INTERVIEW QUESTIONS ON TYPE SCRIPT AND JAVA SCRIPT

Q1: What are the data types in JavaScript?

- **Primitive types:**
  1. **String**: Represents a sequence of characters, e.g., `'hello'`.
  2. **Number**: Represents both integer and floating-point numbers, e.g., `42`, `3.14`.
  3. **BigInt**: Represents large integers, e.g., `1234567890123456789012345678901234567890n`.
  4. **Boolean**: Represents `true` or `false`.
  5. **Undefined**: Represents a variable that has been declared but not assigned a value.
  6. **Symbol**: Represents a unique identifier, often used for object property keys.
  7. **Null**: Represents the intentional absence of any value or object.
- **Non-primitive type:**
  - **Object**: A collection of key-value pairs. Arrays, functions, and other objects are types of objects in JavaScript.
    - **Array**: An ordered list of values, e.g., `[1, 2, 3]`.
    - **Function**: A callable block of code, e.g., `function foo() { return 42; }`.

Q2: What is the difference between var, let, and const?

- **var**:
  - Function-scoped, meaning it is accessible throughout the entire function where it is declared, including inside blocks.
  - Can be redeclared within the same scope.
  - Hoisted, meaning it is moved to the top of the scope with an initial value of `undefined`.
- **let**:
  - Block-scoped, meaning it is only accessible within the block where it is declared (like loops or conditionals).
  - Cannot be redeclared in the same scope.
  - Hoisted but not initialized, leading to a **temporal dead zone** (TDZ) where accessing it before the declaration results in an error.
- **const**:
  - Block-scoped, similar to `let`.
  - Must be initialized at the time of declaration.
  - Cannot be reassigned after initialization. However, if it's an object or array, the contents (properties/elements) can still be modified.

Q3: What are the different types of functions in JavaScript?

1. **Named functions**: A traditional function declaration with a name.

```javascript
function add(a, b) {
    return a + b;
}
```

   - **Characteristics**:
     - Can be called before its declaration due to hoisting.
     - Has a name that can be used to refer to the function.

2. **Anonymous functions**: A function that does not have a name, often assigned to a variable or passed as an argument.

```javascript
const add = function(a, b) {
    return a + b;
};
```

   - **Characteristics**:
     - Cannot be called before its declaration (because of hoisting rules).
     - Typically used for callbacks or passed directly into other functions.

3. **Arrow functions**: A shorter syntax for writing functions, introduced in ES6.

```javascript
const add = (a, b) => a + b;
```

   - **Characteristics**:
     - Always anonymous.
     - Inherits the `this` value from the surrounding context (lexical scoping).
     - Cannot be used as constructors (i.e., they cannot be called with `new`).
     - Do not have the `arguments` object, which is available in regular functions.

4. **Immediately Invoked Function Expressions (IIFE)**: A function that is defined and immediately executed.

```javascript
(function() {
    console.log("IIFE");
})();
```

   - **Characteristics**:
     - Useful for creating a new scope to avoid polluting the global scope.
     - Often used to create isolated modules or for running code that doesn't need to be reused.

Q4: Explain closures in JavaScript.

A **closure** is a function that has access to its own scope, the scope in which it was created, and the global scope. It allows a function to "remember" and access variables from its lexical scope even after that scope has finished execution.

Example:

javascript Copy code

```javascript
function outerFunction(outerVariable) {
    return function innerFunction(innerVariable) {
        console.log(outerVariable); // Accesses outer function's variable
        console.log(innerVariable); // Accesses inner function's variable
    };
}

const closureExample = outerFunction('outer'); // outerFunction is executed
closureExample('inner'); // innerFunction is executed
```

- **Explanation:**
  - `outerFunction` creates a local variable `outerVariable` and returns `innerFunction`.
  - `innerFunction` has access to both its own scope (where `innerVariable` is defined) and the outer function's scope (where `outerVariable` is defined).
  - When `closureExample` is called, it retains access to the `outerVariable`, even though the outer function has already finished execution.

Closures are often used for:

- **Data encapsulation**: Keeping certain variables private to a function while exposing a public interface.
- **Function factories**: Creating functions that can remember and work with data over time.

Q5: What are promises, and how do they work?

A **Promise** is an object in JavaScript that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to handle asynchronous code more effectively by providing methods to handle success (`resolve`) or failure (`reject`) once the operation completes.

States of a Promise:

1. **Pending**: The promise is still in progress (the asynchronous operation is still running).
2. **Fulfilled**: The asynchronous operation completed successfully (the promise was resolved).
3. **Rejected**: The asynchronous operation failed (the promise was rejected).

Promise Syntax:

javascript Copy code

```javascript
const promise = new Promise((resolve, reject) => {
  if (condition) resolve("Success");
  else reject("Error");
});
```

- **resolve()**: This method is called when the asynchronous operation completes successfully.
- **reject()**: This method is called when the asynchronous operation fails.

You can handle the promise's resolution or rejection using `.then()` and `.catch()`:

javascript

```javascript
promise
  .then((result) => {
    console.log(result); // Success: "Success"
  })
  .catch((error) => {
    console.log(error); // Failure: "Error"
  });
```

Q6: What is the difference between async/await and .then()?

Both `async/await` and `.then()` are used to handle asynchronous operations, but they differ in syntax, readability, and how they are used.

1. **Using `.then()`:**

`.then()` is a method that attaches callbacks to a promise to handle the result or error. It is the traditional way to handle asynchronous operations in JavaScript.

- **Syntax:**

```javascript
asyncFunction()
  .then((result) => {
    console.log(result); // Success case
  })
  .catch((error) => {
    console.log(error); // Error case
  });
```

- **Characteristics:**
  - You chain `.then()` for each subsequent operation.
  - Can lead to callback "pyramid" or "callback hell" in complex cases.
  - Works well for simple cases, but becomes harder to read when dealing with multiple asynchronous operations.

2. **Using `async/await`:**

`async/await` is a more modern and concise way to handle asynchronous code, introduced in ES2017. It is built on top of Promises but provides a cleaner syntax that makes asynchronous code look synchronous.

- **Syntax:**

```javascript
async function example() {
  try {
    const result = await asyncFunction();
    console.log(result); // Success case
  } catch (error) {
    console.log(error); // Error case
  }
}
```

- **Characteristics:**
  - `async`: A function declared with `async` always returns a promise.
  - `await`: Pauses the execution of the function until the Promise is resolved or rejected.
  - It makes asynchronous code easier to read and maintain, especially when handling multiple asynchronous operations.
  - It eliminates chaining and makes error handling more straightforward with `try/catch`.

Comparison:

- **Readability:** `async/await` is often considered more readable and maintains a more synchronous flow of code. This reduces the need for chaining multiple `.then()` calls, making the code easier to follow.
- **Error handling:** With `.then()`, you handle errors with `.catch()`, but with `async/await`, you use `try/catch` blocks, which is familiar to handling synchronous code errors.
- **Control flow:** `async/await` provides a more linear and structured way to work with asynchronous code, especially when handling multiple dependent asynchronous operations.

In summary:

- `.then()` is more traditional and works well for simple cases.
- `async/await` makes your code cleaner, more readable, and easier to handle with complex asynchronous flows.

Q7: How do you select elements in the DOM?

There are several ways to select elements in the DOM (Document Object Model) in JavaScript, depending on the requirement and the specificity of the selector:

1. `getElementById()`:
   - Selects an element by its unique `id` attribute.
   - **Example:**

```javascript
const element = document.getElementById('myId');
```

   - This method returns a single element, or `null` if no element with the specified ID is found.
2. `getElementsByClassName()`:
   - Selects all elements with a specified class name.
   - **Example:**

```javascript
const elements = document.getElementsByClassName('myClass');
```

   - This method returns a live HTMLCollection, which automatically updates when the DOM changes.
3. `querySelector()`:
   - Selects the first element that matches a specified CSS selector.
   - **Example:**

```javascript
const element = document.querySelector('.myClass');
```

   - This method returns a single element, or `null` if no element matches the selector.
4. `querySelectorAll()`:
   - Selects all elements that match a specified CSS selector.
   - **Example:**

```javascript
const elements = document.querySelectorAll('div');
```

   - This method returns a static NodeList, meaning it does not automatically update when the DOM changes.

Each of these methods has its strengths and use cases, depending on whether you need to select a single element or multiple elements, and whether you want to use CSS selectors for greater flexibility.

Q8: Explain event delegation in JavaScript.

Event delegation is a technique in JavaScript where a single event listener is attached to a parent element instead of individual child elements. This is particularly useful when you have many child elements or dynamically added elements. The idea is that the event listener will catch events that "bubble up" from the child elements to the parent, allowing you to handle events for child elements without adding an event listener to each one.

How event delegation works:

1. **Event Bubbling**: When an event occurs on an element, it bubbles up through the DOM tree, triggering events on its ancestors.
2. **Event Target**: The `event.target` property refers to the element that triggered the event. You can check `event.target` to determine which child element was interacted with.

Example:

javascript Copy code

```javascript
const parentElement = document.getElementById('parent');

parentElement.addEventListener('click', (event) => {
  if (event.target && event.target.matches('button.className')) {
    console.log('Button clicked');
  }
});
```

- In this example, we attach a single `click` event listener to the `parentElement`.
- The `event.target` inside the listener checks if the clicked element is a `button` with a specific class (`'className'`).
- If it is, the corresponding action is performed.
- This works even if the button elements are dynamically added after the page is loaded, because the event listener is on the parent and will handle clicks on all child buttons.

Advantages of event delegation:

- **Performance**: Instead of attaching multiple event listeners to each child element, only one listener is needed on the parent.
- **Dynamic Content**: Works even if elements are added or removed dynamically after the initial page load.
- **Less memory consumption**: You don't have to create a separate listener for each child element, which is especially beneficial for large numbers of elements.

Q9: What is the difference between `try...catch` and `throw`?

- `try...catch`:
  - **Purpose**: Used to handle exceptions (errors) that occur in a block of code.
  - **Usage**: Wraps code that might throw an error, and provides a way to gracefully handle those errors instead of letting them break the program.
  - **Syntax**:

```javascript
try {
  // Code that might throw an error
  let result = riskyFunction();
} catch (error) {
  // Handle the error
  console.error(error);
}
```

- `throw`:
  - **Purpose**: Used to explicitly raise an error or exception, either built-in or custom.
  - **Usage**: You can throw errors to indicate that something has gone wrong in the program, and it can be caught by a `try...catch` block.
  - **Syntax**:

```javascript
throw new Error('Something went wrong');
```

  - **Note**: `throw` is used to create exceptions, while `try...catch` is used to catch and handle them.

6. **Testing Frameworks for JavaScript**

Q10: What are some common testing frameworks in JavaScript?

- **Answer**: Popular testing frameworks include:
  - **Unit Testing**: Jest, Mocha, Jasmine.
  - **End-to-End Testing**: Cypress, Puppeteer, Playwright.

Q11: What is TypeScript? How is it different from JavaScript?

- **TypeScript** is a **superset** of JavaScript that adds **static typing** to the language. This means you can define the types of variables, function parameters, and return values at compile time, which provides better tooling support and helps catch errors early in development. TypeScript compiles to regular JavaScript, so it can run anywhere JavaScript can run (browsers, Node.js, etc.).
- **Key Differences from JavaScript**:
  1. **Static Typing**: TypeScript allows you to specify types for variables, function parameters, and return values, which helps catch type-related errors during development. JavaScript is dynamically typed, meaning types are determined at runtime.
     - **Example**:

```typescript
let age: number = 30; // Type is statically defined as number
```

  2. **Type Inference**: TypeScript can automatically infer types based on the assigned value, reducing the need to explicitly declare types.
     - **Example**:

```typescript
let age = 30; // TypeScript infers the type as number
```

  3. **Interfaces and Types**: TypeScript introduces **interfaces** and **type aliases** to define object shapes and other complex data types.
     - **Example**:

```typescript
interface Person {
  name: string;
  age: number;
}
const john: Person = { name: "John", age: 25 };
```

  4. **Classes and Modules**: TypeScript has enhanced support for **classes** and **modules**, with stronger type safety and features like access modifiers (`public`, `private`, `protected`).
  5. **Compilation**: TypeScript code must be compiled (transpiled) into JavaScript using the TypeScript compiler (`tsc`), while JavaScript can be executed directly by browsers or Node.js.

In summary, TypeScript enhances JavaScript by adding type checking, object-oriented features, and better development tooling, but still allows you to run the code as JavaScript after compilation.

Q12: What are TypeScript's basic types?

TypeScript offers a variety of **primitive types** and **complex types** that extend JavaScript's capabilities with static typing.

1. `string`: Represents a sequence of characters.
   - **Example**: `let name: string = 'Alice';`
2. `number`: Represents both integer and floating-point numbers.
   - **Example**: `let age: number = 30;`
3. `boolean`: Represents a value of `true` or `false`.
   - **Example**: `let isActive: boolean = true;`
4. `array`: A collection of elements of a specific type.
   - **Example**: `let numbers: number[] = [1, 2, 3];`
   - Alternatively, using a generic type: `let numbers: Array<number> = [1, 2, 3];`
5. `tuple`: Represents an ordered collection of elements with different types.
   - **Example**: `let person: [string, number] = ['Alice', 30];`
6. `enum`: A way to define a set of named constants. Enums can be numeric or string-based.
   - **Example**:

```typescript
enum Direction {
  Up = 1,
  Down,
  Left,
  Right
}
let direction: Direction = Direction.Up;
```

7. `any`: Represents any type. It is used when you don't want to enforce type checking. It can hold any value and bypasses TypeScript's strict type system.
   - **Example**: `let value: any = 'Hello';`
8. `unknown`: Represents any value, but unlike `any`, you must perform a type check before performing any operations on it.
   - **Example**:

```typescript
let value: unknown = 5;
if (typeof value === 'number') {
  console.log(value + 10);
}
```

9. `void`: Represents the absence of any value. It is typically used for functions that do not return a value.
   - **Example**: `function logMessage(message: string): void { console.log(message); }`
10. `never`: Represents a value that never occurs (e.g., a function that always throws an error or an infinite loop).
    - **Example**:

```typescript
function throwError(message: string): never {
  throw new Error(message);
}
```

11. `null`: Represents the absence of any object or value.
    - **Example**: `let x: null = null;`
12. `undefined`: Represents an uninitialized variable or a variable that hasn't been assigned a value.
    - **Example**: `let x: undefined;`

Q13: What is the difference between `interface` and `type` in TypeScript?

Both `interface` and `type` are used to define types in TypeScript, but they have some key differences and specific use cases.

1. `interface`:
   - **Purpose**: Primarily used to define the **structure of objects** (the properties and methods they contain).
   - **Extending**: Interfaces can be extended using the `extends` keyword. This allows you to create a new interface that inherits properties from one or more existing interfaces.
   - **Declaration Merging**: Interfaces support **declaration merging**, meaning you can define an interface with the same name multiple times, and TypeScript will automatically merge their properties.
   - **Example**:

typescript Copy code

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

// Extending an interface
interface Employee extends Person {
  salary: number;
}
```

2. `type`:
   - **Purpose**: More **flexible** than `interface`. A `type` can represent a wide variety of constructs, such as primitive types, unions, intersections, tuples, etc.
   - **Union Types**: Type aliases allow the definition of **union types** (e.g., a variable can be one type or another).
   - **Intersection Types**: Type aliases also support **intersection types**, which combine multiple types into one.
   - **Cannot be merged**: Unlike interfaces, type aliases cannot be declared multiple times to merge their properties.
   - **Example**:

typescript Copy code

```typescript
type Person = {
  name: string;
  age: number;
};

// Union type
type ID = string | number;

// Intersection type
type Employee = Person & { salary: number };
```

Key Differences:

- **Flexibility**: `type` is more flexible and can represent a broader range of types, including primitive types, unions, and intersections. `interface` is more rigid and specifically meant for object structures.
- **Extensibility**: Interfaces can be extended and merged, while `type` aliases cannot.
- **Use Case**: Use `interface` when defining the shape of an object and when you want to take advantage of declaration merging or inhe ↓ ice. Use `type` when you need more complex type combinations (like unions or intersections) or need to represent primitive types.

Q14: What is a union and an intersection type in TypeScript?

In TypeScript, **union types** and **intersection types** are used to combine multiple types, but in different ways.

1. **Union Type (|):**
   - A **union type** allows a variable to be one of several types. It combines multiple types, meaning a variable can hold a value of any one of the types in the union.
   - **Syntax:** `type = type1 | type2 | ...`
   - **Example:**

```typescript
type A = string | number; // A can be either a string or a number
let value: A;

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

2. **Intersection Type (&):**
   - An **intersection type** combines multiple types into one. A variable of an intersection type must satisfy all the types it intersects. In other words, the variable must have all the properties and methods of the combined types.
   - **Syntax:** `type = type1 & type2 & ...`
   - **Example:**

```typescript
type A = { name: string };
type B = { age: number };

type C = A & B;  // C must have both 'name' and 'age'

const person: C = { name: 'Alice', age: 30 }; // valid
const incompletePerson: C = { name: 'Alice' }; // Error: Property 'age' is missing
```

Summary:

- **Union types (|)** represent a value that can be one of several types.
- **Intersection types (&)** represent a value that must be all the types combined.

Use cases:

- **Union types** are useful when you want a variable to accept different types of values (e.g., string or number).
- **Intersection types** are useful when you need a variable to satisfy multiple type constraints simultaneously (e.g., an object that must have the properties of two or more types).

Q15: What are generics in TypeScript? Provide an example.

Generics in TypeScript allow you to create **reusable components** that can work with **multiple types** without losing the benefits of type safety. With generics, you can create functions, classes, or interfaces that can operate on different types while still preserving their type relationships.

- **Generics provide flexibility** by allowing you to specify the type that will be used at runtime, while keeping the implementation generic.
- **Syntax**: The type parameter is enclosed in angle brackets (`<T>`) and is used as a placeholder for a specific type that will be defined when the function or class is used.

Example:

typescript Copy code

```typescript
// A simple generic function that returns the argument passed to it
function identity<T>(arg: T): T {
  return arg;
}

let output1 = identity<string>("Hello, TypeScript!");  // 'output1' is of type 'string'
let output2 = identity<number>(42);  // 'output2' is of type 'number'
```

- **Explanation**:
  - The function `identity` uses a type parameter `<T>` to represent the type of the argument `arg`. When you call the function, you specify the type (`string` or `number` in this case), and TypeScript ensures that the argument passed matches the specified type.
  - This allows you to write code that is flexible and reusable without losing type safety.

Why Use Generics?

- **Type Safety**: They help ensure that the types are correct, even for reusable components.
- **Reusability**: You can create functions and classes that work with any type.
- **No Redundant Type Definitions**: Generics allow you to use a type once and reuse it throughout the function or class.

Q16: What are decorators in TypeScript?

Decorators in TypeScript are a powerful feature that allows you to **modify the behavior** of classes and their members (properties, methods, etc.). They are a form of **metadata** attached to classes or class members and can be used for various purposes such as logging, validation, or adding additional functionality.

- **Decorators are experimental** and require the `experimentalDecorators` compiler option to be enabled in `tsconfig.json`.
- Decorators can be applied to:
  - **Classes**: To modify class behavior or add metadata.
  - **Methods**: To modify method behavior.
  - **Accessors**: To modify getter/setter behavior.
  - **Properties**: To modify property behavior.
  - **Parameters**: To modify parameters.

Why Use Decorators?

- **Separation of Concerns**: Decorators allow you to separate concerns like logging, validation, and transformation from the core logic of a class or method.
- **Reusability**: You can apply the same decorator across multiple classes or methods without repeating code.
- **Metadata**: Decorators provide a way to add metadata to classes or methods, which can be useful in frameworks like Angular for dependency injection, routing, etc.

Summary:

- **Generics** allow creating flexible, reusable components that can work with any type while maintaining type safety.
- **Decorators** allow modifying or enhancing classes, methods, properties, or parameters in a declarative way, adding extra functionality like logging, validation, and more.

Both features are highly useful in TypeScript, making your code more maintainable and powerful while ensuring type safety.

- **Answer**: Decorators are special annotations used for classes and members.

typescript Copy code

```typescript
function Log(target: any, propertyName: string) {
  console.log(`${propertyName} was accessed`);
}
class User {
  @Log
  name: string;
}
```

Q17: What is the `tsconfig.json` file?

The `tsconfig.json` file is a configuration file used to specify options for the TypeScript compiler. It tells the compiler how to compile your TypeScript code into JavaScript and allows you to set various options related to the compilation process.

Key Sections in `tsconfig.json`:

1. **compilerOptions**: This section specifies various compiler settings.
   - `target`: Determines the version of JavaScript to output (e.g., `ES5`, `ES6`, `ES2017`).
   - `module`: Defines the module system to use (e.g., `CommonJS`, `ES6`, `AMD`).
   - `strict`: Enables a wide range of type-checking options that make TypeScript more rigorous (e.g., `strictNullChecks`, `noImplicitAny`, etc.).
   - `outDir`: Specifies the output directory for compiled JavaScript files.
   - `sourceMap`: If set to `true`, generates source maps, which help with debugging by mapping compiled JavaScript code back to the original TypeScript code.
2. **include/exclude**: Specifies the files or directories to include or exclude from compilation.
   - `include`: Defines the file patterns to include in the compilation (e.g., `["src/**/*"]`).
   - `exclude`: Defines the files or directories to exclude from the compilation (e.g., `["node_modules"]`).
3. **files**: This section allows you to explicitly specify files to be included in the compilation.

- **Example**:

```json
{
  "compilerOptions": {
    "target": "ES6",
    "module": "CommonJS",
    "strict": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

Purpose of `tsconfig.json`:

- **Customizable Compilation**: It lets you control various compilation settings, ensuring that TypeScript compiles your code the way you want.
- **Consistent Setup**: The `tsconfig.json` file ensures that all developers working on the project have the same TypeScript settings.
- **Project-wide Settings**: The file can be placed at the root of a project, and its settings will apply to all TypeScript files in the project unless overridden by other `tsconfig.json` files in subdirectories.

Q18: How can TypeScript enhance testing?

TypeScript enhances testing by improving the quality of the code and catching errors early during development. Here's how:

1. **Better Type Safety**:
   - TypeScript ensures that the types of variables, parameters, and return values are consistent and valid. This means that common runtime errors, such as calling methods on `undefined` or passing incorrect types to functions, can be caught during compilation rather than at runtime.
   - **Example**: In a test case, TypeScript would catch errors like passing a string where a number is expected.

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

// TypeScript will catch this error at compile time
add('5', 10); // Error: Argument of type 'string' is not assignable to parameter o
```

2. **Improved Code Readability**:
   - TypeScript's explicit types make it easier to understand the code and the expected input/output types for functions and methods. This can help developers write more reliable tests, as they know exactly what types they are dealing with.
   - **Example**: Instead of guessing the types, you can explicitly define the types in test cases:

```typescript
const testPerson: Person = { name: 'Alice', age: 30 }; // Clear type definition fo
```

3. **Early Error Detection**:
   - TypeScript catches errors at compile time, reducing the likelihood of bugs making it into the production code. This is especially useful in testing, where you want to catch issues early before running the actual tests.
   - **Example**: If a test case uses incorrect or mismatched types or calls undefined properties, TypeScript will flag it before the tests are executed.
4. **Stronger IDE Support**:
   - The strong typing in TypeScript provides better tooling support, such as autocompletion, inline documentation, and error highlighting. This makes writing and maintaining tests much easier, as developers can see potential issues and suggestions while writing tests.
5. **Integration with Testing Frameworks**:
   - TypeScript integrates seamlessly with popular testing frameworks like **Jest**, **Mocha**, **Jasmine**, and others. It ensures that the tests themselves are type-safe, helping you write more accurate and reliable tests.
   - TypeScript also enables type-checking for the test suites and assertions, ensuring that you catch type mismatches or logic errors in your tests.
6. **Better Refactoring**:
   - TypeScript's type system makes refactoring easier and safer. When you modify the code, the TypeScript compiler ensures that any changes in types are reflected across all usages, including in your tests. This helps prevent errors during refactoring, ensuring that tests remain valid after changes.

Q19: Explain prototypes in JavaScript. How is it related to inheritance?

In JavaScript, **prototypes** form the basis of inheritance. Every object in JavaScript has an internal property called `[[Prototype]]` (often accessed via `__proto__`), which points to another object. This forms a chain of prototypes, known as the **prototype chain**.

Key Points about Prototypes:

- Every **function** in JavaScript has a **prototype** property that is used for inheritance. This property contains the methods and properties that are shared across all instances of the function.
- When you create an object from a function (constructor), the object's `[[Prototype]]` is set to the constructor's `prototype` property, allowing the object to inherit properties and methods from the constructor function.

Example:

javascript Copy code

```javascript
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name}`);
};

const john = new Person("John");
john.greet();  // Output: Hello, my name is John
```

- **Explanation**:
  - The `Person` function is a constructor. The `greet` method is added to the `Person.prototype` object.
  - When you create a new instance of `Person` (e.g., `john`), the instance inherits the `greet` method through the **prototype chain**.
  - This allows all instances of `Person` to share the same method, rather than creating a new `greet` method for each instance, saving memory and ensuring efficient method sharing.

Prototypes and Inheritance:

- **Prototype-based inheritance** means that objects inherit directly from other objects (their prototype), rather than from classes as in classical inheritance.
- When a property or method is accessed on an object, JavaScript first looks for it on the object itself. If it's not found, JavaScript checks the **prototype** (i.e., the object's `[[Prototype]]`). This continues up the prototype chain until the property is found or the chain ends (at `null`).

Q20: What is the difference between classical inheritance and prototypal inheritance?

Classical inheritance and **prototypal inheritance** represent two different approaches to inheritance in object-oriented programming.

1. **Classical Inheritance (e.g., Java, C++):**

- **Concept**: In classical inheritance, **classes** define the blueprint for objects. Objects are instances of classes, and inheritance is achieved by extending a base class.
- **Inheritance Mechanism**: A subclass inherits from a superclass, and new objects are created by instantiating classes.
- **Class-based**: Classes are the central concept, and inheritance is achieved through class hierarchies.
- **Example** (Java):

java Copy code

```java
class Animal {
  void speak() {
    System.out.println("Animal speaks");
  }
}

class Dog extends Animal {
  void speak() {
    System.out.println("Dog barks");
  }
}
```

- **Key Characteristics:**
  - A class defines the structure (properties and methods) for all instances.
  - **Instances** are created from **classes**.
  - **Methods** and **properties** are inherited from the base class to derived classes.

2. **Prototypal Inheritance (JavaScript):**

- **Concept**: In prototypal inheritance, objects directly inherit from other objects, not from classes. Objects can share properties and methods with other objects via their **prototype**.
- **Inheritance Mechanism**: An object can inherit from another object by setting its `[[Prototype]]` to point to the prototype of another object.
- **Object-based**: Inheritance is done through objects, rather than classes.
- **Example** (JavaScript):

javascript Copy code

```javascript
const animal = {
  speak: function() {
    console.log("Animal speaks");
  }
};

const dog = Object.create(animal);  // dog inherits from animal
dog.speak = function() {
  console.log("Dog barks");
};

dog.speak();  // Output: Dog barks
```

- **Key Characteristics:**
  - Objects inherit directly from other objects via the prototype chain.
  - Objects are created from other objects (using constructors or `Object.create`).
  - There is no **class** concept in prototype inheritance; inheritance is directly between objects.

Q21: Explain the JavaScript event loop. Why is it important?

The **JavaScript event loop** is the mechanism that allows JavaScript, which is single-threaded, to handle asynchronous operations like I/O tasks (e.g., reading from files, network requests) without blocking the execution of other code.

Event Loop Process:

1. **Call Stack**: The **call stack** is where JavaScript executes synchronous code. When a function is invoked, it's pushed to the stack. Once the function is completed, it's popped off the stack. If there is no synchronous code to execute, the event loop looks to the task queue.
2. **Web APIs**: Asynchronous tasks (like `setTimeout`, `fetch`, etc.) are managed by Web APIs. These APIs handle asynchronous code outside of the JavaScript runtime and then pass the callback function to the task queue once the operation completes.
3. **Task Queue**: The **task queue** holds the asynchronous callbacks (e.g., from `setTimeout`, event listeners, etc.) that are ready to be executed. Once the call stack is empty, the event loop will move the callback functions from the task queue to the call stack for execution.
4. **Event Loop**: The event loop continuously monitors the call stack and task queue. When the call stack is empty, it pushes the first item in the task queue to the stack, allowing asynchronous code to run.

Example:

javascript Copy code

```javascript
console.log("Start");
setTimeout(() => console.log("Async"), 1000);
console.log("End");
```

Execution order:

- "**Start**" is logged first because it's synchronous.
- `setTimeout` is an asynchronous operation, so it goes to the Web API, and its callback is queued in the task queue.
- "**End**" is logged because it's also synchronous.
- After 1 second, the "**Async**" message is logged because the callback from `setTimeout` is moved from the task queue to the call stack once the synchronous code is executed.

Why is it important?

- The **event loop** is crucial because it allows JavaScript to execute non-blocking code despite being single-threaded. This enables JavaScript to perform multiple asynchronous operations efficiently without freezing the main thread, such as handling user input, network requests, or animations.

Q22: What are some important ES6+ features?

ES6 (ECMAScript 2015) introduced several new features to JavaScript that greatly improved its capabilities and developer experience. Subsequent versions of ECMAScript have continued to introduce new features. Here are some of the most important ES6+ features:

1. Let and Const

- `let`: A block-scoped variable declaration, unlike `var` which is function-scoped.
- `const`: Declares a constant variable that cannot be reassigned. The value of objects or arrays declared with `const` can still be modified, but the reference to the variable itself cannot change.

javascript Copy code

```javascript
let a = 10;
a = 20; // valid

const b = 30;
b = 40; // TypeError: Assignment to constant variable.
```

2. Arrow Functions

Arrow functions provide a shorter syntax for writing functions and lexically bind `this` (the value of `this` is inherited from the surrounding context).

javascript Copy code

```javascript
const add = (a, b) => a + b;
const greet = () => console.log("Hello!");
```

3. ES6+ Features

Q22: What are some important ES6+ features?

- **Answer:**
  - **Arrow Functions:** `const add = (a, b) => a + b;`
  - **Template Literals:** `console.log(` Hello ${name} `);`
  - **Destructuring:** `const { name, age } = person;`
  - **Modules:** `export` and `import`.
  - **Spread/Rest Operators:**

javascript Copy code

```javascript
const arr = [1, 2, 3];
const copy = [...arr]; // Spread
const [first, ...rest] = arr; // Rest
```

Q23: How do you validate form inputs in JavaScript?

Form validation in JavaScript is a critical part of ensuring that users provide valid data. You can validate inputs using **event listeners** to capture user input and apply validation rules like **regular expressions (regex)** or **conditional checks**.

Example Code for Email Validation:

```javascript
document.querySelector("#form").addEventListener("submit", (e) => {
  const email = document.querySelector("#email").value;

  // Regular expression for basic email validation
  if (!/\S+@\S+\.\S+/.test(email)) {
    e.preventDefault(); // Prevent form submission
    alert("Invalid email"); // Alert the user
  }
});
```

Steps in this Example:

1. **Event Listener:** We attach a `submit` event listener to the form.
2. **Extracting Value**: The email input value is captured using `document.querySelector("#email").value`.
3. **Validation:** A **regular expression** (`/\S+@\S+\.\S+/`) is used to check if the email follows a basic valid pattern (contains `@` and a domain).
4. **Conditional Check:** If the email doesn't match the pattern, the form submission is prevented (`e.preventDefault()`) and an alert is shown.

This approach can be applied to other types of inputs, such as text, numbers, dates, etc., by using appropriate validation rules (regex, built-in functions, or custom checks).

Q24: How would you test asynchronous functions in JavaScript?

Testing asynchronous functions is essential in modern JavaScript applications, especially with operations like **network requests**, **timeouts**, or **database queries**. Frameworks like Jest make it easy to handle async testing, utilizing features such as **async/await**, `.resolves`, or `.rejects` for promises.

Using async/await with Jest:

```javascript
test('fetches data from API', async () => {
  const data = await fetchData(); // async function
  expect(data).toEqual({ name: 'John' }); // expect correct data
});
```

Using .resolves and .rejects:

- `.resolves` is used for asserting that a promise resolves correctly.
- `.rejects` is used for asserting that a promise rejects with an error.

```javascript
test('fetches user data successfully', () => {
  return expect(fetchUserData()).resolves.toEqual({ name: 'Alice' });
});

test('fetch fails with error', () => {
  return expect(fetchUserData()).rejects.toThrow('Error fetching user');
});
```

Key Concepts:

1. **Mocking:** You may want to mock functions that perform asynchronous operations, like API requests, to simulate different conditions (e.g., success, failure).
   - Jest provides tools like `jest.fn()` or `jest.mock()` to mock asynchronous functions.
2. **Handling Timeouts:** When testing functions involving timeouts, you can use Jest's `done` callback or `async/await` to manage asynchronous code.

Example with Timeouts:

```javascript
test('waits 2 seconds before resolving', async () => {
  const result = await delay(2000); // Function that resolves after 2 seconds
  expect(result).toBe('Done');
});
```

By using async/await in Jest tests, you can make your asynchronous code easier to handle, making the test more readable and reliable.

Q25: What is the difference between modules and namespaces in TypeScript?

In TypeScript, both **modules** and **namespaces** are used for organizing code, but they serve different purposes and work in different ways.

Modules:

- **Definition:** Modules are used to organize code across **multiple files**. They allow you to split your code into smaller, manageable files and use `import` and `export` statements to share functionality between them.
- **Usage:** Modules are external by default in TypeScript, meaning they rely on file-based separation and can be used in both **browser-based** and **Node.js** environments.
- **Example:**
  - **Exporting from a module:**

```typescript
// math.ts
export function add(a: number, b: number): number {
  return a + b;
}
```

  - **Importing from a module:**

```typescript
// app.ts
import { add } from './math';
console.log(add(2, 3)); // Output: 5
```

- **Key Points:**
  - **File-based structure:** Each file acts as its own module.
  - `import/export`: Used to share variables, functions, or classes between files.

Namespaces:

- **Definition:** Namespaces are a way to organize code within a **single file**. They are primarily used to group related code into a single entity, making it easier to manage large codebases in a single file.
- **Usage:** Namespaces are typically used in legacy TypeScript code or in cases where you want to avoid multiple files for modularization (though this is less common today).
- **Example:**

```typescript
namespace MathUtils {
  export function add(a: number, b: number): number {
    return a + b;
  }
}

console.log(MathUtils.add(2, 3)); // Output: 5
```

- **Key Points:**
  - **Single file:** Code is grouped inside a single namespace.
  - `namespace` **keyword:** Used to declare a namespace and organize functions or variables within it.

Key Differences:

- **Modules** are typically used in modern TypeScript to organize code across **multiple files**, while **namespaces** are used to organize code within **a single file**.
- **Modules** rely on the `import/export` system, while **namespaces** use the `namespace` keyword to

Q26: What are type guards in TypeScript? How are they used?

Type guards are a way to narrow down the type of a variable within a specific scope in TypeScript. They are used to tell the TypeScript compiler the type of a variable during runtime, which helps TypeScript perform more accurate type checking and prevent type errors.

What Are Type Guards?

A **type guard** is a mechanism that narrows the type of a variable to a more specific type within a conditional block (e.g., `if`, `switch`). Type guards allow TypeScript to infer more specific types within the scope where they are applied, improving type safety and enabling you to access properties specific to a certain type.

Common Type Guards:

1. `typeof`: You can use `typeof` to check the type of a variable (e.g., `string`, `number`, `boolean`).

```typescript
function printLength(value: string | number): void {
  if (typeof value === "string") {
    console.log(value.length);  // TypeScript knows 'value' is a string here
  } else {
    console.log(value.toFixed(2));  // TypeScript knows 'value' is a number here
  }
}
```

2. `instanceof`: You can use `instanceof` to check if an object is an instance of a class or a constructor function.

2. `instanceof`: You can use `instanceof` to check if an object is an instance of a class or a constructor function.

```typescript
class Dog {
  bark() {
    console.log("Woof!");
  }
}

function handleAnimal(animal: Dog | string): void {
  if (animal instanceof Dog) {
    animal.bark();  // TypeScript knows 'animal' is of type 'Dog' here
  } else {
    console.log(animal);  // TypeScript knows 'animal' is a string here
  }
}
```

3. **User-defined Type Guards:** A user-defined type guard is a function that narrows down the type of a variable. The return type of the function must be in the form of a **type predicate**: `variable is Type`.

```typescript
function isString(value: string | number): value is string {
  return typeof value === "string";
}

function printLength(value: string | number): void {
  if (isString(value)) {
    console.log(value.length);  // TypeScript knows 'value' is a string here
  } else {
    console.log(value.toFixed(2));  // TypeScript knows 'value' is a number here
  }
}
```

How Type Guards Are Used:

- **Type Narrowing:** Type guards narrow the type of a variable so that TypeScript can know which properties or methods are safe to access.
- **Preventing Errors:** They prevent runtime errors by ensuring that a variable is of a specific type before accessing its properties or methods.

Example of Type Guards in Action:

typescript Copy code

```typescript
function printValue(value: string | number): void {
  if (typeof value === "string") {
    console.log(value.toUpperCase());  // Safe to call 'toUpperCase' on a string
  } else {
    console.log(value.toFixed(2));  // Safe to call 'toFixed' on a number
  }
}
```

In this example, TypeScript uses the type guard `typeof value === "string"` to narrow down `value` to a string, so it can safely access the `toUpperCase` method. If the value is not a string, TypeScript knows it must be a number and allows access to the `toFixed` method.

Conclusion:

- **Modules** allow you to organize code across multiple files using `import` and `export`, while **namespaces** help organize code within a single file using the `namespace` keyword.
- **Type guards** in TypeScript are used to narrow the type of a variable at runtime, ensuring that specific properties or methods can be accessed safely. Common type guards include `typeof`, `instanceof`, and user-defined type guards.

Q27: What are some key options in `tsconfig.json`?

- `"strict": true`:
  - Enables all strict type-checking options.
  - Improves code safety by catching potential type issues early.
  - Includes options like `noImplicitAny`, `strictNullChecks`, and others.
- `"noImplicitAny": true`:
  - Disallows implicit `any` type.
  - Forces explicit type annotations for variables, parameters, and return values.
  - Helps prevent unintended use of `any`, ensuring better type safety.
- `"target": "ES6"`:
  - Specifies the ECMAScript version to compile the TypeScript code to.
  - Common options are `"ES5"`, `"ES6"`, `"ESNext"`, etc.
  - Determines features like arrow functions, `let`/`const`, and class syntax in the output JavaScript.
- `"module": "CommonJS"`:
  - Specifies the module system used for importing and exporting code.
  - Common options are `"CommonJS"`, `"ES6"`, `"ESNext"`, `"AMD"`, etc.
  - Affects how modules are handled, especially in Node.js environments.
- `"outDir": "./dist"`:
  - Specifies the output directory for compiled JavaScript files.
  - Ensures organized project structure by placing compiled files in a separate folder.
- `"esModuleInterop": true`:
  - Enables compatibility with CommonJS modules.
  - Allows default imports from modules that don't have a default export.
- `"baseUrl": "./src"`:
  - Defines the base directory for resolving non-relative module imports.
  - Helps in organizing and managing module imports in large projects.
- `"paths"`:
  - Allows custom paths for module resolution.
  - Enables you to map module imports to specific directories, improving project structure.

Q28: How does TypeScript handle errors?

- **Compile-Time Error Handling:**
  - TypeScript uses **static type-checking** to catch errors during the compilation process.
  - Errors related to type mismatches, missing properties, and incorrect function signatures are detected before runtime.

  Example:

```typescript
let message: string = 123;  // Error: Type 'number' is not assignable to type 'string'
```

- **Runtime Error Handling:**
  - TypeScript uses **JavaScript's** `try...catch` mechanism to handle runtime errors.
  - Errors can be caught and managed using standard JavaScript error-handling techniques.

  Example:

```typescript
try {
  throw new Error("Something went wrong");
} catch (error) {
  console.log(error.message);  // "Something went wrong"
}
```

- **Strict Typing for Error Handling:**
  - TypeScript's strict typing ensures that types are properly validated, reducing runtime errors caused by incorrect types or undefined values.
  - The `unknown` type is used to enforce checks on errors before accessing properties.

- **Strict Typing for Error Handling:**
  - TypeScript's strict typing ensures that types are properly validated, reducing runtime errors caused by incorrect types or undefined values.
  - The `unknown` type is used to enforce checks on errors before accessing properties.

Example:

```typescript
try {
  throw new Error("An error occurred");
} catch (error: unknown) {
  if (error instanceof Error) {
    console.log(error.message);  // Access 'message' safely
  }
}
```

- `never` **Type for Functions Throwing Errors:**
  - The `never` type is used for functions that never return, such as those that throw errors or cause infinite loops.

Example:

```typescript
function throwError(message: string): never {
  throw new Error(message);
}
```

Q29: How can TypeScript help in writing automation scripts?

- **Static Typing**:
  - Catches type-related issues at **compile time**, reducing the chance of runtime errors.
  - Ensures that variables, function arguments, and return values are used with the correct types.
- **IntelliSense and Autocompletion**:
  - Provides **autocompletion** and **IntelliSense** in IDEs for faster and more accurate script writing.
  - Reduces errors and increases developer productivity by suggesting valid methods, properties, and signatures.
- **Code Readability and Maintainability**:
  - Promotes better organization and structure using **interfaces**, **classes**, and **modules**.
  - Helps create more readable and maintainable automation scripts, especially in large codebases.
- **Error Checking During Development**:
  - Provides **compile-time checks** to catch common errors like undefined variables or type mismatches.
  - Ensures reliable execution of automation scripts by catching potential issues before runtime.
- **Refactoring Support**:
  - Robust refactoring tools ensure that changes to one part of the script don't break other parts.
  - Makes it easy to modify, extend, or improve automation scripts without introducing bugs.
- **Refactoring Support**:
  - Robust refactoring tools ensure that changes to one part of the script don't break other parts.
  - Makes it easy to modify, extend, or improve automation scripts without introducing bugs.
- **Integration with Testing Frameworks**:
  - Works seamlessly with popular **testing frameworks** like **Jest**, **Mocha**, and **Cypress** for writing automated tests.
  - **Type safety** improves test reliability by ensuring correct test setup and execution.
- **Modularization and Reusability**:
  - Encourages writing **modular**, reusable components for scalability.
  - Makes scripts easier to maintain and extend by breaking them into smaller, manageable parts.
- **Generics for Flexibility**:
  - Use **generics** to write reusable functions that work with different types, improving the flexibility of your automation scripts.
- **Decorators for Behavior Enhancement**:
  - **Decorators** can add custom behavior to functions or classes, such as logging, retry mechanisms, or timing, improving automation script functionality.
- **Better Integration with Build Tools**:
  - TypeScript integrates well with **build tools** like **Webpack**, **Gulp**, or **Grunt** for automating tasks such as testing, building, and deployment.
- **Advanced Language Features**:
  - TypeScript offers advanced features like **type aliases**, **enums**, and **namespaces**, which help in organizing and writing clean, scalable automation scripts.

Q33: How would you approach testing a new feature in JavaScript?

- **Understand the Requirements**:
  - Review documentation, user stories, and feature specifications to understand the desired behavior.
- **Write Unit Tests**:
  - Focus on testing individual components or functions in isolation.
  - Use tools like **Jest**, **Mocha**, or **Jasmine** to write unit tests.
- **Perform Integration Testing**:
  - Ensure that different components work together as expected.
  - Test interactions between modules, services, and APIs to validate end-to-end functionality.
- **Test Edge Cases and User Flows**:
  - Test the feature with both typical and edge-case inputs.
  - Simulate various user actions and workflows to ensure robust handling of different scenarios.

Q34: How would you handle flaky tests in your automation framework?

- **Identify the Root Cause**:
  - Investigate whether issues are due to **timing problems**, **network dependencies**, or **external factors** (e.g., third-party services or hardware).
- **Use Retries for Known Flaky Cases**:
  - Implement retry logic for tests that are known to be flaky or susceptible to timing issues.
  - Retry failed tests a few times before marking them as failed.
- **Mock Dependencies**:
  - Mock external dependencies (e.g., APIs, databases) to eliminate variability caused by network or external services.
  - Use tools like **Sinon**, **Jest Mock**, or **Nock** to mock network calls or service responses during tests.

Q35: How does JavaScript handle memory allocation and garbage collection?

- **Automatic Memory Allocation**:
  - JavaScript automatically allocates memory when objects and variables are created.
  - Developers do not need to manually allocate or deallocate memory.
- **Garbage Collection**:
  - JavaScript uses **garbage collection** to manage memory.
  - Common algorithms like **mark-and-sweep** are used to identify and free memory that is no longer accessible.
- **Mark-and-Sweep Algorithm**:
  - The **mark phase** marks all objects that are still referenced.
  - The **sweep phase** frees memory from unreferenced objects.
- **Example of Memory Leaks**:
  - **Unintentionally retained references**: Objects or variables that should have been removed are still held in memory.
  - **Circular references**: Two or more objects reference each other, preventing the garbage collector from freeing them.

Q36: What is currying in JavaScript?

- **Currying Definition**:
  - Currying is a technique that transforms a function with multiple arguments into a series of functions, each taking one argument.
- **Example**:

```javascript
function add(a) {
  return function(b) {
    return a + b;
  };
}

const add5 = add(5);
console.log(add5(3)); // Output: 8
```

- **Benefits of Currying**:
  - Allows for **partial application** of functions, enabling the creation of specialized functions with preset arguments.
  - Enhances **reusability** and **composability** in functional programming.

Q37: What is the difference between event bubbling and capturing?

- **Event Bubbling**:
  - Events propagate from the **target element** (the element that triggered the event) to the **root** of the DOM.
  - This is the default behavior for most events in JavaScript.
- **Event Capturing**:
  - Events propagate from the **root** of the DOM down to the **target element**.
  - Less commonly used but can be enabled by setting the third argument of `addEventListener` to `true`.
- **Example**:
  - **Bubbling** (default):

```javascript
element.addEventListener("click", handler, false); // Bubbling
```

  - **Capturing**:

```javascript
element.addEventListener("click", handler, true); // Capturing
```

Q38: Explain the difference between deep and shallow copies in JavaScript. How would you implement a deep copy?

- **Shallow Copy**:
  - A shallow copy creates a **new reference** to the original object or array.
  - Nested objects or arrays are not copied, only their references are.

Example:

```javascript
const original = { a: 1, b: { c: 2 } };
const shallowCopy = { ...original }; // Shallow copy

shallowCopy.b.c = 3;
console.log(original.b.c); // Output: 3 (because it's a reference)
```

- **Deep Copy**:
  - A deep copy creates a **completely new object** with copies of all the nested objects or arrays, ensuring no references to the original object.
  - This can be achieved by recursively copying each level of the structure.

Example:

```javascript
function deepCopy(obj) {
  return JSON.parse(JSON.stringify(obj)); // Simple deep copy using JSON methods
}

const original = { a: 1, b: { c: 2 } };
const deepCopyResult = deepCopy(original);

deepCopyResult.b.c = 3;
console.log(original.b.c); // Output: 2 (original is unaffected)
```

Q39: What is the Fetch API? How is it different from XMLHttpRequest?

- **Fetch API:**
  - A modern, **promise-based** API for making HTTP requests.
  - Offers a simpler and more flexible way to handle asynchronous requests compared to **XMLHttpRequest**.
  - Returns a **promise**, making it easier to work with asynchronous code using `.then()` or `async/await`.
- **Differences from XMLHttpRequest:**
  - **Promises:** Fetch uses promises, allowing cleaner syntax with `.then()` and `async/await`, while `XMLHttpRequest` uses callback functions.
  - **Simpler Syntax:** Fetch has a simpler and more readable API compared to the verbose `XMLHttpRequest`.
  - **Response Parsing:** Fetch has built-in methods like `.json()` to parse responses, while with `XMLHttpRequest`, parsing requires additional steps.
  - **No Need for Manual Event Handling:** Fetch handles request state changes internally, reducing boilerplate code.
- **Example:**

javascript Copy code

```javascript
fetch("https://api.example.com/data")
  .then((response) => response.json())  // Parse response as JSON
  .then((data) => console.log(data))    // Handle the parsed data
  .catch((error) => console.error(error)); // Handle any error
```

Q40: What are utility types in TypeScript? Name some commonly used ones.

- **Answer:**
  Utility types are built-in TypeScript types that help transform types.
  - `Partial<T>`: Makes all properties optional.
  - `Readonly<T>`: Makes all properties read-only.
  - `Pick<T, K>`: Selects specific keys.
  - `Omit<T, K>`: Excludes specific keys.

  Example:

typescript Copy code

```typescript
interface User {
  name: string;
  age: number;
  email: string;
}
const partialUser: Partial<User> = { name: "Alice" };
```

Q41: What are mapped types in TypeScript?

- **Mapped Types:**
  - Allow creating new types by transforming the properties of an existing type.
  - Provide a way to apply transformations (like making properties `readonly`, `optional`, etc.) to all properties of a given type.
- **Syntax:**

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

  - In this example, `ReadonlyUser<T>` creates a type where all properties of the given type `T` are **read-only**.
- **Common Use Cases:**
  - Making properties of an existing type optional or required.
  - Applying other transformations like making properties nullable or readonly.

Q42: What are discriminated unions in TypeScript?

- **Discriminated Unions:**
  - A type system feature that allows creating a union of types where each type in the union has a **discriminant property** that helps narrow down which type is being used.
  - The discriminant property has a unique value for each type, enabling TypeScript to differentiate between them.
- **Example:**

```typescript
type Shape =
  | { type: "circle"; radius: number }
  | { type: "rectangle"; width: number; height: number };

function area(shape: Shape) {
  if (shape.type === "circle") {
    return Math.PI * shape.radius ** 2;
  } else {
    return shape.width * shape.height;
  }
}
```

  - In this example, the `type` property is the **discriminant** that distinguishes between a `circle` and a `rectangle`.
  - TypeScript uses this property to infer the type of the object and enable type-safe access to other properties.

Q44: How would you handle runtime errors in JavaScript applications?

- **Use `try...catch` blocks:**
  - Wrap code that may throw errors inside a `try` block, and handle exceptions in the `catch` block.
  - Ensures that the application doesn't crash when encountering an error.
- **Log errors to an error monitoring tool:**
  - Use services like **Sentry**, **LogRocket**, or **Rollbar** to monitor and report runtime errors in production.
  - Helps track and resolve issues that users may encounter in real-time.
- **Example:**

```javascript
try {
  JSON.parse("{ invalid json }");
} catch (error) {
  console.error("Parsing error:", error.message);
  // Optionally send error details to an external monitoring service
  // e.g., Sentry.captureException(error);
}
```

  - The example shows how to handle a `JSON.parse()` error gracefully, log the error to the console, and optionally send the error details to an external service like **Sentry** for further analysis.

Q46: How do you debug JavaScript code?

- **Use `console.log()`:**
  - Insert `console.log()` statements at different points in the code to trace the values of variables and understand the flow of execution.
- **Use browser developer tools:**
  - Inspect the DOM and monitor the console for errors.
  - Use the **Sources** tab to view and step through your code, set breakpoints, and inspect variables.
- **Use breakpoints and watch expressions:**
  - In the browser's developer tools, you can set breakpoints in your JavaScript code to pause execution at specific points.
  - Watch expressions to monitor the values of variables as your code runs.

Q47: How do you ensure compatibility across different browsers?

- **Use polyfills and transpilers:**
  - **Polyfills:** Provide fallback implementations for newer features that may not be supported in older browsers.
  - **Transpilers** (e.g., **Babel**): Convert modern JavaScript code (ES6 and beyond) into older versions that work across more browsers.
- **Use tools like BrowserStack or Sauce Labs:**
  - These are cloud-based testing platforms that allow you to test your web applications across various browsers and devices.
- **Test with feature detection using Modernizr:**
  - **Modernizr** is a JavaScript library that detects whether a browser supports specific features and allows you to apply conditional logic based on that.

Q53: How do you optimize JavaScript for performance?

- **Answer:**
  - Use **debouncing** and **throttling** for event listeners.
  - Minimize DOM manipulation by batching updates.
  - Use **lazy loading** for images and assets.
  - Avoid memory leaks by clearing timers and event listeners.

Debouncing Example:

javascript Copy code

```javascript
function debounce(func, delay) {
  let timeout;
  return function (...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), delay);
  };
}
```

Q54: Explain conditional types in TypeScript. Provide examples.

- **Conditional Types:**
  - Allow creating types based on a condition (similar to ternary operators in JavaScript).
  - The condition is evaluated using `extends`, and the result can be one type or another.
- **Example:**

```typescript
type IsString<T> = T extends string ? "Yes" : "No";
type Test1 = IsString<string>; // "Yes"
type Test2 = IsString<number>; // "No"
```

  - In this example, `IsString` checks if the type `T` is a `string`. If it is, it returns `"Yes"`, otherwise `"No"`.
- **Use Cases:**
  - Conditional types are useful for handling type differences based on parameters, enabling more flexible and dynamic types.

Q55: What is the difference between union and intersection types?

- **Union Types (`|`):**
  - A **union** type allows a value to be **one of several types**.
  - Syntax: `TypeA | TypeB`
  - **Example:**

```typescript
type A = string | number;
let value: A = 42; // or "Hello"
```

  - In this example, `value` can either be a `string` or a `number`.
- **Intersection Types (`&`):**
  - An **intersection** type requires a value to **satisfy all types** it intersects.
  - Syntax: `TypeA & TypeB`
  - **Example:**

```typescript
type B = { name: string } & { age: number };
const person: B = { name: "Alice", age: 30 };
```

  - In this example, `person` must have both `name` and `age` properties, combining both types into one.
- **Summary:**
  - **Union Types**: One of multiple types.
  - **Intersection Types**: All types combined into a single type.

Q56: How do generics work in TypeScript? Why are they useful?

- **Generics**:
  - Allow you to create reusable, type-safe components that work with a variety of types.
  - **Syntax**: The type parameter (e.g., `T`) is used to represent a placeholder for a type that will be provided when the function or class is used.
- **Example**:

```typescript
function identity<T>(value: T): T {
  return value;
}
const num = identity<number>(42); // T is inferred as number
const str = identity<string>("Hello"); // T is inferred as string
```

- **Use Case**:
  - **Reusability**: Generics enable you to write functions, classes, or interfaces that can be reused with different types without losing type safety.
  - **Type Safety**: Ensure that the types remain consistent across different parts of the code, reducing errors at compile time.

---

3. How you can declare a explicit variables in Typescript?

In typeScript, you can declare static variables using the colons **(:)** followed by the data type of the explicit type. You can not assign a value of some other data type to a static variable. The values of the same data type can be assigned.

Syntax:

```
let variable_name: data-type = value;
```

Example:

```
let company_name: string = "GeeksforGeeks";
company_name = "Cricket";
console.log(company_name); // Prints Cricket

company_name = 28;
console.log(company_name);
// Throws an error: Type '28' is not assignable to type 'string'.
```

4. How to declare a function with typed annotation in TypeScript?

In TypeScript, you can declare the functions by defining the type of parameters the take and the type of parameter it will return after the execution.

Example:

```
function annotedFunc(myName: string, age: number): string{
    return `My name is ${myName} and my age is ${age}.`;
}
console.log(annotedFunc("Emrit", 22));

// Prints: My name is Emrit and my age is 22.
console.log(annotedFunc("Neha", "18"));

// Above statement throws an error:
// Argument of type '"18"' is not assignable to parameter of type 'number'.
```

5. Describe the "any" type in TypeScript.

The **any** data type will allow you to assign the value of any data type to a variable. Sometimes, when the data is coming from other resouces like API call or user entered data. In that case, you may not aware of the type of the data so you can use the any data type to assign the value of any kind to a variable.

Example:

```
let studentData: string = `{
    "studentName": "Aakash",
    "studentID": 12345,
    "studentCourse": "B. Tech"
}`;
let student: any = JSON.parse(studentData);

console.log(student.studentName, student.studentID, student.studentCourse);
// Prints: Aakash 12345 B. Tech
```

7. List some disadvantages of using TypeScript.

There also exist some disadvantages of using TypeScript as listed below:

- The concept of abstract classes is not supported by TypeScript.
- Code Compilation is a time taking process in TypeScript.
- A extra step of converting the TypeScript code into JavaScript code requires while running TypeScript.
- A definition file needs to be added for using any external or third party library. All the external libraries not have the definition file.
- The quality of all the definition files need to be correct.

8. Explain the void type in TypeScript.

It is just opposite of **any** type. The **void** type represents the unavailability of the data type for any variable. It is mainly used with the functions that returns nothing. The variables defined using the void keyword can only be assigned with the **null** and **undefined** values.

Example:

```
function favGame(): void{
    console.log("My Favourite game is Cricket.");
}
favGame();
// Prints: My Favourite game is Cricket.
```

9. What is type null and its use in TypeScript?

The **null** keyword is considered as a data type in TypeScript as well as in JavaScript. The null keyword basically indicates the unavailability of a value. It can be used to check whether a value is provided to a particular variable or not.

Example:

```
function getData(orgName: string | null, orgDesc: string | null): void {
    if (orgName === null || orgDesc === null) {
        console.log("Not enough values provided to print.");
    }
    else {
        console.log(`Organization Name: ${orgName},
        \nOrganization Description: ${orgDesc}`);
    }
}

getData(null, null);
getData("GeeksforGeeks",
    "A Computer Science Portal.");
```

Output:

```
Not enough values provided to print.
Organization Name: GeeksforGeeks,
Organization Description: A Computer Science Portal.
```

10. Describe the syntax for creating objects in TypeScript.

A object is basically a collection of key-value pairs, where each key needs to be unique. In TypeScript, the objects can be created by declaring the property name and the type it is going to store.

Example:

```
const myObj: { name: string, desc: string } = {
    name: "GeeksforGeeks",
    desc: "A Computer Science Portal"
};
console.log(myObj);
// Prints: { name: 'GeeksforGeeks', desc: 'A Computer Science Portal' }
```

11. Can we specify the optional properties to TypeScript Object, if Yes, explain How?

Yes, we can declare the TypeScript Objects by specifying the optional properties that may or may not be defined inside the object. We can declare these peoperties by using a **?** symbol just after the property name while creeating the object.

Example:

```
const myObj: { name: string, desc: string, est?: number } = {
    name: "GeeksforGeeks",
    desc: "A Computer Science Portal",
};
console.log(myObj);
// Prints: { name: 'GeeksforGeeks', desc: 'A Computer Science Portal' }
myObj.est = 2008;
console.log(myObj);
// Prints: { name: 'GeeksforGeeks', desc: 'A Computer Science Portal' , est: 2008}
```

12. Explain the undefined type in TypeScript.

The concept **undefined** in TypeScript is similar to the concept of undefined in JavaScript. The undefined is used to indicate a variable which is declared but not assigned a value and it's either hoisted or in temporal dead zone. The memory to these kind of variables is allocated in the memory allocation phase and a **undefined** value assigned to them untill a value is assigned by the developer or programmer.

13. Explain the behavior of arrays in TypeScript.

The arrays defined in typescript are different from JavaScript and they also behave differently from JavaScript arrays. In typescript, the arrays are defined by specifying the static data types and can only store the multiple values of a single kind of data type.

Example:

```
const typedArray1:number[] = [1, 23, 28, 56];console.log(typedArray1);
  // 1, 23, 28, 56
const typedArray2:number[] = [1, 23, 28, 56, "GeeksforGeeks"];
console.log(typedArray2);
 // Throws an error: Type 'string' is not assignable to type 'number'.
```

14. How can you compile a TypeScript file?

To compile a TypeScript file, you use the tsc (TypeScript Compiler) command. You can compile a TypeScript file by running:

```
tsc filename.ts
```

This command compiles the .ts file into a .js (JavaScript) file. Make sure to have TypeScript installed globally or locally in your project for this to work.

15. Differentiate between the .ts and .tsx file extensions given to the TyppeScript file.

The **.ts** file extension is used to create a file that contains the pure TypeScript code inside it. These files are mainly created to implement the classes, functions, reducers and other pure typescript code. These files does not contain any JSX code. On the other hand, the .**tsx** file extensions are used to create the files that contains the JSX code inside it. These files are mainly used to build a react component that returns the JSX code at the end.

16. What is "in" operator and why it is used in TypeScript?

The **in** opertaor is used to check whether a property is present in the testing object or not. It will return true, if it finds the property in the object. Otherwise, it will return false.

17. Explain the union types in TypeScript?

The union types in TypeScript represents that the value of a variable can be one of the specified types. The union type uses a **straight vertical bar(|)** to show the options for the variable types.

18. Explain type alias in TypeScript?

A type alias in TypeScript is used to give a new and the meaning fule name for a combined or new type. The type alias will not create a new type instead it create new names for the type it refers to.

```
type combinedType = number | boolean
```

19. Is TypeScript strictly statically typed language?

No, TypeScript is not a strictly statically typed language it is an optional statically typed language that means it is in our hands that a particular variable has to be statically typed or not. We can use the **any** type and allow a variable to accept the value of any kind of data type. We can also define a variabe with a particular data type that a variable can accept and throws error if a value of some other data type is assigned to it.

20. Is template literal supported by TypeScript?

Yes, template literal is supported by TypeScript. We can interpolate a string using the template literals syntax (``) in TypeScript. The values of different variables can be shown inside a string using **${variable_name}** syntax while interpolating strig using template literals.

21. How to declare a arrow function in TypeScript?

In TypeScript, we can declare the arrow functions in the same format as we declare in vanilla JavaScript. TypeScript allow us to use the typed declaration with by specifying the type of parameters and the type of return value as well.

Example:

```
const typedArrowFunc = (org_name: string, desc: string): string => {
    let company: string = `Organization: ${org_name}, Description: ${desc}`;
    return company;
}
console.log(typedArrowFunc("GeeksforGeeks", "A Computer Science Portal"));
// Prints: Organization: GeeksforGeeks, Description: A Computer Science Portal
```

22. How to define a function which accepts the optional parameters?

You can define a function with optinal parameters by using a **?** symbol in the declaration of the function for the parameter which you want to make the optional as shown below:

Example:

```
function cricketer(c_name: string, runs?: number): void{
    if(!runs){
        console.log(`Cricketer Name: ${c_name}, Runs Scored: Not Available`);
    }
    else{
        console.log(`Cricketer Name: ${c_name}, Runs Scored: ${runs}`);
    }
}
cricketer("Virat Kohli", 26000);
cricketer("Yuzvender Chahal");
// Prints:
// Cricketer Name: Virat Kohli, Runs Scored: 26000
// Cricketer Name: Yuzvender Chahal, Runs Scored: Not Available
```

23. Explain noImplicitAny in TypeScript.

The **noImplicitAny** is a compiler option that we can add to the tsconfig.json file and make the TypeScript compiler to throw an error if a function or method for a particular type is invoked on a parameter of any type.

Generally, TypeScript expect a explicit type to be associated with the parameters, but somethimes we don't need to specify the explicit type. In that case, TypeScript assigns a explicit type as **any type** to that parameter and allows to perform operations. But we can force the compiler to throw an error if a explicit type is not defined for a parameter.

24. What are interfaces in TypeScript?

A interface in TypeScript is used to define a syntax that must be followed by the entity of that interface. An Interface defines the properties, methods and the events and considered as the members of the interface. The interfaces only contain the declarations of the members. The initialisation or the assignment will be done by the class that is deriving the interface. Interfaces are defined using the **interface** keyword.

25. In how many ways you can use the for loop in TypeScript?

There are mainly three ways in which you can use the for loop in TypeScript as listed below:

- **Using the simple for loop:** It is the simple for loop used to iterate through any number of variables by defining and using a variable.

```
for(let i=0; i<n; i++){ // code statement}
```

- **Using the for-of loop:** It can be used to iterate through the array elements without defining the iterator variable.

```
for(let item of myArr){ // code statement}
```

- **Using the forEach() method:** It is a method that takes a callback function and operate the functionality to each item of the array.

```
myArr.forEach(() => { // code statement})
```

26. What is never type and its uses in TypeScript?

A **never** type in typescript is indicate the values that may never be occured. It is mainly used with the function that return nothing and always thrown an exception or error. A **never** type is different from **void** type. Because, a function that returns nothing implicitly returns **undefined** and these functions are inferred using the **void** keyword. But a function that declared using the **never** keyword will never return a undefined it only returns never type. The never type can be used with following cases:

- With an infinite loop.
- In a function that throws exceptions or errors.

Example:

```
function neverFunc(): never{
    // Function Statements
}
```

27. Explain the working of enums in TypeScript?

An **enum** in typescript is used to create a collection of constants. It is basically a class that allow us to create multiple constants of **numeric** as well as **string type.** By default, the value of numeric constant starts from **0** and increases accordingly for every constant by a margin of **1**. You can also change the initialisation value from 0 to any other value of your choice. It is declared using the **enum** keyword followed by the name of enum and constants.

Example:

```
enum demoEnum{
    milk = 1,
    curd,
    butter,
    cheese
}
let btr: demoEnum = demoEnum.butter;
console.log(btr)
// Prints: 3
```

28. Explain the parameter destructuring in TypeScript.

The parameter destructuring is nothing more than the unpacking of the provided or passed object properties individually into the one or more parameters when the object is passed to an function. It can be done as shown below:

```
function getOrganisation({ org_name, org_desc }: { org_name: string, org_desc: string }) {
    console.log(`Organization Name: ${org_name}, \nOrganization Description: ${org_desc}`);
}
getOrganisation({ org_name: "GeeksforGeeks", org_desc: "A Computer Science Portal." });

// Prints:
// Organization Name: GeeksforGeeks,
// Organization Description: A Computer Science Portal.
```

Example:

```
interface interface_name{
    // Define the members like methods properties and events etc.
}
```

29. Explain type inference in TypeScript.

The type inference refers to the automatically assigning the explicit types to the variable which does not declared using the expicit type. Generally, it is done at the time when the variables are declared and initialized at the same time with a value of some data type.

30. What are modules in TypeScript?

Modules are used to create a collection of multiple data types that may include the classes, functions, interfaces and variables. The modules have their own scope. The members defined inside modules can not be accessed directly by the other code. Modules are imported using the **import** statement and defined using the **export** keyword to export it.

Example:

```
module multiply{
    export function product(a: number, b: number): number{
        return a*b
    }
}
```

31. In how many ways you can classify Modules?

There are two types of Modules available in TypeScript:

- **Internal Modules:** These modules are used to specify the collection of classes, interfaces, functions and variables that can be exported to other modules as a single unit.
- **External Modules:** These are the separate TypeScript files that include more code and consist of at least one export or import statement in it.

32. What is the use of tsconfig.json file in TypeScript?

This file helps to compile the project by providing the compiler options. It also makes the working directory as the root directory of the project.

34. How to debug a TypeScript file?

A TypeScript file can be debugged using the **--sourcemap** command that is followed by the file name. It will create a new file with the **fileName.ts.map** name.

```
tsc --sourcemap script.ts
```

35. Describe anonymous functions and their uses in TypeScript?

The anonymous functions are the functions that are declared without a name. These functions are mainly used to pass to another function as a call back function like while attaching an event and calling the setTimeout() method etc.

Example:

```
myBtn.addEventListener('click', function () {
    // Add click functionality
});
```

Example:

```
let unionVar: number | boolean = true;
console.log(unionVar); // true

unionVar = 56;
console.log(unionVar); // 56
```

36. Is it possible to call the constructor function of the base class using the child class?

Yes, the base class constructor can be called using the **super()** method inside the constructor function of the child class with the required parmeters if the base class constructor function takes parameters.

37. How to combine multiple TypeScript files and convert them into single JavaScript file?

There is a command named **--outfile** that is follwed by the JavaScript filename and the multiple TypeScript files. If the JavaScript file name is not provided then all the TypeScript files are combined in the first TypeScript file specified in the format.

```
tsc --outfile combined.js script1.ts script2.ts script3.ts
```

38. Explain type of operator in TypeScript and where to use it.

The **typeof** operator is used to check or get the type of a particular variable. It can also be used to set the similar explicit type to another variables.

Example:

```
const strVar: string = "GeeksforGeeks";
const numVar: number = 28;
console.log(typeof strVar, typeof numVar); //Prints: string number

const numVar2: typeof numVar = 25;
const strVar2: typeof strVar = "Cricket";
console.log(typeof strVar2, typeof numVar2); //Prints: string number
```

39. How you can compile a TypeScript file?

The TypeScript code is not executed directly. It requires to convert the TypeScript code into JavaScript. You can use **tsc <file_name>** command to execute the TypeScript code and convert it into JavaScript.

```
tsc script.ts
```

40. Which principles of Object Oriented Programming are supported by TypeScript?

TypeScript supports all the four principles of Object Oriented Programming as:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

41. Explain Mixins in TypeScript

The mixins are used to create the classes from the reuseable components. They can be built using the different partial classes. In simple language, we can say that instead of a class extends another class, a function can take a class and then return the class as a result. The function that takes the class and returns a class is known as mixin.

42. Is it possible to create the immutable Object properties in TypeScript?

Yes, you can use the **readonly** property before the properties names while defining the objects. It makes the properties of object to be initialized at the time of declaring the object. If you try to update the value of any immutable property it will not allow you to do it as it is a readonly property.

TypeScript Interview Questions For Experienced

43. In what situation you should use a class and a interface?

We can use the classes and interfaces to define our own custom data types. There are different use cases and situations where we can use the interfaces and classes.

An interface can be used in a situation where the shape, structure and the contact will be defined for indicating how a object or class will look like without their implementation. It can also be used to enforce some properties and methods to a class and object.

On the other hand, a class can be used where we need to enclose the data to hide it from outer code and prevent the direct acces of this data. It can be used to implement the Object Oriented Programmin concepts.

44. What are the differences between the classes and the interfaces in TypeScript?

A class is defined using the **class** keyword. The classes can contain the methods, properties and variables. Methods of a class are defined when the class is implemented. A class instance will allow us to access the properties and methods defined inside the class.

Interfaces are defined using the interface keyword. It contains only the declarations of the properties and methods which are implemented by the derived class.

45. How to declare a class in TypeScript?

The syntax for declaring the classes in TypeScript is almost same as in JavaScript. In TypeScript, you can also use the typed declarations for the variables and methods of the class.

Example:

```
 class Cricketer{
    name: string;
    runs: number;

    constructor(name: string, runs: number){
        this.name = name;
        this.runs = runs;
    }

    thisMatchRund(): number{
        this.runs += 139;
        return this.runs;
    }
}
```

46. How the inheritance can be used in TypeScript?

Inheritance is one of the main four pillars of Object Oriented Programming. It allows a class to inherit or acquire the properties and methods of other class and implement them with the instance created using the inherited class. The inheritance can be implemented using the **extends** keyword after the child class name followed by the parent class name.

```
class Child extends Parent{
    // Properties of child class
}
```

47. What are the different ways for controlling the visibility of member data?

TypeScript provide us with the three ways to control the visibility of members like methods and properties in classes.

- **Private:** The private members can be accessed only inside the class. They can not be accessed by the outer code.
- **Public:** It is the default visibilty of the members. These members can be access from any where in the code by defining the class instance.
- **Protected:** These members can only be accessed by the classes that are inherited using the members class. No other code that does not inherit the class can access these members.

48. How to convert a .ts file into TypeScript Definition file?

You can change a .ts file into the definition file with the help of TypeScript compiler by using the **--declaration** command followed by the name od the .ts file. The definition file will make the TypeScript file reuseable.

```
tsc --declaration script.ts
```

50. Expalin conditional typing in TypeScript?

In TypeScript, we can use the ternary operator(condition? true:false) to assign the dynamic types to an property. It will assign a type dynamically based on the condition defined in the ternary operator.

TypeScript Interview Questions - FAQs

Is TypeScript similar to JavaScript?

Yes, TypeScript is a superset of JavaScript, adding optional static typing and extra features on top of JavaScript

Do I need to learn TypeScript if I know JavaScript?

Not mandatory, but TypeScript offers advantages like early error detection, better code maintainability, and productivity, especially in larger projects.

Are TypeScript's types mandatory?

No, types are optional in TypeScript, allowing you to add them gradually for improved code quality and developer experience.

Can I use JavaScript libraries with TypeScript?

Yes, TypeScript works with JavaScript libraries and provides type definitions for many. You can also create custom type definitions if needed.

Is TypeScript suitable for all projects?

TypeScript is ideal for large, scalable projects needing high maintainability but can be used effectively for projects of any size

14. Define static typing

Static typing refers to a compiler that has recognisable variables, arguments, and object members at compile time. This aids in the early detection of faults.

15. What are import and export keywords in TypeScript?

- Import keyword is used to export declaration. Example: export * from module
- Any variable, function, or type alias can be exported with the export keyword

16. Can TypeScript be used for the backend?

By combining TypeScript with Node.js, backend applications can benefit from the added security that the language provides.

17. How will you check if a variable is null or undefined in TypeScript?

if(value) return true if value is not null, undefined, empty, false, 0 or NaN.

18. What are getters/setters?

```
class Person {
  private _age: number;
  private _firstName: string;
  private _lastName: string;

  public get age() {
      return this._age;
  }

  public set age(theAge: number) {
      if (theAge <= 0 || theAge >= 200) {
          throw new Error('The age is invalid');
      }
      this._age = theAge;
  }
```

Getters and setters prevent access to an object's members.They allow you to have more precise control over how a member interacts with each object. A getter method starts with keyword ‘get’ and setter method starts with keyword ‘set’

19. How can a class constant be implemented in TypeScript?

Class properties cannot be declared with the const keyword. The keyword 'const' cannot be used in a class member.

20. What is the Declare Keyword in TypeScript?

Since JavaScript lacks a TypeScript declaration, the declare keyword is used to include it in a TypeScript file without causing a compilation issue. Ambient methods and declarations use the term to define a variable that already exists.

21. What is an Interface with reference to TypeScript?

The interface specifies the syntax that classes must use. All of the members of an interface are implemented by a class that implements it. It's possible to refer to it, but not to use it. A type-checking interface is used by the TypeScript compiler.

22. Describe ‘as’ syntax in TypeScript.

In TypeScript, the 'as' syntax is used for Type assertion. It was created because the original syntax was incompatible with JSX. Only as-style assertions can be used with JSX and TypeScript.

Example:

```
let stdid: any=007

let stdid= id as number;
```

23. Define Lambda function.

For defining function expressions, TypeScript provides a shortcut syntax. A lambda function is an unnamed anonymous function.

Example:

```
let sum=(a: num, b: num): num=>{ return a+b;}

console.log(sum(5,10)); //returns 15
```

Here, ?=>? is a lambda operator.

24. How to create objects in Typescript?

Objects are collections of keys and values that resemble a dictionary. The keys must be one-of-a-kind. They resemble arrays and are sometimes referred to as associative arrays. An array, on the other hand, employs numbers to index the values, whereas an object lets you use any type as the key.

An Object type in TypeScript refers to any value with properties. It can be defined simply by specifying the properties and the kinds of those properties. As an example,

```
let pt: { x: number; y: number } = {

  x: 10,

  y: 20

};
```

27. Explain Tuples in Typescript With Example

Tuples are a collection of values that are diverse. It allows for the storage of many fields of various sorts. Tuples can also be used as function parameters.

There are instances when it is necessary to save a collection of values of various types. Arrays will not suffice in this situation. TypeScript provides a data type called tuple that aids in this endeavor.

Syntax:

var tuple_name = [value c,value b,value c,...value n]

For Example:

var yourtuple = [12,"Hi"];

28. What is Anonymous Function in TypeScript?

Anonymous functions are those that have no identifier (function name) attached to them. At runtime, these functions are dynamically declared. Anonymous functions, like normal functions, can accept inputs and return outputs. After its initial creation, an anonymous function is normally inaccessible. An anonymous function can be assigned to variables.

Syntax:

```
var res = function( [arguments] ) { ... }
```

Example:

```
var msg = function() {
   return "hello world";
}
console.log(msg())
```

33. What is Namespace and how to declare it?

The namespace is used to group functionalities logically. To enable a single or a group of linked functionalities, a namespace can include interfaces, classes, functions, and variables.

The namespace keyword, followed by the namespace name, can be used to construct a namespace. Curly brackets can be used to define all interfaces, classes, and other objects.

Syntax:

namespace <name>

{

}

34. What are Rest Parameters in Typescript?

- You can use rest parameters when the number of parameters that a function will get is unknown or varies.
- The rest parameter can take zero or more parameters. The compiler will generate an array of arguments containing the name of the rest parameter.

```
let Greet = (greeting: string, ...names: string[]) => {

 return greeting + " " + names.join(", ") + "!";

}

Greet("Hi!", "John", "Sam"); // returns "Hi John, Sam"

Greet("Hi!");// returns "Hi !
```

39. What are Conditional Types in TypeScript ?

Based on a condition given as a type relationship test, a conditional type chooses one of two alternative types:

T extends U ? X : Y

When T can be assigned to U, the type is X, and when it can't, the type is Y.

Because the condition depends on one or more type variables, a conditional type T extends U? X: Y and is either resolved to X or Y or delayed. Whether to resolve to X or Y, or to defer, when T or U contains type variables is determined by whether the type system has enough information to conclude that T is always assignable to U.

40. What are Distributive Conditional Types?

Distributive conditional types are conditional types in which the checked type is a bare type parameter. During instantiation, distributive conditional types are automatically distributed over union types.

For example, an instantiation of T extends U ? X : Y with the type argument A | B | C for T is resolved as (A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y).

41. Explain how TypeScript files can be supported from Node Modules

TypeScript includes a series of declaration files to guarantee that TypeScript and JavaScript support works well right out of the box (.d.ts files). The various APIs in the JavaScript language, as well as the standard browser DOM APIs, are represented in these declaration files. While there are some fair defaults based on your target, you can configure the lib setting in the tsconfig.json to specify which declaration files your program uses.

TypeScript has a feature similar to @types/ support that allows you to override a specific built-in lib. TypeScript will check for a scoped @typescript/lib-* package in node modules when selecting which lib files to include. After that, you can use your package manager to install a specific package to take over for a particular library.

44. Explain the Drawbacks of using Declaration Files with Typescript

There are two drawbacks to using these declaration files with TypeScript:

Since you upgrade TypeScript, you must also deal with changes to TypeScript's built-in declaration files, which can be difficult when the DOM APIs change so regularly.

Customizing these files to meet your needs and the demands of your project's dependencies is difficult (e.g. if your dependencies declare that they use the DOM APIs, you might also be forced into using the DOM APIs).

45. How to compile Typescript with Visual Studio Code?

- Visual Studio Code includes TypeScript language support but does not include the TypeScript compiler.
- You need to install the TypeScript compiler either globally or in your workspace to transpile TypeScript source code to JavaScript
- The easiest way to install TypeScript is through npm, the Node.js Package Manager.
  If you have npm installed, you can install TypeScript globally (-g) on your computer by:

```
npm install -g typescript
```

- You can test your install by checking the version or help.

```
tsc --version
```

49. What is Type Assertion? Explain its types

You can find yourself in a scenario where you know a type for an entity that is more specific than its present type.

A type assertion is similar to a type cast in other languages, but it does not do any additional data verification or restructuring. It has no effect on runtime and is only used by the compiler. TypeScript expects that you, the programmer, have completed any necessary specific checks.

There are two types of type assertions.

One is the as-syntax:

```
let someValue: unknown = "this is a string";

let strLength: number = (someValue as string).length;
```

The other version is the “angle-bracket” syntax:

```
let someValue: unknown = "this is a string";

let strLength: number = (<string>someValue).length;
```

Both samples are identical. selecting one over the other is basically a matter of preference; however, only as-style assertions are allowed when combining TypeScript with JSX.

50. What is Recursive Type Aliases?

The ability to "recursively" reference type aliases has always been limited. The reason for this is because each type alias must be capable of substituting itself for whatever it aliases. Because this isn't always possible, the compiler rejects some recursive aliases.

Interfaces can be recursive, but their expressiveness is limited, and type aliases cannot. That involves combining the two: creating a type alias and extracting the type's recursive portions into interfaces. It's effective.

```
type ValueOrArray<T> = T | ArrayOfValueOrArray<T>;

interface ArrayOfValueOrArray<T> extends Array<ValueOrArray<T>> {}
```

What's the difference between JavaScript and Java?

JavaScript	Java
JavaScript is an object-oriented scripting language.	Java is an object-oriented programming language.
JavaScript applications are meant to run inside a web browser.	Java applications are generally made for use in operating systems and virtual machines.
JavaScript does not need compilation before running the application code.	Java source code needs a compiler before it can be ready to run in realtime.

3. What are the various data types that exist in JavaScript?

These are the different types of data that JavaScript supports:

- Boolean - For true and false values
- Null - For empty or unknown values
- Undefined - For variables that are only declared and not defined or initialized
- Number - For integer and floating-point numbers
- String - For characters and alphanumeric values
- Object - For collections or complex values
- Symbols - For unique identifiers for objects

4. What are the features of JavaScript?

These are the features of JavaScript:

- Lightweight, interpreted programming language
- Cross-platform compatible
- Open-source
- Object-oriented
- Integration with other backend and frontend technologies
- Used especially for the development of network-based applications

5. What are the advantages of JavaScript over other web technologies?

These are the advantages of JavaScript:

Enhanced Interaction

JavaScript adds interaction to otherwise static web pages and makes them react to users' inputs.

Quick Feedback

There is no need for a web page to reload when running JavaScript. For example, form input validation.

Rich User Interface

JavaScript helps in making the UI of web applications look and feel much better.

Frameworks

JavaScript has countless frameworks and libraries that are extensively used for developing web applications and games of all kinds.

6. How do you create an object in JavaScript?

Since JavaScript is essentially an object-oriented scripting language, it supports and encourages the usage of objects while developing web applications.

```
const student = {

   name: 'John',

   age: 17

}
```

7. How do you create an array in JavaScript?

Here is a very simple way of creating arrays in JavaScript using the array literal:

```
var a = [];

var b = ['a', 'b', 'c', 'd', 'e'];
```

9. What are the scopes of a variable in JavaScript?

The scope of a variable implies where the variable has been declared or defined in a JavaScript program. There are two scopes of a variable:

Global Scope

Global variables, having global scope are available everywhere in a JavaScript code.

Local Scope

Local variables are accessible only within a function in which they are defined.

10. What is the 'this' keyword in JavaScript?

The 'this' keyword in JavaScript refers to the currently calling object. It is commonly used in constructors to assign values to object properties.

11. What are the conventions of naming a variable in JavaScript?

Following are the naming conventions for a variable in JavaScript:

- Variable names cannot be similar to that of reserved keywords. For example, var, let, const, etc.
- Variable names cannot begin with a numeric value. They must only begin with a letter or an underscore character.
- Variable names are case-sensitive.

12. What is Callback in JavaScript?

In JavaScript, functions are objects and therefore, functions can take other functions as arguments and can also be returned by other functions.

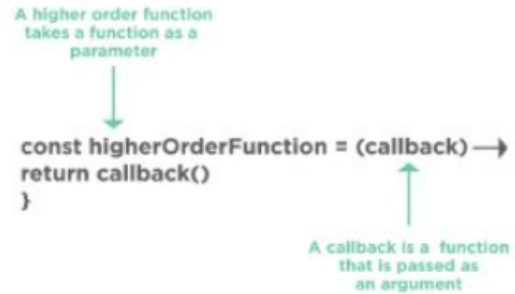

Fig: Callback function

A callback is a JavaScript function that is passed to another function as an argument or a parameter. This function is to be executed whenever the function that it is passed to gets executed.

13. How do you debug a JavaScript code?

All modern web browsers, such as Chrome, Firefox, etc., have an inbuilt debugger that can be accessed anytime by pressing the relevant key, usually the F12 key. The debugging tools offer several features.

We can also debug JavaScript code inside a code editor that we use to develop a JavaScript application, such as Visual Studio Code, Atom, Sublime Text, etc.

14. What is the difference between Function declaration and Function expression?

Function declaration	Function expression
Declared as a separate statement within the main JavaScript code	Created inside an expression or some other construct
Can be called before the function is defined	Created when the execution point reaches it; can be used only after that
Offers better code readability and better code organization	Used when there is a need for a conditional declaration of a function

Example:
function abc() {
return 5;
}
	Example:
	var a = function abc() {
	return 5;
	}

15. What are the ways of adding JavaScript code in an HTML file?

There are primarily two ways of embedding JavaScript code:

- We can write JavaScript code within the script tag in the same HTML file; this is suitable when we need just a few lines of scripting within a web page.
- We can import a JavaScript source file into an HTML document; this adds all scripting capabilities to a web page without cluttering the code.

16. What do you understand about cookies?

Fig: Browser cookies

A cookie is generally a small data that is sent from a website and stored on the user's machine by a web browser that was used to access the website. Cookies are used to remember information for later use and also to record the browsing activity on a website.

17. How would you create a cookie?

The simplest way of creating a cookie using JavaScript is as below:

```
document.cookie = "key1 = value1; key2 = value2; expires = date";
```

18. How would you read a cookie?

Reading a cookie using JavaScript is also very simple. We can use the document.cookie string that contains the cookies that we just created using that string.

The document.cookie string keeps a list of name-value pairs separated by semicolons, where 'name' is the name of the cookie, and 'value' is its value. We can also use the split() method to break the cookie value into keys and values.

19. How would you delete a cookie?

To delete a cookie, we can just set an expiration date and time. Specifying the correct path of the cookie that we want to delete is a good practice since some browsers won't allow the deletion of cookies unless there is a clear path that tells which cookie to delete from the user's machine.

```
function delete_cookie(name) {

  document.cookie = name + "=; Path=/; Expires=Thu, 01 Jan 1970 00:00:01 GMT;";

}
```

20. What's the difference between let and var?

Both let and var are used for variable and method declarations in JavaScript. So there isn't much of a difference between these two besides that while var keyword is scoped by function, the let keyword is scoped by a block.

23. What are the different ways an HTML element can be accessed in a JavaScript code?

Here are the ways an HTML element can be accessed in a JavaScript code:

- getElementByClass('classname'): Gets all the HTML elements that have the specified classname.
- getElementById('idname'): Gets an HTML element by its ID name.
- getElementbyTagName('tagname'): Gets all the HTML elements that have the specified tagname.
- querySelector(): Takes CSS style selector and returns the first selected HTML element.

24. What are the ways of defining a variable in JavaScript?

There are three ways of defining a variable in JavaScript:

Var

This is used to declare a variable and the value can be changed at a later time within the JavaScript code.

Const

We can also use this to declare/define a variable but the value, as the name implies, is constant throughout the JavaScript program and cannot be modified at a later time.

Let

This mostly implies that the values can be changed at a later time within the JavaScript code.

25. What are Imports and Exports in JavaScript?

Imports and exports help in writing modular code for our JavaScript applications. With the help of imports and exports, we can split a JavaScript code into multiple files in a project. This greatly simplifies the application source code and encourages code readability.

calc.js

```
export const sqrt = Math.sqrt;

export function square(x) {

 return x * x;

}

export function diag(x, y) {

 return sqrt(square(x) + square(y));

}
```

This file exports two functions that calculate the squares and diagonal of the input respectively.

main.js

```
import { square, diag } from "calc";

console.log(square(4)); // 16

console.log(diag(4, 3)); // 5
```

Therefore, here we import those functions and pass input to those functions to calculate square and diagonal.

26. What is the difference between Document and Window in JavaScript?

Document	Window
The document comes under the windows object and can also be considered as its property.	Window in JavaScript is a global object that holds the structure like variables, functions, location, history, etc.

27. What are some of the JavaScript frameworks and their uses?

JavaScript has a collection of many frameworks that aim towards fulfilling the different aspects of the web application development process. Some of the prominent frameworks are:

- React - Frontend development of a web application
- Angular - Frontend development of a web application
- Node - Backend or server-side development of a web application

28. What is the difference between Undefined and Undeclared in JavaScript?

Undefined	Undeclared
Undefined means a variable has been declared but a value has not yet been assigned to that variable.	Variables that are not declared or that do not exist in a program or application.

29. What is the difference between Undefined and Null in JavaScript?

Undefined	Null
Undefined means a variable has been declared but a value has not yet been assigned to that variable.	Null is an assignment value that we can assign to any variable that is meant to contain no value.

76. Are Java and JavaScript the same?

Java and JavaScript are not identical; they are distinct programming languages with different purposes and characteristics. Java is a high-level, object-oriented programming language designed for building platform-independent applications, often used in enterprise environments for server-side applications, mobile applications, and large systems. It requires compilation and runs on the Java Virtual Machine (JVM). Conversely, JavaScript is a lightweight, interpreted scripting language primarily used to create dynamic and interactive content on websites. It runs directly in web browsers and is an essential technology for web development alongside HTML and CSS. Despite their similar names, their syntax, use cases, and execution environments are quite different.

77. How to detect the OS of the client machine using JavaScript?

The OS on the client machine can be detected with the help of navigator.appVersion string

78. Requirement of debugging in JavaScript

- We can use web browsers such as Google Chrome and Mozilla Firefox to debug the code.
- We can debug in JavaScript using two methods: console.log() and the debugger keyword.

79. What are the pop-up boxes available in JavaScript?

Pop-up boxes available in JavaScript are Alert Box, Confirm Box, and Prompt Box.

71. Difference between Async/Await and Generators

- Async/Await
- Async-await functions are executed sequentially one after another in an easier way.
- Async/Await function might throw an error when the value is returned.
- Generators
- Generator functions are executed with one output at a time by the generator's yield by yield.
- The 'value: X, done: Boolean' is the output result of the Generator function.

72. Primitive data types

The primitive data types are capable of displaying one value at a time. It consists of Boolean, Undefined, Null, Number, and String data types.

73. Role of deferred scripts

The Deferred scripts are used for the HTML parser to finish before executing it.

74. What is Lexical Scoping?

Lexical Scoping in JavaScript can be performed when the internal state of the JavaScript function object consists of the function's code as well as references concerning the current scope chain.

75. What is this [[[]]]?

This '[[[]]]' is a three-dimensional array.

66. What is a WeakMap?

Weakmap is referred to as an object having keys and values, if the object is without reference, it is collected as garbage.

67. What is Object Destructuring? (with examples)

Object destructuring is a method to extract elements from an array or an object.

Example 1: Array Destructuring

```
const arr = [1, 2, 3];

const first = arr[0];

const second = arr[1];

const third = arr[2];
```

Example 2: Object Destructuring

```
const arr = [1, 2, 3];

const [first,second,third,fourth] = arr;

console.log(first); // Outputs 1

console.log(second); // Outputs 2

console.log(third); // Outputs 3
```

68. Prototypal vs Classical Inheritance

- Prototypal Inheritance
- Prototypal inheritance allows any object to be cloned via an object linking method and it serves as a template for those other objects, whether they extend the parent object or not.
- Classical Inheritance
- Classical inheritance is a class that inherits from the other remaining classes.

69. What is a Temporal Dead Zone?

Temporal Dead Zone is a behavior that occurs with variables declared using let and const keywords before they are initialized.

70. JavaScript Design Patterns

When we build JavaScript browser applications, there might be chances to occur errors where JavaScript approaches it in a repetitive manner. This repetitive approach pattern is called JavaScript design patterns. JavaScript design patterns consist of Creational Design Pattern, Structural Design Pattern, and Behavioral Design patterns.

63. What are generator functions?

Generator functions are declared with a special class of functions and keywords using function*. It does not execute the code, however, it returns a generator object and handles the execution.

64. What is WeakSet?

WeakSet is a collection of unique and ordered elements that contain only objects which are referenced weakly.

65. What is the use of callbacks?

- A callback function is used to send input into another function and is performed inside another function.
- It also ensures that a particular code does not run until another code has completed its execution.

55. Which method is used to retrieve a character from a certain index?

We can retrieve a character from a certain index with the help of charAt() function method.

56. What is BOM?

BOM is the Browser Object Model where users can interact with browsers that is a window, an initial object of the browser. The window object consists of a document, history, screen, navigator, location, and other attributes. Nevertheless, the window's function can be called directly as well as by referencing the window.

57. Difference between client-side and server-side

- Client-side JavaScript
- Client-side JavaScript is made up of fundamental language and predefined objects that perform JavaScript in a browser.
- Also, it is automatically included in the HTML pages where the browser understands the script.
- Server-side Javascript
- Server-side JavaScript is quite similar to Client-side javascript.
- Server-side JavaScript can be executed on a server.
- The server-side JavaScript is deployed once the server processing is done.

49. Advantages of using External JavaScript

- External Javascript allows web designers and developers to collaborate on HTML and javascript files.
- It also enables you to reuse the code.
- External javascript makes Code readability simple.

50. What are object prototypes?

Following are the different object prototypes in javascript that are used to inherit particular properties and methods from the Object.prototype.

1. Date objects are used to inherit properties from the Date prototype
2. Math objects are used to inherit properties from the Math prototype
3. Array objects are used to inherit properties from the Array prototype.

51. Types of errors in javascript

Javascript has two types of errors, Syntax error, and Logical error.

52. What is memoization?

In JavaScript, when we want to cache the return value of a function concerning its parameters, it is called memoization. It is used to speed up the application especially in case of complex, time consuming functions.

53. Recursion in a programming language

Recursion is a technique in a programming language that is used to iterate over an operation whereas a function calls itself repeatedly until we get the result.

54. Use of a constructor function (with examples)

Constructor functions are used to create single objects or multiple objects with similar properties and methods.

Example:

```
function Person(name,age,gender)
{
 this.name = name;
 this.age = age;
 this.gender = gender;
}
var person1 = new Person("Vivek", 76, "male");
console.log(person1);
var person2 = new Person("Courtney", 34, "female");
console.log(person2);
```

47. difference between exec () and test () methods

- exec()
- It is an expression method in JavaScript that is used to search a string with a specific pattern.
- Once it has been found, the pattern will be returned directly, otherwise, it returns an “empty” result.
- test ()
- It is an expression method in JavaScript that is also used to search a string with a specific pattern or text.
- Once it has been found, the pattern will return the Boolean value 'true', else it returns ‘false’.

48. currying in JavaScript (with examples)

In JavaScript, when a function of an argument is transformed into functions of one or more arguments is called Currying.

Example:

```
function add (a) {

 return function(b){

   return a + b;

 }

}

add(3)(4)
```

44. Characteristics of javascript strict-mode

- Strict mode does not allow duplicate arguments and global variables.
- One cannot use JavaScript keywords as a parameter or function name in strict mode.
- All browsers support strict mode.
- Strict mode can be defined at the start of the script with the help of the keyword ‘use strict’.

45. Higher Order Functions (with examples)

Higher-order functions are the functions that take functions as arguments and return them by operating on other functions

Example:

```
function higherOrder(fn)

 {

  fn();

}

higherOrder(function() { console.log("Hello world") });
```