationen für gängige und weniger verbreitete JavaScript-Bibliotheken an. Sogar die offiziellen Typdeklarationen für React finden sich dort. Die Entwicklung und Pflege hat nicht das React-Team, sondern eine Community

## TypeScript ausprobieren

Um TypeScript ohne Installation auszuprobieren, bietet sich der TypeScript Playground an (der Link findet sich unter ix.de/zcry). Dabei handelt es sich um einen Online-TypeScript-Editor, der über das typische Tooling verfügt, insbesondere Autovervollständigung und die Ausgabe von Compile-Fehlern. Außerdem kann der Editor anzeigen, wie der Code aussehen würde, wenn man ihn nach JavaScript übersetzt.



**Der TypeScript Playground vermittelt einen ersten Eindruck der Eigenschaften von TypeScript (Abb. 4).**

```

Listing 14: Mapped Types

type ValidatedObject<O> = {
  [Key in keyof O]: boolean
}

function validate<O extends object>(object: O):
ValidatedObject<O> {
  // Implementierung ausgelassen
}

let validatedObject = validate({
  firstname: "Susi",
  lastname: "Meier",
});

let vF: boolean = validatedObject.firstname;
// OK: firstname ist jetzt boolean
let vL: string = validatedObject.lastname;
// Fehler: Type 'boolean' is not assignable to type 'string'
let vC: boolean = validatedObject.city;
// Fehler: Property 'city' does not exist on type 'ValidatedObject'
    
```

übernommen. Da weder Browser noch Node.js TypeScript-Code ausführen können, gilt es vor dem Start, den Code zu kompilieren. Der Compiler überprüft zunächst die korrekte Verwendung der Typen und erzeugt danach ausführbaren JavaScript-Code. Dabei entfernt er im Wesentlichen die Typnotationen, die nicht dem JavaScript-Sprachstandard entsprechen. Daher gibt es zur Laufzeit keinen Weg, in ähnlicher Weise auf die Typinformationen zuzugreifen. Etwas Ähnliches wie die Reflection-API von Java ist somit nicht verfügbar.

### Reif für den Einsatz

TypeScript erweitert JavaScript um ein Typsystem, mit dem Code ohne großen Aufwand typsicher wird. Die Flexibilität des Systems ermöglicht auch in komplexem Code die korrekte Typisierung. Viele typische Hindernisse bei der Arbeit mit JavaScript lassen sich damit schon zur Entwicklungszeit verhindern.

Nahezu alle gängigen IDEs, Editoren und Build-Tools aus dem JavaScript-Umfeld arbeiten mit TypeScript, und für die meisten JavaScript-Bibliotheken stehen Typdefinitionen zur Verfügung. Dem produktiven Einsatz von TypeScript im eigenen Projekt steht also nichts im Weg. (rme@ix.de)

### Quellen

Die Links zu den Listings, dem TypeScript Playground, dem Definitely-Typed-Projekt und Rod Johnsons Äußerungen über TypeScript finden sich unter ix.de/zcry



### Nils Hartmann

Nils Hartmann ist freiberuflicher Softwareentwickler und -architekt. Er unterstützt, berät und schult Teams bei der Arbeit mit Java, Spring, GraphQL, React und TypeScript.

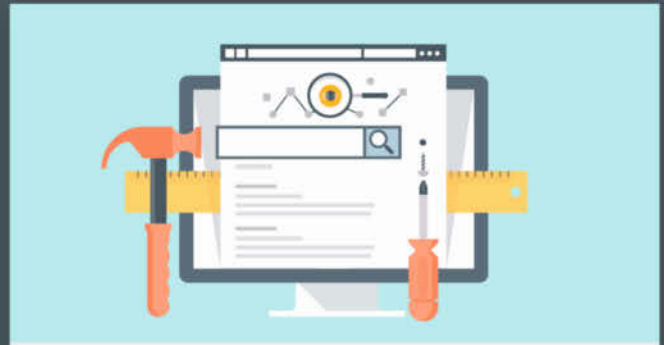


# WORKSHOPS 2022



**26. – 27. September 2022**

Terraform: Infrastructure as Code



**05. – 06. Oktober 2022**

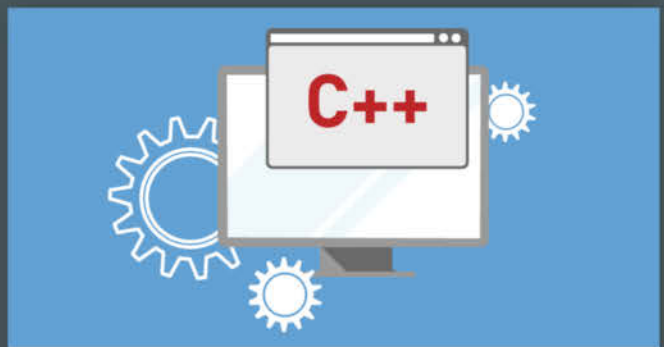
Moderne Desktop-Applikationen entwickeln mit WinUI 3



**Jenkins**

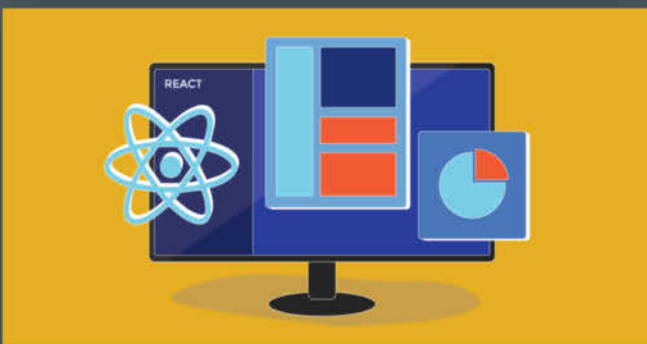
**17. – 18. Oktober 2022**

Continuous Integration mit Jenkins



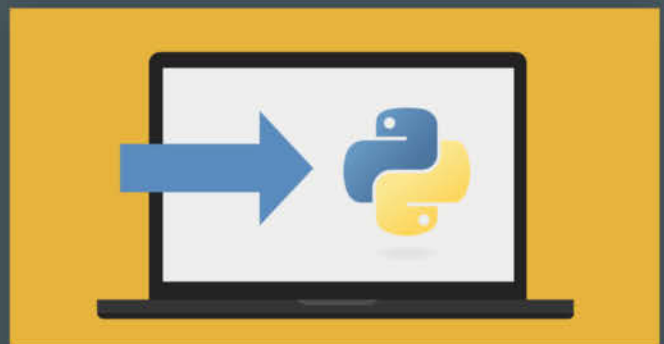
**25. – 27. Oktober 2022**

C++20: die Neuerungen umfassend erklärt



**02. – 04. November 2022**

Webanwendungen entwickeln mit React



**30. November – 02. Dezember 2022**

Python für Umsteiger von anderen Programmiersprachen