* **What is TypeScript, and how does it differ from JavaScript?**

Typescript is a strongly typed programming language that build on javascript, adding features like static typing, type interface, and interfaces to improve code quality and development efficiency.

It provides compile-time error detection, ensuring that many issues are caught during development rather than at runtime.

Typescript introduces static types, allowing developers to explicitly define the type of variables, function parameters, and return values, reducing unexpected behavior.

It supports modern javascript features and ensures compatibility by compiling to plan javascript for execution in any environment.

With typescript, we get access to better tooling in IDEs, such as autocompletion, intelligent code navigation, and scalability, making it ideal for large and complex projects.

Typescript improves code maintainability and scalability, making it ideal for large and complex projects.

It enables to use of advanced features like generics, enum and interfaces, promoting reusable and modular code.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Typescript** | **Javascript** |
| Definition | Typescript is a superset of javascript that adds static typing and ad | Javascript is a lightweight , interpreted language for web development |
| Typing | Uses static typing where types are defined at compile tim, reducing runtime issues | Uses dynamic typing determining types at runtim |
| Error Detection | Error are detected at compile tim | Error are detected at runtime, which can lead to unexpected behavior |
| Syntax | Adds features like interfaces, enums, and type annotations. | Uses a simpler syntax without support for static types / interfaces. |
| Compatibility | Compiles down to plain Javascript, ensuring it runs in any javascript environment. | Runs directly in the browser / server without the need for compilation. |
| Tooling support | Provides better IDE support with features like autocompletion, refactoring, and inline error checking. | Limited IDE support compared to Typescript unless combined with tools like ESLint. |
| Learning Curve | It need learning additional syntax and type related concepts. | Easier to learn and use for beginners due to its flexibility and simplicity. |
| Code scalability | More suitable for large scale projects due to maintainable and reusable code structures. | Can become difficult to manage as the codebase grows, especially in large projects. |
| Community and libraries | Has growing community support, and many javascript libraries offer typescript typings. | Has an established and large community with widespread library support. |
| Project configuration | Requires setup with tsconfig.json for project-specific settings. | Does not requires any configuration; ready to use out of the box. |

* **Explain the pros and cons of TypeScript.**

Typescript key goals are

* + To statically identify constructs that are likely to cause errors.
  + To provide a structuring mechanism for larger codebase.

**Pros of typescript**

* **Static Typing**

Helps catch errors during development, reducing bugs and improving reliability.

Enforces type safety by defining types for variables, function parameters, and return values.

* **Improved Code Maintainability**

Easier to understand and refactor code in large projects due to clearly defined types and structures.

Reduces ambiguity in code, making it easier for teams to collaborate.

* **Better IDE Support**

Provides features like autocompletion, type hints, and real-time error detection in popular editors like VS Code.

Enhances productivity with intelligent tools for navigation and debugging.

* **Compatibility with JavaScript**

Allows gradual adoption by letting developers use existing JavaScript code with TypeScript.

Compiles down to plain JavaScript, making it compatible with all JavaScript environments.

* **Modern JavaScript Features**

Supports ES6/ESNext features, ensuring future-proof code.

Offers features like async/await, decorators, and modules even on older environments.

* **Reusability with Generics**

Generics allow developers to write reusable and type-safe code, improving flexibility and reducing duplication.

* **Scalability**

Ideal for large-scale applications with multiple developers, as the codebase remains organized and less prone to errors.

* **Integration with Libraries and Frameworks**

Many popular libraries (like React, Angular, and Vue) have built-in or community-supported TypeScript definitions

Cons of typescript

* **Learning Curve**

Requires additional time to learn new concepts like static typing, generics, and interfaces, especially for JavaScript developers.

* **Compilation Overhead**

TypeScript needs to be compiled into JavaScript, adding an extra step to the development process.

* **Verbose Syntax**

Code can become more verbose due to type annotations, which might be cumbersome for small projects.

* **Configuration Required**

Requires setting up tsconfig.json and other tools, which can be intimidating for beginners.

* **Library Compatibility Issues**

Some JavaScript libraries may not have TypeScript definitions, requiring developers to write or import type declarations.

* **Increased Development Time**

Enforcing strict typing can slow down development for smaller projects or rapid prototyping.

* **Runtime Type Issues**

TypeScript only checks types at compile time and cannot enforce type safety at runtime.

* **Community Size Compared to JavaScript**

While growing, TypeScript community and ecosystem are smaller compared to JavaScripts

* **What are types and interfaces in TypeScript, and how do they differ?**

Types

* + A type in typescript is used to define the shape, structure, behaviour of data.
  + It can describe primitives, object, arrays, functions, even union and intersection types.
  + It is versatile and supports complex type definitions.

Interfaces

* Interfaces is used to define the structure of object, primarily for object oriented design.
* It allows creating reusable and extensible blueprint for objects and classes.
* It is ideal for modelling the shape of object / a class.

|  |  |  |
| --- | --- | --- |
| Feature | Type | Interfaces |
| Declaration | We can define primitives, unions, intersection, arrays, and objects. | Used mainly for defining the shape of object / class. |
| Extensibility | Extended using intersection types & | Extended using extends |
| Implementation | Cant be implemented by a class | Can be implemented by a class |
| Reopening / Merging | Types cant be reopened / merged after being defined. | Interfaces can be reopened and merged to add new properties. |
| Complex types | Supports unions, intersections, and conditional types. | Does not support unions / conditional types directly. |
| Function declaration | Can define functions as types | Function can only be declared using callable signatures. |
| Usage | Best for complex type definitions, utility types, and scenarios needs unions | Best for defining the shape of objects, classes, and API contracts. |
| Performance | Slightly slower during compilation due to additional flexibility | Slightly faster during compilations for object structure. |
| Literal use | Allows defining type aliases for literal values. | inactive |
| Recommended use | Use when we need advanced features like unions, intersections, | Use when defining object shapes / creating reusable and extendable contract. |

* **What are Generics in TypeScript, and why are they useful?**

Generics in typescript allow to create reusable, type-safe components, function, classes that can work with a variety of types instead of being restricted to a specific one. They act as placeholders for types, which are provided when the component, function, classes is used.

* 1. Reusability
     + Generics allow to write code that works with different data types without duplication logic.
     + A generic function to handle arrays of any type.
  2. Type safety
     + Generics ensure that operations are performed on data of the expected type, reducing runtime error.
     + A generic function that enforces type constraints.
  3. Flexibility
     + Instead of hardcoding types, generics allow the caller to specify the desired type, making the code adaptable to various use cases.
     + A generic stack class for any type of data.
  4. Scalability

Generics help build scalable systems by avoiding redundant code and reducing the complexity of maintaining multiple implementation for different types.

* 1. Efficiency

Reduces redundancy and simplifies maintenance.

* 1. Error prevention

Catch mismatched types during compile tim instead of runtime

* 1. Consistency

Enforces type uniformity across operations.

* **What are Enums, and how are they used in TypeScript?**

Enums in typescript are a way to define a set of named constants. They provide a convenient way to organize and group related values, often representing a collection of options or a fixed set of possible states.

Enums can be numeric. String-based, or even a combination of both.

Enums are used because they improve

* 1. Readability

They make the code more descriptive and meaningful compared to using raw numbers or strings.

* 1. Type safety

Enums help ensure only valid values are used in the code, reducing runtime errors.

* 1. Reusability

They create reusable constants, especially for states, roles, or options.

Cons of enums

1. Additional ode after compilation

Enums are dont a native javascript feature and generate extra code during compilation.

1. Less flexible than literal types

String literal unions “A”|”B” can often replace enums with simpler syntax.

* **What is the purpose of readonly in TypeScript?**

The readonly modifier in typescript is used to mark properties of object or class members as immutable once they are initialized.this ensures that their values cant be reassigned after their initial assignment, enhancing the immutability and type safety of the code.

* 1. Immutable properties

Properties marked as readonly can be set only once at the tim of declaration, initialization, or in the constructor.

* 1. Compile tim enforcement

Any attempt to modify a readonly property after initialization will result in a compile tim error.

* 1. Data integrity

Prevent mutation of properties or array in APIs, ensuring stability and predictability.

* 1. Improved readability

Includes the intent of immutability, making code easier to understand and maintain.

|  |  |  |
| --- | --- | --- |
| Feature | readonly | const |
| Scope | Used with properties in objects, classes, or arrays. | Used with variables |
| Initialization | Property can be initialized in the declaration or constructor | Variable must be initialized at declaration |
| Reassignment | Property cant be reassigned after initialization | Variable cant be reassigned |
| Object / Arrays | Ensures immutability of the property reference | Ensures immutability of the variable reference. |

* **What are utility types in TypeScript, such as Pick and Partial?**

Utility types in typescript are predefined generic types that help manipulate or transform types in a concise and reusable way. These utility types simplify common tasks, such as creating optional properties, picking subsets of properties, or making types read-only.

**Code reusability**

Avoid repetitive type definitions.

**Flexibility**

Easily modify and manipulate existing types to meet specific needs.

**Type safety**

It retain strong typing while performing complex type transformations.

* 1. Partial<type>
     + It converts all properties of a given type to optional
     + It is useful for scenarios like updating objects where not all properties need to be present.

interface User {

  id: number;

  name: string;

  email: string;

}

type PartialUser = Partial<User>;

const updateUser: PartialUser = { name: "john" };

* 1. Pick<type, keys>
     + It creates a new type by picking a subset of properties from existing type.
     + Useful for extracting fields from a complex type.

interface User {

  id: number;

  name: string;

  email: string;

  age: number;

}

type PickedUser = Pick<User, "id" | "email">;

const updateUser: PickedUser = { id: 1, email: "john.doe@gmail.com" };

* 1. Omit<type, keys>
     + It creates a new type by omitting specific properties from existing type.
     + It is useful for excluding unnecessary fields.

interface User {

  id: number;

  name: string;

  email: string;

  age: number;

}

type OmitedUser = Omit<User, "email" | "age">;

const updateUser: OmitedUser = { id: 1, name: "john" };

* 1. Readonly<type>
     + It makes all properties of a type read only. It ensures immutability.

interface User {

  id: number;

  name: string;

  email: string;

  age: number;

}

type ReadonlyUser = Readonly<User>;

const user: ReadonlyUser = {

  id: 1,

  name: "john",

  email: "john.doe@gamil.com",

  age: 20,

};

// user.id = 2; // Error: Cannot assign to 'id' because it is a read-only property

* 1. Record<keys, type>
     + Constructs object type where keys are of a specific type, and values are of another specific type.
     + It is useful for defining mappings or dictionaries.

type Role = "admin" | "user" | "moderator";

type RolePermissions = Record<Role, string[]>;

const permissions: RolePermissions = {

  admin: ["read", "write", "delete"],

  user: ["read"],

  moderator: ["read", "write"],

};

* **Explain how to configure a TypeScript project using tsconfig.json.**
* **How does TypeScript handle modules and namespaces?**
* **What are type guards, and how are they implemented in TypeScript**