TypeScript for Svelte Developers

> *A focused course that teaches TypeScript through the lens of Svelte development — from basic types to advanced patterns used in real SvelteKit applications.*

Table of Contents

Chapter 1 — Why TypeScript?

1.1 The Problem TypeScript Solves

JavaScript is dynamically typed. This flexibility is powerful but introduces an entire class of bugs that only surface at runtime:

```
// JavaScript — no error until runtime
function greet(name) {
  return `Hello, ${name.toUpperCase()}!`;
}

greet(42); // Runtime error: name.toUpperCase is not a function
```

TypeScript catches this at **compile time**:

```
function greet(name: string): string {
  return `Hello, ${name.toUpperCase()}!`;
}

greet(42); // TS Error: Argument of type 'number' is not assignable to parameter of type
'string'
```

1.2 TypeScript and Svelte

SvelteKit has first-class TypeScript support:

- `.svelte` files support `<script lang="ts">`
- SvelteKit auto-generates types for load functions, actions, and params (`$types`)
- The Svelte language server provides full IntelliSense in editors

1.3 Setting Up

TypeScript is included by default when you create a new SvelteKit project:

```
npx sv create my-app
# Select "Yes, using TypeScript syntax" when prompted
```

1.4 The `tsconfig.json`

SvelteKit generates a `.svelte-kit/tsconfig.json` that your root `tsconfig.json` extends:

```
{
  "extends": "./.svelte-kit/tsconfig.json",
  "compilerOptions": {
    "strict": true
  }
}
```

Exercise 1

1. Create a new SvelteKit project with TypeScript enabled.
2. Open `tsconfig.json` and review the compiler options.
3. Introduce a deliberate type error in a `.ts` file and observe the editor feedback.

Chapter 2 — Basic Types

2.1 Primitive Types

```
let name: string = 'Ada';
let age: number = 36;
let isActive: boolean = true;
let nothing: null = null;
let notDefined: undefined = undefined;
```

2.2 Type Inference

TypeScript infers types when you provide an initial value:

```
let count = 0;        // inferred as number
let label = 'hello';  // inferred as string
```

2.3 Arrays

```
let numbers: number[] = [1, 2, 3];
let names: string[] = ['Ada', 'Grace'];

// Alternative syntax
let scores: Array<number> = [100, 95, 87];
```

2.4 Tuples

Fixed-length arrays with specific types at each position:

```
let pair: [string, number] = ['Ada', 36];
let rgb: [number, number, number] = [255, 128, 0];
```

2.5 Enums and Literal Types

```
// String literal union (preferred in Svelte)
type Status = 'idle' | 'loading' | 'success' | 'error';

// Enum (less common in Svelte codebases)
enum Direction {
  Up = 'UP',
  Down = 'DOWN',
  Left = 'LEFT',
  Right = 'RIGHT'
}
```

2.6 `any`, `unknown`, and `never`

```
let flexible: any = 'anything';      // Opt out of type checking (avoid!)
let safe: unknown = 'anything';      // Must narrow before use
let impossible: never;               // A value that can never exist
```

Exercise 2

1. Declare variables for each primitive type without explicit annotations — verify inference.
2. Create an array of objects and type it explicitly.
3. Write a function that accepts a `Status` literal union and returns a corresponding message.

Chapter 3 — Objects and Interfaces

3.1 Object Types

```
let user: { name: string; age: number; email?: string } = {
  name: 'Ada',
  age: 36
};
```

3.2 Interfaces

```
interface User {
  id: number;
  name: string;
  email: string;
  avatar?: string;
}

const user: User = {
  id: 1,
  name: 'Ada Lovelace',
  email: 'ada@example.com'
};
```

3.3 Type Aliases

```ts
type Point = {
  x: number;
  y: number;
};

type ID = string | number;
```

3.4 Interface vs Type — When to Use Which

Feature	`interface`	`type`
Object shapes	Yes	Yes
Extending	`extends`	`&`
Union / intersection	No	Yes
Declaration merging	Yes	No
Primitive aliases	No	Yes

General rule: Use `interface` for object shapes; use `type` for unions, intersections, and primitives.

3.5 Extending Interfaces

```ts
interface Person {
  name: string;
  age: number;
}

interface Employee extends Person {
  company: string;
  role: string;
}
```

3.6 Intersection Types

```ts
type Timestamped = {
  createdAt: Date;
  updatedAt: Date;
};

type Post = {
  title: string;
  body: string;
};

type TimestampedPost = Post & Timestamped;
```

3.7 Index Signatures

```ts
interface Dictionary {
  [key: string]: string;
}
```

```ts
const colors: Dictionary = {
  primary: '#3b82f6',
  secondary: '#10b981'
};
```

Exercise 3

1. Define a `BlogPost` interface with `title`, `slug`, `content`, `author`, `publishedAt`, and optional `tags`.
2. Extend `BlogPost` into a `DraftPost` that adds a `lastSavedAt` field.
3. Create a `type` alias for an API response: `{ data: T; error: string | null }`.

Chapter 4 — Functions

4.1 Typing Parameters and Return Values

```ts
function add(a: number, b: number): number {
  return a + b;
}

const multiply = (a: number, b: number): number => a * b;
```

4.2 Optional and Default Parameters

```ts
function greet(name: string, greeting: string = 'Hello'): string {
  return `${greeting}, ${name}!`;
}

function log(message: string, level?: string): void {
  console.log(`[${level ?? 'INFO'}] ${message}`);
}
```

4.3 Rest Parameters

```ts
function sum(...numbers: number[]): number {
  return numbers.reduce((total, n) => total + n, 0);
}
```

4.4 Function Overloads

```ts
function format(value: string): string;
function format(value: number): string;
function format(value: string | number): string {
  if (typeof value === 'string') return value.trim();
  return value.toFixed(2);
}
```

4.5 Function Types

```ts
type Comparator<T> = (a: T, b: T) => number;

const byAge: Comparator<{ age: number }> = (a, b) => a.age - b.age;
```

4.6 `void` vs `undefined`

```ts
function logMessage(msg: string): void {
  console.log(msg);
  // implicitly returns undefined — void means "ignore the return"
}
```

Exercise 4

1. Write a `filter` function that accepts an array and a predicate, both typed.
2. Create a typed `debounce` function.
3. Write a function with two overload signatures: one accepting a `string`, one accepting a `string[]`.

Chapter 5 — Union Types and Narrowing

5.1 Union Types

A value can be one of several types:

```ts
type Result = string | number;
type Status = 'idle' | 'loading' | 'success' | 'error';
```

5.2 Type Narrowing

TypeScript narrows types through control flow analysis:

```ts
function display(value: string | number) {
  if (typeof value === 'string') {
    // TypeScript knows value is string here
    console.log(value.toUpperCase());
  } else {
    // TypeScript knows value is number here
    console.log(value.toFixed(2));
  }
}
```

5.3 Discriminated Unions

The most powerful pattern for modelling state:

```ts
type LoadingState =
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'success'; data: string[] }
  | { status: 'error'; error: Error };

function render(state: LoadingState) {
  switch (state.status) {
    case 'idle':
```

```
      return 'Ready.';
    case 'loading':
      return 'Loading...';
    case 'success':
      return state.data.join(', '); // data is accessible
    case 'error':
      return state.error.message;   // error is accessible
  }
}
```

5.4 Type Guards

Custom type guard functions:

```
interface Fish { swim(): void; }
interface Bird { fly(): void; }

function isFish(animal: Fish | Bird): animal is Fish {
  return (animal as Fish).swim !== undefined;
}
```

5.5 The `in` Operator

```
function move(animal: Fish | Bird) {
  if ('swim' in animal) {
    animal.swim();
  } else {
    animal.fly();
  }
}
```

5.6 `satisfies` Operator

Validate that a value matches a type without widening it:

```
type Color = 'red' | 'green' | 'blue';
type Theme = Record<Color, string | number[]>;

const theme = {
  red: '#ff0000',
  green: [0, 255, 0],
  blue: '#0000ff'
} satisfies Theme;

// theme.green is still inferred as number[] (not string | number[])
theme.green.map(v => v * 2); // works!
```

Exercise 5

1. Model a `NetworkRequest<T>` discriminated union with `idle`, `loading`, `success`, and `error` variants.
2. Write a function that accepts `NetworkRequest<User[]>` and returns an appropriate message for each state.
3. Create a type guard function `isError` that narrows to the error variant.

Chapter 6 — Generics

6.1 Generic Functions

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

const str = identity('hello'); // type: string
const num = identity(42);      // type: number
```

6.2 Generic Interfaces

```
interface ApiResponse<T> {
  data: T;
  status: number;
  message: string;
}

const userResponse: ApiResponse<User> = {
  data: { id: 1, name: 'Ada', email: 'ada@example.com' },
  status: 200,
  message: 'OK'
};
```

6.3 Generic Constraints

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const user = { name: 'Ada', age: 36 };
getProperty(user, 'name'); // OK — returns string
getProperty(user, 'foo');  // Error: 'foo' is not assignable to 'name' | 'age'
```

6.4 Multiple Type Parameters

```
function mapEntries<K extends string, V, R>(
  record: Record<K, V>,
  fn: (value: V, key: K) => R
): Record<K, R> {
  const result = {} as Record<K, R>;
  for (const key in record) {
    result[key] = fn(record[key], key);
  }
  return result;
}
```

6.5 Generic Defaults

```ts
interface PaginatedResponse<T, M = { page: number; total: number }> {
  items: T[];
  meta: M;
}
```

Exercise 6

1. Write a generic `Stack<T>` class with `push`, `pop`, and `peek` methods.
2. Create a generic `Result<T, E = Error>` type that is either `{ ok: true; value: T }` or `{ ok: false; error: E }`.
3. Write a generic `groupBy<T>` function that groups an array by a key.

Chapter 7 — Utility Types

7.1 Built-in Utility Types

TypeScript provides many utility types out of the box:

```ts
interface User {
  id: number;
  name: string;
  email: string;
  avatar: string;
}
```

7.2 `Partial<T>`

Make all properties optional:

```ts
type UpdateUser = Partial<User>;
// { id?: number; name?: string; email?: string; avatar?: string }
```

7.3 `Required<T>`

Make all properties required:

```ts
type CompleteUser = Required<User>;
```

7.4 `Pick<T, K>` and `Omit<T, K>`

```ts
type UserPreview = Pick<User, 'id' | 'name'>;
// { id: number; name: string }

type UserWithoutAvatar = Omit<User, 'avatar'>;
// { id: number; name: string; email: string }
```

7.5 `Record<K, V>`

```ts
type UserRoles = Record<string, 'admin' | 'editor' | 'viewer'>;

const roles: UserRoles = {
  alice: 'admin',
```

```
  bob: 'editor'
};
```

7.6 `Exclude<T, U>` and `Extract<T, U>`

```
type Status = 'idle' | 'loading' | 'success' | 'error';

type ActiveStatus = Exclude<Status, 'idle'>;
// 'loading' | 'success' | 'error'

type CompletionStatus = Extract<Status, 'success' | 'error'>;
// 'success' | 'error'
```

7.7 `NonNullable<T>`

```
type MaybeString = string | null | undefined;
type DefiniteString = NonNullable<MaybeString>; // string
```

7.8 `ReturnType<T>` and `Parameters<T>`

```
function createUser(name: string, age: number) {
  return { id: Math.random(), name, age };
}

type NewUser = ReturnType<typeof createUser>;
// { id: number; name: string; age: number }

type CreateUserParams = Parameters<typeof createUser>;
// [name: string, age: number]
```

7.9 `Awaited<T>`

Unwrap a `Promise`:

```
type ResolvedData = Awaited<Promise<User[]>>; // User[]
```

Exercise 7

1. Given a `Product` interface, create a `ProductFormData` type using `Omit` (remove `id` and `createdAt`).
2. Create an `UpdateProduct` type using `Partial` and `Pick` (partial fields except `id` which is required).
3. Use `Record` to type a configuration object mapping feature flags to booleans.

Chapter 8 — TypeScript in Svelte Components

8.1 Typed Components

```
<script lang="ts">
  interface Props {
    title: string;
    description?: string;
    variant: 'primary' | 'secondary' | 'danger';
```

```
    onclick?: (event: MouseEvent) => void;
  }

  let { title, description, variant = 'primary', onclick }: Props = $props();
</script>

<div class="alert alert-{variant}" role="alert">
  <h3>{title}</h3>
  {#if description}
    <p>{description}</p>
  {/if}
  {#if onclick}
    <button onclick={onclick}>Dismiss</button>
  {/if}
</div>
```

8.2 Typed `$state` and `$derived`

```
<script lang="ts">
  interface Todo {
    id: number;
    text: string;
    done: boolean;
  }

  let todos: Todo[] = $state([]);
  let remaining = $derived(todos.filter(t => !t.done).length);
</script>
```

8.3 Typed Snippets

```
<script lang="ts">
  import type { Snippet } from 'svelte';

  interface Props {
    header: Snippet;
    row: Snippet<[item: Item]>;
    children: Snippet;
  }

  let { header, row, children }: Props = $props();
</script>
```

8.4 Typed Context

```
// src/lib/context.ts
import { getContext, setContext } from 'svelte';

interface AppContext {
  user: User;
  theme: 'light' | 'dark';
}
```

```ts
const KEY = Symbol('app-context');

export function setAppContext(ctx: AppContext) {
  setContext(KEY, ctx);
}

export function getAppContext(): AppContext {
  return getContext<AppContext>(KEY);
}
```

8.5 Typing Event Handlers

```svelte
<script lang="ts">
  function handleSubmit(event: SubmitEvent) {
    event.preventDefault();
    const formData = new FormData(event.currentTarget as HTMLFormElement);
    // ...
  }
</script>

<form onsubmit={handleSubmit}>
  <!-- ... -->
</form>
```

8.6 SvelteKit Auto-Generated Types

SvelteKit generates `$types` for every route:

```ts
// src/routes/blog/[slug]/+page.server.ts
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async ({ params }) => {
  // params.slug is typed as string
  const post = await getPost(params.slug);
  return { post };
};
```

```svelte
<!-- src/routes/blog/[slug]/+page.svelte -->
<script lang="ts">
  import type { PageData } from './$types';

  let { data }: { data: PageData } = $props();
  // data.post is fully typed
</script>
```

Exercise 8

1. Convert a JavaScript Svelte component to TypeScript with a fully typed `Props` interface.
2. Create a typed `createTodoStore` function in a `.svelte.ts` file.
3. Set up typed context for a theme provider.

Chapter 9 — Advanced Patterns

9.1 Mapped Types

```
type Readonly<T> = {
  readonly [K in keyof T]: T[K];
};

type Optional<T> = {
  [K in keyof T]?: T[K];
};

type Nullable<T> = {
  [K in keyof T]: T[K] | null;
};
```

9.2 Conditional Types

```
type IsString<T> = T extends string ? true : false;

type A = IsString<'hello'>; // true
type B = IsString<42>;      // false
```

9.3 Template Literal Types

```
type EventName = 'click' | 'focus' | 'blur';
type Handler = `on${Capitalize<EventName>}`;
// 'onClick' | 'onFocus' | 'onBlur'
```

9.4 `infer` Keyword

Extract types from within other types:

```
type UnwrapPromise<T> = T extends Promise<infer U> ? U : T;

type A = UnwrapPromise<Promise<string>>; // string
type B = UnwrapPromise<number>;          // number
```

9.5 Type-Safe Event Emitters

```
type EventMap = {
  'user:login': { userId: string };
  'user:logout': undefined;
  'notification': { message: string; level: 'info' | 'error' };
};

class TypedEmitter<T extends Record<string, unknown>> {
  private handlers = new Map<keyof T, Set<Function>>();

  on<K extends keyof T>(event: K, handler: (payload: T[K]) => void) {
    if (!this.handlers.has(event)) this.handlers.set(event, new Set());
    this.handlers.get(event)!.add(handler);
  }
```

```
  emit<K extends keyof T>(event: K, payload: T[K]) {
    this.handlers.get(event)?.forEach(fn => fn(payload));
  }
}

const emitter = new TypedEmitter<EventMap>();
emitter.on('user:login', ({ userId }) => console.log(userId)); // typed!
```

9.6 Builder Pattern with Types

```
class QueryBuilder<T> {
  private filters: Partial<T> = {};

  where<K extends keyof T>(key: K, value: T[K]): this {
    this.filters[key] = value;
    return this;
  }

  build(): Partial<T> {
    return { ...this.filters };
  }
}

const query = new QueryBuilder<User>()
  .where('name', 'Ada')
  .where('age', 36)
  .build();
```

9.7 Branded Types

Prevent accidental value mixing:

```
type UserId = string & { __brand: 'UserId' };
type PostId = string & { __brand: 'PostId' };

function createUserId(id: string): UserId {
  return id as UserId;
}

function getUser(id: UserId): User { /* ... */ }

const userId = createUserId('abc123');
const postId = 'xyz789' as PostId;

getUser(userId); // OK
getUser(postId); // Error!
```

Exercise 9

1. Create a mapped type `FormFields<T>` that transforms each property of `T` into `{ value: T[K]; error: string | null }`.
2. Write a conditional type `ArrayElement<T>` that extracts the element type from an array.
3. Implement a type-safe `createStore` function using generics and mapped types.

Chapter 10 — Real-World Exercises

Exercise 10.1 — Typed API Client

Build a fully typed API client for a REST API:

```
interface ApiClient {
  get<T>(url: string): Promise<ApiResponse<T>>;
  post<T>(url: string, body: unknown): Promise<ApiResponse<T>>;
  put<T>(url: string, body: unknown): Promise<ApiResponse<T>>;
  delete(url: string): Promise<ApiResponse<void>>;
}
```

Requirements:

- Type the request and response for each endpoint
- Handle errors with a discriminated union result type
- Create typed endpoint definitions

Exercise 10.2 — Typed Form Validation

Create a type-safe form validation library:

```
const schema = createSchema({
  name: [required(), minLength(2)],
  email: [required(), email()],
  age: [required(), min(18), max(120)]
});

const result = validate(schema, formData);
// result is typed: { name: string; email: string; age: number } | ValidationErrors
```

Requirements:

- Validators are composable functions
- The output type is inferred from the schema
- Error messages are typed per field

Exercise 10.3 — SvelteKit Blog with Full Types

Build a blog application with:

```
src/
├── lib/
│   ├── types.ts              # Shared type definitions
│   ├── api.ts                # Typed API functions
│   └── components/
│       ├── PostCard.svelte    # Typed props
│       └── CommentForm.svelte # Typed form
├── routes/
│   ├── +layout.server.ts     # Typed layout load
│   ├── blog/
│   │   ├── +page.server.ts   # Typed list load
│   │   └── [slug]/
│   │       ├── +page.server.ts  # Typed detail load
│   │       └── +page.svelte     # Typed page props
```

Requirements:

- All load functions and actions fully typed
- Shared types in `$lib/types.ts`
- Components with typed props interfaces
- Form actions with typed validation

Exercise 10.4 — Type Challenges

Solve these progressively harder type challenges:

1. **Easy:** Create a type `First<T>` that extracts the first element type from an array type.
2. **Medium:** Create a type `DeepPartial<T>` that makes all nested properties optional.
3. **Medium:** Create a type `PathKeys<T>` that generates all dot-notation paths for an object type.
4. **Hard:** Create a type `FromEntries<T>` that converts a tuple of `[key, value]` pairs into an object type.

```
// Example solutions to verify:

type First<T extends unknown[]> = T extends [infer F, ...unknown[]] ? F : never;
type A = First<[string, number, boolean]>; // string

type DeepPartial<T> = {
  [K in keyof T]?: T[K] extends object ? DeepPartial<T[K]> : T[K];
};
```

Appendix — TypeScript Cheat Sheet for Svelte

Common Patterns

```
// Props
interface Props {
  required: string;
  optional?: number;
  withDefault: boolean; // use default in destructuring
  union: 'a' | 'b' | 'c';
  callback: (value: string) => void;
  children: Snippet;
  namedSnippet: Snippet<[item: Item]>;
}

// Reactive state
let count: number = $state(0);
let items: Item[] = $state([]);
let selected: Item | null = $state(null);

// Derived
let total: number = $derived(items.length);

// Load function
import type { PageServerLoad } from './$types';
export const load: PageServerLoad = async ({ params, fetch }) => { ... };

// Form action
```

```
import type { Actions } from './$types';
export const actions: Actions = { default: async ({ request }) => { ... } };
```

Congratulations! You now have a solid TypeScript foundation tailored for Svelte development. Apply these patterns in your SvelteKit projects for safer, more maintainable code.