TypeScript Basics: Consolidated Guide

1. Type Annotations

TypeScript allows you to explicitly define the type of variables, function parameters, and return values. These types ensure that values are consistent and help catch errors during development.

Basic Types

- **String**: Represents text.

  ```typescript
  let name: string = "Alice";
  ```

- **Number**: Represents numeric values.

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

- **Boolean**: Represents true/false values.

  ```typescript
  let isAvailable: boolean = true;
  ```

- **Array**: Collection of values of a specific type.

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

- **Tuple**: Fixed-size array with specific types at each index.

  ```typescript
  let user: [string, number] = ["Alice", 30];
```

  ```

- **Enum**: Represents a set of named constants.

  ```typescript

  enum Color { Red, Green, Blue }

  let color: Color = Color.Green; // Outputs 1

  ```

---

2. Type Inference and Type Assertions

TypeScript can infer types based on the assigned value, reducing the need for explicit type annotations.

Example: Type Inference

```typescript

let city = "Lagos"; // Inferred as a string

let count = 10;     // Inferred as a number

```

Type Assertions

Type assertions override inferred types when you're sure of a variable's type.

```typescript

let value: any = "Hello, TypeScript!";

let stringLength: number = (value as string).length;

```

```

3. Union and Intersection Types

Union Types

Union types allow variables to have multiple possible types.

```typescript
let result: string | number;
result = "Passed";  // OK
result = 100;       // OK
```

Intersection Types

Intersection types combine multiple types into one.

```typescript
interface A { id: number; }
interface B { name: string; }
type C = A & B; // Must have both id and name
```

---

4. Literal Types and Type Aliases

Literal Types

Restrict variables to specific, predefined values.

```typescript
let trafficLight: "red" | "yellow" | "green";
trafficLight = "red"; // Valid
```

Type Aliases

Simplify complex types with aliases.

```typescript
type Status = "active" | "inactive" | "suspended";
let userStatus: Status = "active";
```

---

5. Nullable Types and Null Handling

Handle null and undefined explicitly using union types and strict null checks.

Example: Nullable Types

```typescript
let username: string | null = null;
username = "John"; // Allowed
```

Optional Chaining and Nullish Coalescing

```typescript
let user = { address: { city: "Lagos" } };
console.log(user?.address?.city ?? "Unknown City");
```

This document provides a detailed explanation of TypeScript basics with examples for easy understanding. For further insights into advanced concepts, refer to the complete guide.