Q1: What is TypeScript and how does it enhance JavaScript?

TypeScript is a superset of JavaScript that introduces optional static typing, classes, and interfaces. It enhances code editor support for error detection, making JavaScript more maintainable and scalable.

TypeScript code compiles down to pure JavaScript, allowing it to run in any environment that supports JavaScript.

Q2: What are the three components of TypeScript?

The three main components of TypeScript are:

- The language itself
- The Compiler
- The Language Service

These components work together to provide a comprehensive development experience, enhancing productivity and code quality.

Q3: What is the difference between JavaScript and TypeScript?

The main difference is that TypeScript adds optional static typing and advanced language features such as interfaces, while JavaScript is dynamically typed. TypeScript is a superset of JavaScript, providing better tooling, earlier error detection, and improved maintainability for large codebases.

This allows for better tooling, earlier error detection, and improved maintainability for large codebases.

Q4: What is static typing in TypeScript?

Static typing in TypeScript means that variable types are declared and enforced at compile time. This allows for early detection of type-related errors and improves code maintainability.

```
let name: string = "John";
let age: number = 30;
let isStudent: boolean = true;
```

Q5: What are some user-defined data types in TypeScript?

TypeScript allows developers to create custom data types. Some user-defined data types include:

- Enumerations (sets of constants)
- Classes
- Interfaces
- Arrays
- Tuples (arrays with a fixed number of elements)

These types provide more structure and type safety to your code.

Q6: What are some built-in data types in TypeScript?

TypeScript includes several built-in data types commonly used in programming. These include:

- Number
- String
- Boolean
- Null
- Undefined

- Void
- Symbol
- BigInt

These types form the foundation for more complex type structures in TypeScript.

Q7: What is an interface in TypeScript?

An interface in TypeScript is a contract for a structure. It defines the shape of an object, specifying the properties and methods it should have.

```
interface Person {
 name: string;
 age: number;
 greet: () => void;
}

const Greeting = ({ name, age, greet }: Person) => {
 return (
<div>
<h1>{name}</h1>
<p>Age: {age}</p>
<button onClick={greet}>Say Hello</button>
</div>
);
};

const App = () => {
 const person: Person = {
 name: "Alice",
 age: 30,
 greet: () => alert("Hello!")
};

 return <Greeting {...person} />;
};
```

In this example, the Person interface defines the structure, and the Greeting component uses it as a type for its props. The App component creates an object that adheres to the Person interface.

Q8: What are access modifiers in TypeScript?

Access modifiers in TypeScript control the visibility and accessibility of class members. The three main access modifiers are:

- Public (accessible to all)
- Private (accessible only within the declaring class)
- Protected (accessible within declaring class and its subclasses)

These modifiers help in implementing encapsulation, a key principle of object-oriented programming.

Q9: What is the 'any' type in TypeScript and how is it used?

The 'any' type in TypeScript represents any JavaScript value. It's used when we don't know the variable type in advance.

```
const employee: any = JSON.parse(employeeData);
```

While 'any' provides flexibility, it should be used sparingly as it bypasses TypeScript's type checking.

Q10: When is the 'void' type used in TypeScript?

The 'void' type in TypeScript represents the absence of a type. It's typically used in functions that don't return a value.

```
const notify = (): void => {
 alert('Hi');
};
```

This type can only be assigned null or undefined, making it useful for indicating that a function performs an action but doesn't produce a result.

Q11: What is the 'unknown' type in TypeScript and how does it differ from 'any'?

The 'unknown' type is a type-safe counterpart of 'any'. Anything can be assigned to 'unknown', but 'unknown' isn't assignable to anything but itself and 'any' without type assertion or control flow based narrowing.

```
let foo: unknown = 'xd';
// let bar: string = foo; // Error: Type 'unknown' is not assignable to type 'string'
```

This makes 'unknown' safer to use than 'any' when the type is truly unknown.

Q12: How are generic types implemented in TypeScript?

Generic types in TypeScript allow creating reusable code components that work with different types. They provide a way to make components type-safe for any type that is provided.

```
const identity = <T>(arg: T): T => {
 return arg;
};

const result = identity<string>("Hello, TypeScript");
console.log(result); // Output: Hello, TypeScript
```

This function can work with any type, preserving type information throughout the function. Generics are particularly useful for creating flexible, reusable functions and classes.

Q13: What is a module in TypeScript?

A module in TypeScript is a container for related code, enhancing maintainability. It can contain classes, interfaces, functions, and variables.

```
export class MathOperations {
 static add(a: number, b: number): number {
 return a + b;
}
}
```

Modules help in organizing code and controlling the scope of declarations.

Q14: How do arrays work in TypeScript?

Arrays in TypeScript store multiple values of a specified type. They can be declared using type annotations or type inference.

```
let values: number[] = [1, 2, 3, 4];
```

TypeScript provides type checking for arrays, ensuring that only elements of the specified type are added.

Q15: What are primitive types in TypeScript?

Primitive types in TypeScript are the most basic data types available. They include:

- Number
- String
- Boolean
- Null
- Undefined
- Symbol
- BigInt

These types are immutable and are passed by value rather than by reference.

Q16: Demonstrate the syntax for a function with type annotations in TypeScript.

In TypeScript, you can add type annotations to function parameters and return values. This helps in specifying the expected types for inputs and outputs, improving code clarity and catching type-related errors early.

```
const greet = (name: string): string => {
 return 'Hello, ' + name;
};

const result = greet("TypeScript");
console.log(result); // Output: Hello, TypeScript
```

In this example, the function expects a string parameter and returns a string value. TypeScript will enforce these type constraints, helping prevent type-related bugs.

Q17: How are objects created in TypeScript?

In TypeScript, objects can be created using object literals. The object type refers to any value with properties.

```
let point: { x: number, y: number } = { x: 10, y: 20 };
```

TypeScript allows you to define the shape of objects using interfaces or type aliases for more complex structures.

Q18: Does TypeScript support all object-oriented principles?

Yes, TypeScript supports all four main principles of object-oriented programming:

- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

These principles allow for creating more organized, reusable, and maintainable code in TypeScript.

Q19: How are class constants defined in TypeScript?

In TypeScript, class constants are defined using the 'readonly' modifier. This ensures that the value cannot be changed after initialization.

```
class Example {
 readonly constant: string = 'constant';
}
```

The 'readonly' modifier can be used for properties in interfaces as well.

Q20: Demonstrate function overloading in TypeScript with an example.

Function overloading in TypeScript allows defining multiple function signatures with the same name but different parameter types and/or return types. The implementation must be compatible with all signatures.

```
const add = (a: number | string, b: number | string): number | string => {
 if (typeof a === 'number' && typeof b === 'number') {
 return a + b;
}
 if (typeof a === 'string' && typeof b === 'string') {
 return a + b;
}
 throw new Error('Invalid arguments');
};

console.log(add(5, 10));        // Output: 15
console.log(add('Hello, ', 'TypeScript')); // Output: Hello, TypeScript
```

This approach uses a single function with union types for parameters and return value, providing type safety and flexibility. It allows the function to handle different types of inputs while maintaining clear type information.

Q21: What is a Tuple in TypeScript?

A Tuple in TypeScript is a type that allows expressing an array where the type of certain elements is known. It's useful when you want an array with a fixed number of elements where each element may be of a different type.

```
let x: [string, number] = ['hello', 10];
```

Tuples provide more type safety than regular arrays when working with fixed-length, mixed-type collections.

Q22: What is an Enum in TypeScript?

An Enum in TypeScript is a way of giving friendly names to sets of numeric values. It allows for defining a set of named constants.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

Enums make it easier to document intent or create a set of distinct cases.

Q23: Explain the 'never' type in TypeScript and provide an example of its use.

The 'never' type in TypeScript represents the type of values that never occur. It's often used for functions that never return or always throw an error.

```
const throwError = (message: string): never => {
 throw new Error(message);
};

const infiniteLoop = (): never => {
 while (true) {}
};
```

The 'never' type is a subtype of every type, but no type is a subtype of 'never' (except 'never' itself). It's useful for representing impossible cases in type narrowing scenarios.

Q24: How does type assertion work in TypeScript?

Type assertion in TypeScript is a way to tell the compiler to treat a value as a specific type. It's used when you have information about the type of a value that TypeScript can't infer on its own.

```
const someValue: any = "This is a string";
const strLength: number = (someValue as string).length;

// Alternative syntax using angle brackets
const altStrLength: number = (<string>someValue).length;
```

Type assertions don't perform any special checking or restructuring of data. They're simply a way to tell the TypeScript compiler to trust your judgment about a type.

Q25: How to define a class in TypeScript?

Classes in TypeScript are fundamental entities used to create reusable components. They include properties and methods, providing a way to structure and organize code.

```
class Person {
 constructor(public name: string) { }

 greet() {
 console.log('Hello, my name is ' + this.name);
}
}
```

This example defines a Person class with a constructor and a method. The 'public' keyword in the constructor automatically creates and initializes a name property.

Q26: How do optional parameters work in TypeScript functions?

In TypeScript, parameters can be made optional by appending a '?' to the end of the parameter name. Optional parameters must follow required parameters in the function signature.

```
const buildName = (firstName: string, lastName?: string): string => {
 if (lastName) {
 return firstName + ' ' + lastName;
}
 return firstName;
};

console.log(buildName("John"));        // Output: John
console.log(buildName("John", "Doe")); // Output: John Doe
```

This feature allows functions to be called with fewer arguments than they're defined to accept. It's particularly useful when certain parameters aren't always needed.

Q27: How do default parameters work in TypeScript?

Default parameters in TypeScript are parameters that are given a default value in the function declaration. They come after all required parameters and are automatically optional.

```
const buildName = (firstName: string, lastName = 'Smith'): string => {
 return firstName + ' ' + lastName;
};

console.log(buildName('John'));     // Output: John Smith
console.log(buildName('Jane', 'Doe')); // Output: Jane Doe
```

In this example, 'Smith' is the default value for lastName. If no second argument is provided, 'Smith' will be used. This feature allows for more flexible function calls and can help reduce boilerplate code.

Q28: How do union types work in TypeScript?

Union types in TypeScript allow a variable to hold values of multiple types. They are declared using the '|' symbol between the types.

```
let value: string | number;
value = 'Hello'; // Valid
value = 42;      // Also valid
// value = true; // Error: Type 'boolean' is not assignable to type 'string | number'
```

This feature provides flexibility when a value could be of more than one type, which is common in dynamic programming scenarios.

Q29: How do interfaces with optional properties work in TypeScript?

In TypeScript, interfaces can include optional properties, which are marked by appending a '?' to the property name. This allows for flexibility in object shapes while still maintaining type safety.

```
interface SquareConfig {
 color?: string;
 width?: number;
}

const createSquare = (config: SquareConfig): { color: string; area: number } => {
 const newSquare = { color: 'white', area: 100 };
 if (config.color) {
 newSquare.color = config.color;
}
 if (config.width) {
 newSquare.area = config.width * config.width;
}
 return newSquare;
};

const mySquare = createSquare({ color: 'black' });
console.log(mySquare); // Output: { color: 'black', area: 100 }
```

In this example, both color and width are optional properties in the SquareConfig interface. Objects of this type may include either, both, or neither of these properties, providing flexibility in how the interface is used.

Q30: How does Type Inference work in TypeScript?

Type Inference in TypeScript is a feature where the compiler automatically determines the data type of a variable if it's not explicitly declared. This occurs in variable initialization and function return types.

```
const x = 3; // TypeScript infers x is a number
const y = [0, 1, null]; // TypeScript infers y is an array of numbers or null

const getLength = (obj: string | string[]) => {
 return obj.length; // TypeScript infers the return type is number
};
```

Type inference helps in reducing the amount of explicit type annotations needed while still maintaining type safety. It makes the code cleaner and more readable, while still providing the benefits of static typing.

Q31: Demonstrate the use of getters and setters in TypeScript with a functional component.

In TypeScript, we can use getters and setters to control access to an object's properties. In a functional React component, we can achieve similar functionality using state and custom hooks.

```
import React, { useState } from 'react';

interface EmployeeProps {
 initialName: string;
}

const useEmployee = (initialName: string) => {
 const [name, setName] = useState(initialName);

 return {
 get fullName(): string {
 return name;
},
 set fullName(newName: string) {
 if (newName && newName.length > 0) {
 setName(newName);
} else {
 throw new Error('Invalid name');
}
}
};
};

const Employee = ({ initialName }: EmployeeProps) => {
 const employee = useEmployee(initialName);

 const handleNameChange = (event: React.ChangeEvent<HTMLInputElement>) => {
 try {
 employee.fullName = event.target.value;
} catch (error) {
 console.error(error);
}
};

 return (
<div>
<input value={employee.fullName} onChange={handleNameChange} />
<p>Current name: {employee.fullName}</p>
</div>
);
};

// Usage
const App = () => <Employee initialName="John Doe" />;
```

This example demonstrates how to implement getter and setter-like behavior in a functional component using a custom hook. The useEmployee hook encapsulates the state and provides getter and setter methods, while the Employee component uses these to manage the fullName property.

Q32: Explain the concept of mixins in TypeScript with an example.

Mixins in TypeScript are a way to create classes that combine behavior from multiple sources. This approach allows for the composition of functionality from different classes into a single class.

```
// Mixin constructor type
type Constructor<T = {}> = new (...args: any[]) => T;

// Mixin function
function Timestamped<TBase extends Constructor>(Base: TBase) {
 return class extends Base {
 timestamp = Date.now();
};
}

// Another mixin function
function Activatable<TBase extends Constructor>(Base: TBase) {
 return class extends Base {
 isActive = false;
 activate() {
 this.isActive = true;
}
 deactivate() {
 this.isActive = false;
}
};
}

// Base class
class User {
 name: string;
 constructor(name: string) {
 this.name = name;
}
}

// Create a new class using the mixins
const TimestampedActivatableUser = Timestamped(Activatable(User));

// Use the new class
const user = new TimestampedActivatableUser("John");
console.log(user.timestamp);
user.activate();
console.log(user.isActive);
```

This example demonstrates how to use mixins to combine the functionalities of Timestamped and Activatable into the User class. Mixins provide a flexible way to add functionality to classes without deep inheritance hierarchies.

Q33: Describe namespaces in TypeScript.

Namespaces in TypeScript are a way to logically group related code. They help in organizing code and preventing naming collisions in large applications.

```
namespace Geometry {
 export interface Point {
 x: number;
 y: number;
}

 export class Circle {
 constructor(public center: Point, public radius: number) {}

 area(): number {
 return Math.PI * this.radius ** 2;
}
}
}

let circle = new Geometry.Circle({x: 0, y: 0}, 5);
console.log(circle.area()); // Output: 78.53981633974483
```

In this example, the Geometry namespace encapsulates related types and classes. The 'export' keyword makes these accessible outside the namespace.

Q34: How does inheritance work in TypeScript?

Inheritance in TypeScript allows a class to extend a base class, inheriting its properties and methods. This promotes code reuse and the creation of class hierarchies.

```
class Animal {
 move(distanceInMeters: number = 0) {
 console.log('Animal moved ' + distanceInMeters + 'm.');
}
}
class Dog extends Animal {
 bark() {
 console.log('Woof! Woof!');
}
}
const dog = new Dog();
dog.bark();  // Output: Woof! Woof!
dog.move(10);  // Output: Animal moved 10m.
```

In this example, Dog inherits the move method from Animal and adds its own bark method. The extends keyword is used to create a class inheritance.

Q35: Explain abstract classes in TypeScript.

Abstract classes in TypeScript are base classes from which other classes may be derived, but cannot be instantiated directly. They may contain abstract methods, which must be implemented by derived classes.

```
abstract class Animal {
 abstract makeSound(): void;

 move(): void {
 console.log('moving...');
}
}

class Dog extends Animal {
 makeSound(): void {
 console.log('Woof! Woof!');
}
}

// let animal = new Animal(); // Error: Cannot create an instance of an abstract class
let dog = new Dog();
dog.makeSound(); // Output: Woof! Woof!
dog.move(); // Output: moving...
```

Abstract classes are useful for defining a common interface for a set of subclasses, ensuring that certain methods are implemented while providing shared functionality.

Q36: How do static properties and methods work in TypeScript, and how can we achieve similar functionality in functional components?

Static properties and methods in TypeScript belong to the class itself rather than to instances of the class. They can be accessed without creating an instance of the class. In functional components, we can achieve similar functionality using object literals or modules.

```
// Class-based approach
class MathUtils {
 static PI: number = 3.14159;

 static square(x: number): number {
 return x * x;
}
}

console.log(MathUtils.PI);  // Output: 3.14159
console.log(MathUtils.square(4));  // Output: 16

// Functional approach
const FunctionalMathUtils = {
 PI: 3.14159,

 square: (x: number): number => x * x
};
```

```
console.log(FunctionalMathUtils.PI);  // Output: 3.14159
console.log(FunctionalMathUtils.square(4));  // Output: 16

// Usage in React components
type MathProps = { value: number };

// Class component using static method
class ClassMathComponent extends React.Component<MathProps> {
 render() {
 return <div>{MathUtils.square(this.props.value)}</div>;
}
}

// Functional component using FunctionalMathUtils
const FunctionalMathComponent = ({ value }: MathProps) => (
<div>{FunctionalMathUtils.square(value)}</div>
);
```

Both approaches allow you to access properties and methods without instantiation. The class-based approach uses the 'static' keyword, while the functional approach uses an object literal. In React components, you can use either method to access these shared utilities.

Q37: Explain conditional types in TypeScript and their use cases.

Conditional types in TypeScript allow for the creation of types that depend on other types. They use a syntax similar to conditional expressions. Conditional types are powerful for creating flexible, reusable type definitions and for type inference in complex scenarios.

```
type CheckNumber<T> = T extends number ? "Is a number" : "Not a number";

type Result1 = CheckNumber<5>; // "Is a number"
type Result2 = CheckNumber<"hello">; // "Not a number"

type ArrayOrSingle<T> = T extends any[] ? T[number] : T;

type Item = ArrayOrSingle<string[]>; // string
type SingleItem = ArrayOrSingle<number>; // number
```

Q38: Describe the concept of mapped types in TypeScript with an example.

Mapped types in TypeScript allow the creation of new types based on existing ones by transforming properties. They're particularly useful for creating variations of existing types. Mapped types provide a powerful way to transform types systematically, reducing code duplication and enhancing type safety.

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

interface Mutable {
 a: number;
 b: string;
}

type ReadonlyMutable = Readonly<Mutable>;

// ReadonlyMutable is equivalent to:
// {
//   readonly a: number;
//   readonly b: string;
// }
```

Q39: How does the 'keyof' operator function in TypeScript?

The 'keyof' operator in TypeScript is used to get the union of keys from an object type. It's particularly useful when creating types based on the properties of other types. The 'keyof' operator is often used in generic types and functions to create flexible, type-safe operations on objects.

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

type PersonKeys = keyof Person; // "name" | "age" | "location"

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

const person: Person = {
 name: "Alice",
 age: 30,
 location: "New York"
};

const name = getProperty(person, "name"); // type is string
const age = getProperty(person, "age"); // type is number
```

Q40: Describe TypeScript's handling of null and undefined with strict null checks.

TypeScript provides special handling for null and undefined, particularly with the strict null checks option enabled. This feature helps catch errors related to these values by enforcing more rigorous type checking. Strict null checks force developers to handle null and undefined explicitly, reducing runtime errors and improving overall code quality.

```
// With strict null checks (tsconfig.json: "strictNullChecks": true)
let age: number;
```

```
// console.log(age); // Error: Variable 'age' is used before being assigned

let height: number | null = null; // Union type
height = 175; // OK
// height = undefined; // Error

const greet = (name: string | null) => {
 if (name === null) {
 console.log("Hello, guest!");
} else {
 console.log("Hello, " + name.toUpperCase() + "!");
}
};

greet("Alice"); // OK
greet(null); // OK
// greet(undefined); // Error
```

Q41: Describe discriminated unions in TypeScript and their use in modeling state.

Discriminated unions in TypeScript are a pattern where a common property (the discriminant) is used to differentiate between different types in a union. They're particularly useful for modeling different states of a system, such as loading, success, or error states in asynchronous operations. This pattern provides type safety when dealing with different variants of a type, making it especially valuable in state management scenarios.

```
type LoadingState = { state: "loading" };
type SuccessState = { state: "success"; data: string[] };
type ErrorState = { state: "error"; error: string };

type FetchState = LoadingState | SuccessState | ErrorState;

const handleState = (state: FetchState) => {
 switch (state.state) {
 case "loading":
 console.log("Loading...");
 break;
 case "success":
 console.log("Data:", state.data);
 break;
 case "error":
 console.log("Error:", state.error);
 break;
}
};

const loadingState: FetchState = { state: "loading" };
const successState: FetchState = { state: "success", data: ["item1", "item2"] };
const errorState: FetchState = { state: "error", error: "Failed to fetch" };

handleState(loadingState);
handleState(successState);
handleState(errorState);
```