Question 1: Basic Types and Interfaces

Create an interface called `User` with properties for id (number), name (string), email (string), and isActive (boolean). Then, create a function called `createUser` that takes a user object of type `User` and returns it. Finally, write code to create a new user and call the function.

Question 2: Union Types and Type Guards

Create a type called `Input` that can be either a number or a string. Then write a function called `processInput` that takes an argument of type `Input` and returns a string. If the input is a number, convert it to a string and prepend "Number: " to it. If the input is already a string, prepend "String: " to it. Use type guards to check the type of input.

Question 3: Classes and Inheritance

Create a base class called `Vehicle` with properties for make (string), model (string), and year (number). Include a method called `getInfo()` that returns a string with the vehicle information. Then create two subclasses: `Car` and `Motorcycle`. The `Car` class should have an additional property for doors (number), and the `Motorcycle` class should have a property for hasSidecar (boolean). Override the `getInfo()` method in each subclass to include the additional information.

Question 4: Access Modifiers and Getters/Setters

Create a class called `BankAccount` with a private property for balance (number), a private readonly property for accountNumber (string), a constructor that initializes both properties, a getter method for balance, a getter method for accountNumber, a method called `deposit(amount: number)` that adds to the balance, and a method called `withdraw(amount: number)` that subtracts from the balance but prevents overdrafts by throwing an error if the amount is greater than the balance. Test the class by creating an account, making deposits and withdrawals, and trying to access the private properties directly.

Question 5: Abstract Classes

Create an abstract class called `Shape` with a protected property for color (string), a constructor that sets the color, an abstract method called `calculateArea()` that returns a number, and a concrete method called `getColor()` that returns the color. Then create two concrete classes that extend `Shape`: `Circle` with a property for radius (number), and `Rectangle` with properties for width (number) and height (number). Implement the `calculateArea()` method in each subclass. Then create instances of both shapes, calculate their areas, and get their colors.

Question 6: Types and Interfaces - Creating and Using Interfaces

Create an interface called `Product` with the following properties: `id` (number), `name` (string), `price` (number), and `category` (string). Then, create a function `createProduct` that accepts an object of type `Product` and returns it. Finally, create a new product object and pass it to `createProduct()`.

Question 7: Class Inheritance - Extending a Base Class

Create a base class `Employee` with `name` (string), `salary` (number), and a method `getDetails()` that returns the employee's name and salary. Then, create two subclasses: `Developer` with an additional property `programmingLanguage` (string) and an overridden `getDetails()` method, and `Designer` with an additional property `toolUsed` (string) and an overridden `getDetails()` method. Create instances of both classes and call `getDetails()` on each.

Question 8: Access Modifiers - Using Private, Protected, and Readonly

Create a class `Student` with a `public` property `name` (string), a `private` property `grades` (array of numbers), a `protected` property `school` (string), and a `readonly` property `studentID` (number) that is initialized in the constructor. Implement methods to add a grade to the `grades` array and get the average grade. Try accessing the properties from inside and outside the class to test access modifiers.

Question 9: Union Types and Type Guards

Create a type `Response` that can be either `{ success: true, data: string }` or `{ success: false, error: string }`. Then, write a function `handleResponse` that takes an argument of type `Response` and logs `"Data received: {data}"` if `success` is `true`, and logs `"Error occurred: {error}"` if `success` is `false`. Use type guards to differentiate between the two cases.

Question 10: Abstract Classes - Creating and Extending

Create an abstract class `Animal` with a `protected` property `species` (string), a constructor that sets the species, an abstract method `makeSound()` that returns a string, and a concrete method `getSpecies()` that returns the species. Then, create two classes: `Dog` with a `makeSound()` method that returns `"Woof!"`, and `Cat` with a `makeSound()` method that returns `"Meow!"`. Create instances of both classes, call `makeSound()`, and `getSpecies()`.

Question 11: Generics - Creating a Generic Function with Index Search

Create a generic function `findIndex<T>(arr: T[], value: T): number` that takes an array of any type and a value to search for. The function should return the index of the value if found; otherwise, return `-1`. Call this function with different types of arrays (e.g., an array of numbers, an array of strings) and log the results..

Question 12: Type Guards - Checking Object Types

Create two interfaces: `Car` with a property `drive()` that returns `"Driving a car!"`, and `Bike` with a property `ride()` that returns `"Riding a bike!"`. Then, create a function `useVehicle(vehicle: Car | Bike)` that calls `drive()` if it's a `Car` and calls `ride()` if it's a `Bike`, using a type guard to differentiate between the two. Create instances of `Car` and `Bike`, and pass them to `useVehicle()`.

Question 13: Interface Intersection - Merging Multiple Interfaces

Create two interfaces: `Person` with properties `name` (string) and `age` (number), and `Employee` with property `jobTitle` (string). Then, create a new type `FullTimeEmployee` that combines both interfaces using intersection (`&`). Write a function `describeEmployee(emp: FullTimeEmployee)` that logs the `name`, `age`, and `jobTitle`. Create a `FullTimeEmployee` object and pass it to the function.

Question 14: Interface Union - Handling Multiple Object Types

Create two interfaces: `Dog` with a method `bark()` returning `"Woof!"`, and `Cat` with a method `meow()` returning `"Meow!"`. Then, create a type `Pet` that can be either a `Dog` or a `Cat`. Write a function `makeSound(pet: Pet)` that calls `bark()` if it's a `Dog` and calls `meow()` if it's a `Cat`, using a type guard to determine the correct method. Create instances of `Dog` and `Cat`, and call `makeSound()` on both.

Question 15: Implementing an Interface in a Class

Create an interface `Shape` with a method `calculateArea(): number` and a method `getType(): string`. Then, create a class `Circle` that implements `Shape` with a property `radius` (number), implements `calculateArea()` to return the area of the circle, and implements `getType()` to return `"Circle"`. Create an instance of `Circle`, set the radius, and call both methods.