

# 1000 TypeScript MCQs:

### Topic: helloworld

1. **What is the correct way to define a `hello world` function in TypeScript?**

- A. `function helloWorld(): void { console.log('Hello, World!'); }`
- B. `void helloWorld() { console.log('Hello, World!'); }`
- C. `def helloWorld(): void { console.log('Hello, World!'); }`
- D. `helloWorld() { console.log('Hello, World!'); }`

2. **How do you compile a TypeScript file named `hello.ts`?**

- A. `tsc hello.ts`
- B. `tsc compile hello.ts`
- C. `typescript hello.ts`
- D. `compile hello.ts`

### Topic: json\_objects

3. **Which TypeScript type can be used to represent JSON objects?**

- A. `object`
- B. `JSON`
- C. `any`
- D. `Record<string, any>`

4. **How can you parse a JSON string in TypeScript?**

- A. ``JSON.parse(jsonString)``
- B. ``JSON.parseString(jsonString)``
- C. ``JSON.toObject(jsonString)``
- D. ``JSON.toJSON(jsonString)``

### Topic: syntax\_error

5. **\*\*What will cause a syntax error in TypeScript?\*\***

- A. ``let num: number = 'string';``
- B. ``function hello { console.log('Hello'); }``
- C. ``const PI: number = 3.14;``
- D. ``import * as fs from 'fs';``

6. **\*\*Which of the following code will generate a syntax error in TypeScript?\*\***

- A. ``let x = 10``
- B. ``const hello = (): string => 'Hello';``
- C. ``let a: number, let b: number;``
- D. ``interface Person { name: string; age: number; }``

### Topic: type\_error

7. **\*\*What is the result of attempting to assign a string to a number variable in TypeScript?\*\***

- A. Compile-time error
- B. Run-time error
- C. Silent type coercion
- D. No error

8. **\*\*Which code snippet will result in a type error?\*\***

- A. ``let isDone: boolean = false;``
- B. ``let num: number = 'string';``
- C. ``let name: string = 'Alice';``
- D. ``let list: number[] = [1, 2, 3];``

### Topic: assignability\_error

9. **\*\*What causes an assignability error in TypeScript?\*\***

- A. Assigning a value of type ``string`` to a variable of type ``number``
- B. Using ``let`` instead of ``const``
- C. Importing a module incorrectly
- D. Defining a function without a return type

10. **\*\*Which of the following assignments will cause an assignability error?\*\***

- A. ``let count: number = 42;``
- B. ``let name: string = 'John';``
- C. ``let isValid: boolean = 1;``
- D. ``let arr: number[] = [1, 2, 3];``

### Topic: strong\_typing

11. **\*\*Which of the following demonstrates strong typing in TypeScript?\*\***

- A. ``let value: any = 5;``
- B. ``let value = 5;``

- C. `let value: number = 5;`
- D. `let value; value = 5;`

12. **What is an advantage of strong typing in TypeScript?**

- A. More flexible code
- B. Reduced code readability
- C. Improved code quality and maintainability
- D. Increased run-time errors

### Topic: const\_let

13. **What is the difference between `const` and `let` in TypeScript?**

- A. `const` variables can be reassigned; `let` variables cannot
- B. `const` is block-scoped; `let` is function-scoped
- C. `const` variables cannot be reassigned; `let` variables can
- D. `const` variables are globally scoped; `let` variables are not

14. **Which of the following code snippets is valid TypeScript?**

- A. `const x = 10; x = 20;`
- B. `let x = 10; let x = 20;`
- C. `let x = 10; x = 20;`
- D. `const x = 10; let x = 20;`

### Topic: modules

15. **How do you export a function from a module in TypeScript?**

- A. ``exports function myFunction() { }``
- B. ``export function myFunction() { }``
- C. ``module.exports = function myFunction() { }``
- D. ``export { function myFunction() { } }``

16. **\*\*How do you import a function from a module in TypeScript?\*\***

- A. ``import { myFunction } from './myModule';``
- B. ``require { myFunction } from './myModule';``
- C. ``import myFunction from './myModule';``
- D. ``include { myFunction } from './myModule';``

### Topic: native\_ECMAScript\_modules

17. **\*\*Which of the following is a native ECMAScript module syntax for exporting?\*\***

- A. ``exports.myFunction = function() { };``
- B. ``module.exports = function() { };``
- C. ``export function myFunction() { };``
- D. ``import function myFunction() { };``

18. **\*\*What is the correct way to import a default export from a native ECMAScript module?\*\***

- A. ``import { default } from 'module';``
- B. ``import default from 'module';``
- C. ``import * as default from 'module';``
- D. ``import myFunction from 'module';``

### Topic: import\_inquirer\_ECMAScript\_module

19. **\*\*Which of the following correctly imports the `inquirer` module in TypeScript using ECMAScript module syntax?\*\***

- A. ``import * as inquirer from 'inquirer';``
- B. ``const inquirer = require('inquirer');``
- C. ``import inquirer = require('inquirer');``
- D. ``include inquirer from 'inquirer';``

20. **\*\*How do you use the `inquirer` module to ask a question in TypeScript?\*\***

- A. ``inquirer.askQuestion('What is your name?');``
- B. ``inquirer.prompt([ { type: 'input', name: 'name', message: 'What is your name?' } ]);``
- C. ``inquirer.ask([ { type: 'input', name: 'name', message: 'What is your name?' } ]);``
- D. ``inquirer.question([ { type: 'input', name: 'name', message: 'What is your name?' } ]);``

### Topic: chalk

21. **\*\*Which of the following correctly imports and uses the `chalk` module in TypeScript?\*\***

- A. ``import chalk from 'chalk'; console.log(chalk.green('Hello'));``
- B. ``const chalk = require('chalk'); console.log(chalk.green('Hello'));``
- C. ``import { chalk } from 'chalk'; console.log(chalk.green('Hello'));``
- D. ``include chalk from 'chalk'; console.log(chalk.green('Hello'));``

22. **\*\*What does `chalk` module primarily do?\*\***

- A. Provides utilities for working with JSON
- B. Enhances console output with colors and styles
- C. Manages file I/O operations

- D. Facilitates HTTP requests

### ### Topic: unions\_literals

23. **\*\*Which of the following correctly defines a union type in TypeScript?\*\***

- A. `let id: string | number;`
- B. `let id: union string, number;`
- C. `let id: (string, number);`
- D. `let id: [string, number];`

24. **\*\*What is the purpose of literal types in TypeScript?\*\***

- A. To define a variable's exact value
- B. To define multiple data types for a variable
- C. To create arrays of different types
- D. To enable dynamic typing

### ### Topic: objects

25. **\*\*Which of the following defines an object type in TypeScript?\*\***

- A. `let person: { name: string, age: number };`
- B. `let person: Object = { name: string, age: number };`
- C. `let person = new Object(name: string, age: number);`
- D. `let person`

`= Object.create({ name: string, age: number });`

26. **\*\*How do you access a property of an object in TypeScript?\*\***

- A. ``object.property``
- B. ``object->property``
- C. ``object[property]``
- D. ``object:property``

### Topic: object\_aliasing

27. **\*\*How do you create an alias for an object type in TypeScript?\*\***

- A. ``alias Person = { name: string, age: number };``
- B. ``type Person = { name: string, age: number };``
- C. ``let Person = { name: string, age: number };``
- D. ``typedef Person { name: string, age: number };``

28. **\*\*What is the purpose of using type aliases for objects in TypeScript?\*\***

- A. To simplify complex type definitions
- B. To create mutable objects
- C. To enforce stricter type checking
- D. To enable dynamic typing

### Topic: structural\_typing\_object\_literals

29. **\*\*Which statement is true about structural typing in TypeScript?\*\***

- A. Type compatibility is determined by structure, not name
- B. Type compatibility requires explicit type names
- C. Structural typing enforces strict equality of types



- D. Structural typing is only applicable to primitive types

30. **\*\*How does structural typing work with object literals in TypeScript?\*\***

- A. By comparing the names of the objects
- B. By comparing the properties and their types
- C. By comparing the memory addresses
- D. By comparing the methods of the objects

### Topic: nested\_objects

31. **\*\*How do you define a nested object in TypeScript?\*\***

- A. ``let car: { model: string, engine: { type: string, horsepower: number } };``
- B. ``let car: { model: string; engine: { type: string; horsepower: number; } };``
- C. ``let car = { model: string, engine: { type: string, horsepower: number } };``
- D. ``let car = { model: string; engine: { type: string; horsepower: number; } };``

32. **\*\*How do you access a nested property in a TypeScript object?\*\***

- A. ``object.nested.property``
- B. ``object->nested->property``
- C. ``object[nested][property]``
- D. ``object:nested:property``

### Topic: intersection\_types

33. **\*\*Which of the following correctly defines an intersection type in TypeScript?\*\***

- A. ``type Person = { name: string } & { age: number };``

- B. ``type Person = { name: string | age: number };``
- C. ``type Person = { name: string && age: number };``
- D. ``type Person = (name: string, age: number);``

34. **What is an advantage of using intersection types in TypeScript?**

- A. They allow for type unions
- B. They combine multiple types into one
- C. They make types optional
- D. They enforce stricter type checking

### Topic: `any__unknown_never_types`

35. **What is the difference between ``any``, ``unknown``, and ``never`` types in TypeScript?**

- A. ``any`` allows any type, ``unknown`` requires type checking, ``never`` represents unreachable code
- B. ``any`` allows any type, ``unknown`` restricts types, ``never`` allows null values
- C. ``any`` allows primitive types, ``unknown`` allows object types, ``never`` allows no types
- D. ``any`` allows string and number, ``unknown`` allows boolean, ``never`` allows undefined

36. **Which of the following is true about the ``never`` type?**

- A. It can be assigned to any other type
- B. It represents values that never occur
- C. It is a subtype of all other types
- D. It can hold any value

### Topic: `explicit_casting`

37. **\*\*How do you explicitly cast a type in TypeScript?\*\***

- A. ``let value: number = <number>someValue;``
- B. ``let value: number = (number)someValue;``
- C. ``let value: number = {number}someValue;``
- D. ``let value: number = (someValue as number);``

38. **\*\*Which of the following demonstrates explicit casting to a string type?\*\***

- A. ``let str: string = someValue as string;``
- B. ``let str: string = <string>someValue;``
- C. ``let str: string = String(someValue);``
- D. All of the above

### Topic: enum

39. **\*\*How do you define an enum in TypeScript?\*\***

- A. ``enum Color { Red, Green, Blue }``
- B. ``enum Color = { Red, Green, Blue }``
- C. ``const enum Color { Red, Green, Blue }``
- D. ``type Color { Red, Green, Blue }``

40. **\*\*How do you access an enum value in TypeScript?\*\***

- A. ``Color.Red``
- B. ``Color[Red]``
- C. ``Color::Red``
- D. ``Color->Red``

### ### Topic: const\_enum

41. **\*\*What is a `const enum` in TypeScript?\*\***

- A. An enum that can be reassigned
- B. An enum that is inlined at compile time
- C. An enum with constant values
- D. An enum that cannot be used in switch statements

42. **\*\*Which of the following is true about `const enum`?**

- A. They can be used with `let` and `var`
- B. They are removed during compilation
- C. They can hold only string values
- D. They are mutable

### ### Topic: arrays

43. **\*\*How do you define an array of numbers in TypeScript?**

- A. `let arr: number[];`
- B. `let arr: Array<number>;`
- C. `let arr: [number];`
- D. Both A and B

44. **\*\*Which method can be used to add an element to an array in TypeScript?**

- A. `arr.add(element)`
- B. `arr.push(element)`
- C. `arr.append(element)`

- D. `arr.insert(element)`

### ### Topic: functions

45. **\*\*How do you define a function with a return type in TypeScript?\*\***

- A. `function add(a: number, b: number): number { return a + b; }`
- B. `function add(a, b): number { return a + b; }`
- C. `function add(a: number, b: number) { return a + b; }`
- D. `function add(a, b) { return a + b; }`

46. **\*\*What is the correct way to define a function type in TypeScript?\*\***

- A. `let add: (a: number, b: number) => number;`
- B. `let add: function(a: number, b: number): number;`
- C. `let add: (a, b) => number;`
- D. `let add: (a: number, b: number): number;`

### ### Topic: function\_optional\_parameter

47. **\*\*How do you define a function with an optional parameter in TypeScript?\*\***

- A. `function greet(name?: string) { }`
- B. `function greet(name: string?) { }`
- C. `function greet(name: string = undefined) { }`
- D. `function greet(?name: string) { }`

48. **\*\*What happens if an optional parameter is not provided in TypeScript?\*\***

- A. It throws a compile-time error

- B. It throws a run-time error
- C. It is assigned `undefined`
- D. It is assigned `null`

### Topic: function\_default\_parameter

49. **\*\*How do you define a function with a default parameter in TypeScript?\*\***

- A. ``function greet(name: string = 'Guest') { }``
- B. ``function greet(name = 'Guest') { }``
- C. ``function greet(name?: string = 'Guest') { }``
- D. ``function greet(name: 'Guest') { }``

50. **\*\*What is the default value of a parameter if not provided in TypeScript?\*\***

- A. `null`
- B. `undefined`
- C. The specified default value
- D. An empty string

### Topic: function\_rest\_parameter

51. **\*\*How do you define a function with a rest parameter in TypeScript?\*\***

- A. ``function sum(...numbers: number[]): number { }``
- B. ``function sum(numbers...: number[]): number { }``
- C. ``function sum(numbers: ...number[]): number { }``
- D. ``function sum(...numbers: [number]): number { }``

52. **\*\*What is the type of a rest parameter in TypeScript?\*\***

- A. ``Array``
- B. ``number[]``
- C. ``Rest``
- D. ``any[]``

### Topic: async

53. **\*\*Which keyword is used to define an asynchronous function in TypeScript?\*\***

- A. ``async``
- B. ``await``
- C. ``promise``
- D. ``defer``

54. **\*\*How do you handle asynchronous operations in TypeScript?\*\***

- A. Using callbacks
- B. Using promises
- C. Using async/await
- D. All of the above

### Topic: function\_overloads

55. **\*\*How do you define function overloads in TypeScript?\*\***

- A. By defining multiple functions with the same name
- B. By using the ``overload`` keyword
- C. By defining multiple signatures for a single function

- D. By using `any` type for parameters

56. **\*\*Which of the following correctly demonstrates function overloads in TypeScript?\*\***

- A. `function add(a: number, b: number): number; function add(a: string, b: string): string; function add(a: any, b: any) { return a + b; }`
- B. `function add(a: number, b: number) { return a + b; } function add(a: string, b: string) { return a + b; }`
- C. `overload function add(a: number, b: number): number; overload function add(a: string, b: string): string;`
- D. `function add(a: number, b: number): number; function add(a: string, b: string): string;`

### Topic: tuples

57. **\*\*How do you define a tuple type in TypeScript?\*\***

- A. `let tuple: [string, number];`
- B. `let tuple: (string, number);`
- C. `let tuple: {string, number};`
- D. `let tuple: <string, number>;`

58. **\*\*How do you access the elements of a tuple in TypeScript?\*\***

- A. `tuple[0], tuple[1]`
- B. `tuple.item(0), tuple.item(1)`
- C. `tuple.0, tuple.1`
- D. `tuple.first, tuple.second`

### Topic: functions



59. **\*\*What is the difference between `void` and `never` return types in TypeScript?\*\***

- A. `void` represents functions that return no value, `never` represents functions that never return
- B. `void` represents functions that return null, `never` represents functions that return undefined
- C. `void` is used for functions, `never` is used for variables
- D. `void` is a subtype of `never`

60. **\*\*How do you define a function that never returns in TypeScript?\*\***

- A. `function fail(): never { throw new Error('Something failed'); }`
- B. `function fail(): void { throw new Error('Something failed'); }`
- C. `function fail(): any { throw new Error('Something failed'); }`
- D. `function fail(): undefined { throw new Error('Something failed'); }`

### Topic: async

61. **\*\*How do you wait for an asynchronous operation to complete in TypeScript?\*\***

- A. Using `await`
- B. Using `wait`
- C. Using `pause`
- D. Using `hold`

62. **\*\*Which of the following is true about async functions in TypeScript?\*\***

- A. They always return a promise
- B. They cannot contain synchronous code
- C. They must be named with the `async` prefix
- D. They cannot be used with the `await` keyword

### ### Topic: object\_aliasing

63. **\*\*What is an alias type in TypeScript and how is it defined?\*\***

- A. An alias type is another name for an existing type and is defined using the `type` keyword
- B. An alias type is a copy of an existing type and is defined using the `alias` keyword
- C. An alias type is a new type created from a class and is defined using the `class` keyword
- D. An alias type is an interface implementation and is defined using the `interface` keyword

64. **\*\*Which of the following correctly defines an alias for a complex object type?\*\***

- A. `type Point = { x: number; y: number; };`
- B. `alias Point = { x: number; y: number; };`
- C. `interface Point { x: number; y: number; };`
- D. `class Point { x: number; y: number; };`

### ### Topic: nested\_objects

65. **\*\*How do you access a deeply nested property in a TypeScript object?\*\***

- A. `object.level1.level2.property`
- B. `object[level1][level2][property]`
- C. `object->level1->level2->property`
- D. `object:level1:level2:property`

66. **\*\*What is the best way to safely access nested properties in TypeScript?\*\***

- A. Using optional chaining `?.`
- B. Using a try-catch block
- C. Using strict null checks

- D. Using the ``in`` operator

### ### Topic: intersection\_types

67. **\*\*Which of the following best describes intersection types in TypeScript?\*\***

- A. They combine multiple types into one, requiring all type properties to be present
- B. They create a union of multiple types, allowing any of the type properties to be present
- C. They restrict a type to a subset of its properties
- D. They enable implicit type conversion

68. **\*\*How do you define an intersection type in TypeScript?\*\***

- A. ``type Combined = TypeA & TypeB;``
- B. ``type Combined = TypeA | TypeB;``
- C. ``type Combined = TypeA - TypeB;``
- D. ``type Combined = TypeA + TypeB;``

### ### Topic: enum

69. **\*\*How do you define an enum with string values in TypeScript?\*\***

- A. ``enum Color { Red = "Red", Green = "Green", Blue = "Blue" }``
- B. ``enum Color = { Red = "Red", Green = "Green", Blue = "Blue" }``
- C. ``const enum Color { Red = "Red", Green = "Green", Blue = "Blue" }``
- D. ``type Color = { Red: "Red", Green: "Green", Blue: "Blue" }``

70. **\*\*What is the default underlying type of an enum in TypeScript if not specified?\*\***

- A. ``number``

- B. ``string``
- C. ``boolean``
- D. ``object``

### Topic: `const_enum`

71. **What is the advantage of using ``const enum`` in TypeScript?**

- A. Improved performance due to inlining of values
- B. Ability to modify enum values at runtime
- C. Easier to debug
- D. Greater type safety

72. **Which of the following code snippets correctly defines a ``const enum``?**

- A. ``const enum Direction { Up, Down, Left, Right }``
- B. ``const enum Direction = { Up, Down, Left, Right }``
- C. ``enum const Direction { Up, Down, Left, Right }``
- D. ``enum Direction { Up, Down, Left, Right }``

### Topic: `arrays`

73. **How do you define an array of tuples in TypeScript?**

- A. ``let arr: [number, string][];``
- B. ``let arr: Array<[number, string]>;``
- C. Both A and B
- D. ``let arr: [(number, string)];``

74. **\*\*Which method removes the last element from an array in TypeScript?\*\***

- A. `arr.pop()`
- B. `arr.shift()`
- C. `arr.splice()`
- D. `arr.remove()`

### Topic: function\_optional\_parameter

75. **\*\*How do you provide a default value for an optional parameter in TypeScript?\*\***

- A. `function greet(name: string = 'Guest') { }`
- B. `function greet(name?: string = 'Guest') { }`
- C. `function greet(name: string?) { }`
- D. `function greet(name = 'Guest'? ) { }`

76. **\*\*What is the syntax to call a function with an optional parameter without providing the parameter in TypeScript?\*\***

- A

`.greet()`

- B. `greet(undefined)`
- C. Both A and B
- D. `greet(null)`

### Topic: function\_default\_parameter

77. **\*\*Which statement about default parameters in TypeScript is true?\*\***

- A. Default parameters can only be used at the end of the parameter list
- B. Default parameters must be the first parameters in the list
- C. Default parameters can be used anywhere in the parameter list
- D. Default parameters cannot be used in TypeScript

78. **\*\*How are default parameters different from optional parameters in TypeScript?\*\***

- A. Default parameters provide a value if not supplied, optional parameters can be `undefined`
- B. Default parameters can be `undefined`, optional parameters provide a value if not supplied
- C. There is no difference
- D. Optional parameters must always be supplied

### Topic: json\_objects

79. **\*\*How do you parse a JSON string into an object in TypeScript?\*\***

- A. `JSON.parse(jsonString)`
- B. `JSON.toObject(jsonString)`
- C. `JSON.decode(jsonString)`
- D. `JSON.stringify(jsonString)`

80. **\*\*How do you convert an object into a JSON string in TypeScript?\*\***

- A. `JSON.stringify(object)`
- B. `JSON.toJSON(object)`
- C. `JSON.encode(object)`
- D. `JSON.parse(object)`

Certainly! I'll continue from where we left off:

### Topic: helloworld

81. **\*\*What is the correct TypeScript syntax to print "Hello, World!" to the console?\*\***

- A. ``console.log("Hello, World!");``
- B. ``print("Hello, World!");``
- C. ``echo("Hello, World!");``
- D. ``write("Hello, World!");``

82. **\*\*What is the TypeScript equivalent of a traditional "Hello, World!" script in JavaScript?\*\***

- A. There is no difference
- B. ``console.log("Hello, World!");``
- C. ``console.write("Hello, World!");``
- D. ``print("Hello, World!");``

### Topic: json\_objects

83. **\*\*How do you handle a TypeScript error when parsing an invalid JSON string?\*\***

- A. Using a try-catch block
- B. Using ``JSON.safeParse``
- C. Using ``JSON.tryParse``
- D. It automatically handles the error

84. **\*\*Which of the following is a valid JSON object?\*\***

- A. ``{"name": "John", "age": 30}``
- B. ``{"name": "John", age: 30}``

- C. `{name: "John", "age": 30}`
- D. `{name: "John", age: 30}`

### Topic: syntax\_error

85. **\*\*What will cause a syntax error in TypeScript?\*\***

- A. Missing a semicolon at the end of a statement
- B. Incorrectly nested curly braces
- C. Incorrect data type assignment
- D. Using a variable before it is declared

86. **\*\*Which of the following will cause a syntax error in TypeScript?\*\***

- A. `let a: number = 5`
- B. `let a: number = '5'`
- C. `let a = 5;`
- D. `let a: number = 5;`

### Topic: type\_error

87. **\*\*What is a type error in TypeScript?\*\***

- A. Assigning a value of an incorrect type to a variable
- B. Using an undefined variable
- C. Syntax errors in the code
- D. None of the above

88. **\*\*Which of the following will cause a type error in TypeScript?\*\***



- A. `let a: number = 'hello';`
- B. `let a: string = 'hello';`
- C. `let a = 5;`
- D. `let a: any = 'hello';`

### Topic: assignability\_error

89. **What will cause an assignability error in TypeScript?**

- A. Assigning a string to a variable typed as number
- B. Declaring a variable without a type
- C. Using an undeclared variable
- D. Writing a function without a return type

90. **Which of the following code snippets will cause an assignability error?**

- A. `let a: number = 5;`
- B. `let a: string = 'hello';`
- C. `let a: number = 'hello';`
- D. `let a: any = 5;`

### Topic: strong\_typing

91. **What is meant by "strong typing" in TypeScript?**

- A. Variables are bound to specific data types
- B. All variables must be declared
- C. Variables can change type dynamically
- D. Functions must have a return type

92. **\*\*Which of the following demonstrates strong typing in TypeScript?\*\***

- A. ``let a: number = 5;``
- B. ``let a = 5;``
- C. ``let a: any = 'hello';``
- D. ``let a = 'hello';``

### Topic: const\_let

93. **\*\*What is the difference between ``const`` and ``let`` in TypeScript?\*\***

- A. ``const`` declares a constant variable, ``let`` declares a block-scoped variable
- B. ``let`` declares a constant variable, ``const`` declares a block-scoped variable
- C. Both are function-scoped variables
- D. ``const`` allows reassignment, ``let`` does not

94. **\*\*Which of the following is a correct use of ``const`` in TypeScript?\*\***

- A. ``const a = 5; a = 10;``
- B. ``const a = 5;``
- C. ``const a; a = 5;``
- D. ``const a = 'hello'; a = 'world';``

### Topic: modules

95. **\*\*How do you export a function from a TypeScript module?\*\***

- A. ``export function myFunction() { }``
- B. ``function export myFunction() { }``

- C. `function myFunction() export { }`
- D. `myFunction export function() { }`

96. **\*\*How do you import a function from a TypeScript module?\*\***

- A. `import { myFunction } from './module';`
- B. `import myFunction from './module';`
- C. `require { myFunction } from './module';`
- D. `require myFunction from './module';`

### Topic: native\_ECMAScript\_modules

97. **\*\*What syntax is used for native ECMAScript modules in TypeScript?\*\***

- A. `import { myFunction } from './module';`
- B. `import myFunction from './module';`
- C. `require { myFunction } from './module';`
- D. `require myFunction from './module';`

98. **\*\*How do you export a variable from a native ECMAScript module in TypeScript?\*\***

- A. `export const myVariable = 5;`
- B. `const myVariable = 5 export;`
- C. `const export myVariable = 5;`
- D. `export variable myVariable = 5;`

### Topic: import\_inquirer\_ECMAScript\_module

99. **\*\*What is the correct way to import the `inquirer` module in a TypeScript file using ECMAScript module syntax?\*\***

- A. ``import inquirer from 'inquirer';``
- B. ``import { inquirer } from 'inquirer';``
- C. ``require inquirer from 'inquirer';``
- D. ``require { inquirer } from 'inquirer';``

100. **\*\*How do you use the `inquirer` module to prompt the user in TypeScript?\*\***

- A.

```
```typescript
```

```
import inquirer from 'inquirer';
```

```
inquirer.prompt([ { type: 'input', name: 'name', message: 'What is your name?' } ])
```

```
.then(answers => console.log(answers.name));
```

```
```
```

- B.

```
```typescript
```

```
import { inquirer } from 'inquirer';
```

```
inquirer.prompt([ { type: 'input', name: 'name', message: 'What is your name?' } ])
```

```
.then(answers => console.log(answers.name));
```

```
```
```

- C.

```
```typescript
```

```
const inquirer = require('inquirer');
```

```
inquirer.prompt([ { type: 'input', name: 'name', message: 'What is your name?' } ])
```

```
.then(answers => console.log(answers.name));
```

```
```
```

- D.

```
``typescript
```

```
import * as inquirer from 'inquirer';
```

```
inquirer.prompt([ { type: 'input', name: 'name', message: 'What is your name?' } ])
```

```
.then(answers => console.log(answers.name));
```

```
...
```

### Topic: chalk

101. \*\*How do you import the `chalk` module in TypeScript using ECMAScript module syntax?\*\*

- A. `import chalk from 'chalk';`
- B. `import { chalk } from 'chalk';`
- C. `require chalk from 'chalk';`
- D. `require { chalk } from 'chalk';`

102. \*\*How do you use the `chalk` module to print colored text in TypeScript?\*\*

- A. `console.log(chalk.red('Hello, World!'));`
- B. `console.log(chalk.color('red').text('Hello, World!'));`
- C. `console.log(chalk.text('Hello, World!').red);`
- D. `console.log(chalk.color('Hello, World!').red);`

### Topic: unions\_literals

103. \*\*What is a union type in TypeScript?\*\*

- A. A type that can be one of several types
- B. A type that combines multiple types into one

- C. A type that can only be a string or number
- D. A type that can be any type

104. **\*\*How do you define a union type in TypeScript?\*\***

- A. ``let value: string | number;``
- B. ``let value: string & number;``
- C. ``let value: (string, number);``
- D. ``let value: { string, number };``

### Topic: objects

105. **\*\*How do you define an object type in TypeScript?\*\***

- A. ``let obj: { name`  
: string, age: number };``
- B. ``let obj = { name: string, age: number };``
- C. ``let obj: { name: string; age: number; } = {};``
- D. ``let obj = { name: "string", age: "number" };``

106. **\*\*Which of the following is a valid object in TypeScript?\*\***

- A. ``{ name: 'John', age: 30 }``
- B. ``{ 'name': 'John', 'age': 30 }``
- C. ``{ name: 'John', 'age': '30' }``
- D. ``{ name: "John", age: "thirty" }``

### Topic: object\_alias

107. **\*\*What is an alias for an object type in TypeScript?\*\***

- A. A way to give a type a new name
- B. A way to create a new object
- C. A way to define a class
- D. A way to export an object

108. **\*\*How do you create an alias for an object type in TypeScript?\*\***

- A. ``type Person = { name: string, age: number };``
- B. ``interface Person { name: string; age: number; }``
- C. ``alias Person = { name: string; age: number };``
- D. ``let Person = { name: string, age: number };``

### Topic: structural\_typing\_object\_literals

109. **\*\*What is structural typing in TypeScript?\*\***

- A. A type system where the compatibility of types is determined by their structure
- B. A type system where types must be explicitly declared
- C. A type system where types are determined by their names
- D. A type system where types can change dynamically

110. **\*\*Which of the following best describes structural typing with object literals?\*\***

- A. Objects are compatible if they have the same structure
- B. Objects are compatible if they have the same name
- C. Objects are compatible if they have the same type
- D. Objects are not compatible

### ### Topic: nested\_objects

111. **\*\*How do you define a nested object type in TypeScript?\*\***

- A. ``let obj: { name: string, address: { street: string, city: string } };``
- B. ``let obj = { name: string, address: { street: string, city: string } };``
- C. ``let obj: { name: string; address: { street: string; city: string; }; } = {};`
- D. ``let obj = { name: "string", address: { street: "string", city: "string" } };``

112. **\*\*Which of the following is a valid nested object in TypeScript?\*\***

- A. ``{ name: 'John', address: { street: 'Main St', city: 'New York' } }``
- B. ``{ 'name': 'John', 'address': { 'street': 'Main St', 'city': 'New York' } }``
- C. ``{ name: 'John', address: { street: 'Main St', city: 'NY' } }``
- D. ``{ name: 'John', address: { street: 'Main St', city: 'New York', country: 'USA' } }``

### ### Topic: intersection\_types

113. **\*\*What is an intersection type in TypeScript?\*\***

- A. A type that combines multiple types into one
- B. A type that can be one of several types
- C. A type that extends another type
- D. A type that can be any type

114. **\*\*How do you define an intersection type in TypeScript?\*\***

- A. ``type Combined = Type1 & Type2;``
- B. ``type Combined = Type1 | Type2;``



- C. ``type Combined = (Type1, Type2);``
- D. ``type Combined = { Type1, Type2 };``

### Topic: `any__unknown_never_types`

115. **\*\*What is the ``any`` type in TypeScript?\*\***

- A. A type that can be any type
- B. A type that must be explicitly defined
- C. A type that is only used for numbers
- D. A type that cannot be changed

116. **\*\*What is the ``unknown`` type in TypeScript?\*\***

- A. A type-safe counterpart of ``any``
- B. A type that can be any type
- C. A type that is only used for strings
- D. A type that must be explicitly defined

117. **\*\*What is the ``never`` type in TypeScript?\*\***

- A. A type that represents values that never occur
- B. A type that can be any type
- C. A type that must be explicitly defined
- D. A type that is only used for functions

118. **\*\*Which of the following is a correct use of the ``never`` type in TypeScript?\*\***

- A. ``function error(message: string): never { throw new Error(message); }``
- B. ``function error(message: string): void { throw new Error(message); }``

- C. ``function error(message: string): any { throw new Error(message); }``
- D. ``function error(message: string): unknown { throw new Error(message); }``

### Topic: explicit\_casting

119. **\*\*How do you explicitly cast a variable in TypeScript?\*\***

- A. ``let num: number = <number>value;``
- B. ``let num: number = (number)value;``
- C. ``let num: number = value as number;``
- D. ``let num: number = (number)value as number;``

120. **\*\*Which of the following is a valid explicit casting in TypeScript?\*\***

- A. ``let str: string = value as string;``
- B. ``let str: string = (string)value;``
- C. ``let str: string = <string>value;``
- D. All of the above

### Topic: enum

121. **\*\*What is an enum in TypeScript?\*\***

- A. A way to define a set of named constants
- B. A way to define a variable
- C. A way to define a function
- D. A way to define an array

122. **\*\*How do you define an enum in TypeScript?\*\***

- A.

```
``typescript
enum Colors {
    Red,
    Green,
    Blue
}
``
```

- B.

```
``typescript
enum Colors {
    Red = 'Red',
    Green = 'Green',
    Blue = 'Blue'
}
``
```

- C.

```
``typescript
enum Colors {
    'Red',
    'Green',
    'Blue'
}
``
```

- D.

```
``typescript
```

```
enum Colors = {  
    Red,  
    Green,  
    Blue  
}  
...
```

### Topic: const\_enum

123. \*\*What is a `const enum` in TypeScript?\*\*

- A. An enum that is inlined and optimized at compile time
- B. An enum that cannot be changed
- C. An enum that is mutable
- D. An enum that must be explicitly declared

124. \*\*How do you define a `const enum` in TypeScript?\*\*

- A.

```
```typescript  
const enum Colors {  
    Red,  
    Green,  
    Blue  
}  
...
```

- B.

```
```typescript
```

```
const enum Colors {  
    Red = 'Red',  
    Green = 'Green',  
    Blue = 'Blue'  
}
```

...

- C.

```
```typescript
```

```
const enum Colors {  
    'Red',  
    'Green',  
    'Blue'  
}
```

...

- D.

```
```typescript
```

```
const enum Colors = {  
    Red,  
    Green,  
    Blue  
}
```

...

### Topic: arrays

125. **\*\*How do you define an array in TypeScript?\*\***

- A. ``let arr: number[] = [1, 2, 3];``
- B. ``let arr: Array<number> = [1, 2, 3];``
- C. ``let arr = [1, 2, 3];``
- D. All of the above

126. **\*\*Which of the following is a valid way to declare a tuple in TypeScript?\*\***

- A. ``let tuple: [number, string] = [1, 'hello'];``
- B. ``let tuple = [1, 'hello'];``
- C. ``let tuple: [number, string]; tuple = [1, 'hello'];``
- D. All of the above

### Topic: functions

127. **\*\*How do you declare a function in TypeScript?\*\***

- A.

```
```typescript
```

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

```
```
```

- B.

```
```typescript
```

```
let add = (a: number, b: number): number => {
    return a + b;
}
```

```
```
```

- C.

```
```typescript
```

```
let add: (a: number, b: number) => number = function(a, b) {  
    return a + b;  
}
```

```
```
```

- D. All of the above

128. \*\*Which of the following is a valid way to define a function with an optional parameter in TypeScript?\*\*

- A.

```
```typescript
```

```
function greet(name: string, age?: number): void {  
    console.log(`Hello, ${name}!`);  
}
```

```
```
```

- B.

```
```typescript
```

```
function greet(name: string, age: number | undefined): void {  
    console.log(`Hello, ${name}!`);  
}
```

```
```
```

- C.

```
```typescript
```

```
function greet(name: string, age: number = 0): void {
```

```
    console.log(`Hello, ${name}!`);  
  }  
  ...
```

- D.

```
```typescript
```

```
function greet(name: string, age: number | null): void {  
    console.log(`Hello, ${name}!`);  
}  
...
```

### Topic: function\_optional\_parameter

129. **\*\*How do you specify an optional parameter in a TypeScript function?\*\***

- A. ``function greet(name: string, age?: number): void``
- B. ``function greet(name: string, age: number?): void``
- C. ``function greet(name: string, age: number | undefined): void``
- D. ``function greet(name: string, age: number = 0): void``

130. **\*\*Which of the following demonstrates a function with an optional parameter?\*\***

- A.

```
```typescript
```

```
function greet(name: string, age?: number): void {  
    console.log(`Hello, ${name}!`);  
}  
...
```

- B.



```
```typescript
```

```
function greet(name: string, age: number | undefined): void {
```

```
    console.log(`Hello, ${name}!`);
```

```
}
```

```
```
```

- C.

```
```typescript
```

```
function greet(name: string, age: number = 0): void {
```

```
    console.log(`Hello, ${name}!`);
```

```
}
```

```
```
```

- D.

```
```typescript
```

```
function greet(name: string, age: number | null): void {
```

```
    console.log(`Hello, ${name}!`);
```

```
}
```

```
```
```

### Topic: function\_default\_parameter

131. **\*\*How do you specify a default parameter in a TypeScript function?\*\***

- A. ``function greet(name: string, age: number = 0): void``

- B. ``function greet(name: string, age?: number): void``

- C. ``function greet(name: string, age: number | undefined): void``

- D. ``function greet(name: string, age: number | null): void``

132. **\*\*Which of the following demonstrates a function with a default parameter?\*\***

- A.

```
```typescript
```

```
function greet(name: string, age: number = 0): void {  
    console.log(`Hello, ${name}!`);  
}
```

```
```
```

- B.

```
```typescript
```

```
function greet(name: string, age?: number): void {  
    console.log(`Hello, ${name}!`);  
}
```

```
```
```

- C.

```
```typescript
```

```
function greet(name: string, age: number | undefined): void {  
    console.log(`Hello, ${name}!`);  
}
```

```
```
```

- D.

```
```typescript
```

```
function greet(name: string, age: number | null): void {  
    console.log(`Hello, ${name}!`);  
}
```

```
```
```

### ### Topic: function\_rest\_parameter

133. **\*\*How do you specify a rest parameter in a TypeScript function?\*\***

- A. ``function add(...numbers: number[]): number``
- B. ``function add(...numbers?: number[]): number``
- C. ``function add(...numbers: number[] = []): number``
- D. ``function add(...numbers: number[] | undefined): number``

### ### Topic: tuples (continued)

141. **\*\*Which TypeScript feature allows you to define a fixed-length array with specified types for each element?\*\***

- A. Enum
- B. Array
- C. Tuple
- D. Object

142. **\*\*How do you access the second element in a TypeScript tuple ``let tuple: [number, string] = [1, 'hello'];``?**

- A. ``tuple[1]``
- B. ``tuple[0]``
- C. ``tuple[2]``
- D. ``tuple['hello']``

143. **\*\*Which of the following is a valid way to define a tuple with mixed types in TypeScript?**

- A. ``let mixed: [number, boolean, string] = [42, true, 'hello'];``

- B. ``let mixed: [number, boolean, string] = [42, 'true', 'hello'];``
- C. ``let mixed: [number, boolean, string] = ['42', true, 'hello'];``
- D. ``let mixed: [number, boolean, string] = [42, true, 123];``

144. **\*\*How can you update the value of the second element in a TypeScript tuple ``let tuple: [number, string] = [1, 'hello'];``?**

- A. ``tuple[1] = 'world';``
- B. ``tuple[0] = 'world';``
- C. ``tuple[2] = 'world';``
- D. ``tuple['hello'] = 'world';``

145. **\*\*Which of the following is true about tuples in TypeScript?**

- A. Tuples allow you to define an array with a fixed number of elements.
- B. Tuples allow you to define an array with elements of the same type.
- C. Tuples allow you to define an array with a variable number of elements.
- D. Tuples allow you to define an array with elements of any type.

### Topic: helloworld (continued)

146. **\*\*Which TypeScript keyword is used to declare a variable?**

- A. ``var``
- B. ``let``
- C. ``const``
- D. All of the above

147. **\*\*How do you compile a TypeScript file named ``hello.ts`` to JavaScript?**

- A. ``tsc hello.ts``
- B. ``ts hello.ts``
- C. ``node hello.ts``
- D. ``npm hello.ts``

148. **\*\*What is the output of the following TypeScript code? `console.log('Hello, World!');`\*\***

- A. ``Hello, World!``
- B. ``hello, world!``
- C. ``Hello, world!``
- D. ``hello, World!``

149. **\*\*Which of the following is a valid TypeScript comment?\*\***

- A. ``// This is a comment``
- B. ``/* This is a comment */``
- C. Both A and B
- D. None of the above

150. **\*\*How do you define a string variable in TypeScript?\*\***

- A. ``let greeting: string = 'Hello, World!';``
- B. ``let greeting = 'Hello, World!';``
- C. ``let greeting: 'Hello, World!';``
- D. ``let greeting = Hello, World!;``

### Topic: `json_objects` (continued)

151. **\*\*How do you parse a JSON string in TypeScript?\*\***

- A. ``JSON.parse(jsonString)``
- B. ``JSON.stringify(jsonString)``
- C. ``JSON.convert(jsonString)``
- D. ``JSON.toString(jsonString)``

152. **\*\*Which TypeScript type is typically used to represent a parsed JSON object?\*\***

- A. ``object``
- B. ``any``
- C. ``string``
- D. ``JSON``

153. **\*\*How do you convert a TypeScript object to a JSON string?\*\***

- A. ``JSON.stringify(object)``
- B. ``JSON.parse(object)``
- C. ``JSON.convert(object)``
- D. ``JSON.toString(object)``

154. **\*\*What is the output of the following TypeScript code? ``JSON.stringify({ name: 'John', age: 30 })``\*\***

- A. ``{"name":"John","age":30}``
- B. ``{ name: 'John', age: 30 }``
- C. ``['John', 30]``
- D. ``null``

155. **\*\*Which method would you use to deeply copy a JSON object in TypeScript?\*\***

- A. ``JSON.parse(JSON.stringify(object))``
- B. ``Object.assign({}, object)``

- C. ``Object.create(object)``
- D. ``object.clone()``

### Topic: syntax\_error (continued)

156. **\*\*What is a syntax error in TypeScript?\*\***

- A. An error due to incorrect syntax
- B. An error due to incorrect type
- C. An error due to incorrect logic
- D. An error due to incorrect runtime behavior

157. **\*\*Which of the following will cause a syntax error in TypeScript?\*\***

- A. ``let name = 'John;``
- B. ``let age = 30;``
- C. ``let isActive: boolean = true;``
- D. ``const PI = 3.14;``

158. **\*\*How can you identify syntax errors in TypeScript?\*\***

- A. By running the TypeScript compiler
- B. By running the JavaScript engine
- C. By using the ``console.log`` method
- D. By using a debugger

159. **\*\*Which tool helps in identifying syntax errors during development in TypeScript?\*\***

- A. TypeScript compiler (tsc)
- B. Node.js runtime

- C. npm
- D. Git

160. **\*\*Which of the following is a valid TypeScript variable declaration?\*\***

- A. ``let age: number = 25;``
- B. ``let age number = 25;``
- C. ``let age: number 25;``
- D. ``let age = number 25;``

### Topic: type\_error (continued)

161. **\*\*What is a type error in TypeScript?\*\***

- A. An error due to incorrect type assignment
- B. An error due to incorrect syntax
- C. An error due to incorrect logic
- D. An error due to incorrect runtime behavior

162. **\*\*Which of the following will cause a type error in TypeScript?\*\***

- A. ``let name: string = 123;``
- B. ``let age: number = 30;``
- C. ``let isActive: boolean = true;``
- D. ``const PI = 3.14;``

163. **\*\*How can you identify type errors in TypeScript?\*\***

- A. By running the TypeScript compiler
- B. By running the JavaScript engine



- C. By using the `console.log` method
- D. By using a debugger

164. **\*\*Which tool helps in identifying type errors during development in TypeScript?\*\***

- A. TypeScript compiler (tsc)
- B. Node.js runtime
- C. npm
- D. Git

165. **\*\*Which of the following is a valid TypeScript variable declaration with type?\*\***

- A. `let age: number = 25;`
- B. `let age number = 25;`
- C. `let age: number 25;`
- D. `let age = number 25;`

### Topic: assignability\_error (continued)

166. **\*\*What is an assignability error in TypeScript?\*\***

- A. An error due to incorrect assignment between types
- B. An error due to incorrect syntax
- C. An error due to incorrect logic
- D. An error due to incorrect runtime behavior

167. **\*\*Which of the following will cause an assignability error in TypeScript?\*\***

- A. `let name: string = 123;`
- B. `let age: number = 30;`

- C. `let isActive: boolean = true;`
- D. `const PI = 3.14;`

168. **\*\*How can you identify assignability errors in TypeScript?\*\***

- A. By running the TypeScript compiler
- B. By running the JavaScript engine
- C. By using the `console.log` method
- D. By using a debugger

169. **\*\*Which tool helps in identifying assignability errors during development in TypeScript?\*\***

- A. TypeScript compiler (tsc)
- B. Node.js runtime
- C. npm
- D. Git

170. **\*\*Which of the following is a valid TypeScript variable assignment?\*\***

- A. `let age: number = 25;`
- B. `let age: number = '25';`
- C. `let age: number = true;`
- D. `let age: number = {};`

### Topic: strong\_typing (continued)

171. **\*\*What does strong typing mean in TypeScript?\*\***

- A. Enforcing type rules strictly
- B. Allowing dynamic type assignments

- C. Allowing type coercion

### Topic: strong\_typing (continued)

172. **\*\*Which TypeScript feature helps enforce strong typing?\*\***

- A. Type annotations
- B. Dynamic typing
- C. Type inference
- D. Type coercion

173. **\*\*What is the benefit of strong typing in TypeScript?\*\***

- A. It helps catch type-related errors at compile time.
- B. It allows for faster execution.
- C. It simplifies the code.
- D. It removes the need for type definitions.

174. **\*\*Which of the following correctly demonstrates strong typing in TypeScript?\*\***

- A. `let age: number = 30;`
- B. `let age: any = '30';`
- C. `let age = '30';`
- D. `let age: unknown = 30;`

175. **\*\*How does TypeScript handle type mismatches with strong typing?\*\***

- A. It reports errors at compile time.
- B. It ignores the mismatch and runs the code.
- C. It automatically converts types.

- D. It throws runtime exceptions.

### Topic: const\_let (continued)

176. **\*\*Which keyword allows reassignment of variables in TypeScript?\*\***

- A. `let`
- B. `const`
- C. `var`
- D. `readonly`

177. **\*\*What is the primary difference between `const` and `let` in TypeScript?**

- A. `const` cannot be reassigned, while `let` can.
- B. `const` can be reassigned, while `let` cannot.
- C. `const` is used for function declarations, while `let` is used for variable declarations.
- D. There is no difference.

178. **\*\*Which keyword would you use if you need to declare a variable whose value can change?**

- A. `let`
- B. `const`
- C. `var`
- D. `readonly`

179. **\*\*Which of the following is a valid usage of `const` in TypeScript?**

- A. `const PI = 3.14;`
- B. `const name: string = 'Alice';`
- C. `const numbers: number[] = [1, 2, 3];`

- D. All of the above

180. **\*\*Can you reassign a variable declared with `let` in TypeScript?\*\***

- A. Yes
- B. No
- C. Only within the same block scope
- D. Only if it's initialized with `null`

### Topic: modules (continued)

181. **\*\*How do you import a default export from a module in TypeScript?\*\***

- A. `import defaultExport from './module';`
- B. `import { defaultExport } from './module';`
- C. `import * as defaultExport from './module';`
- D. `import './module';`

182. **\*\*Which keyword is used to export a member from a TypeScript module?\*\***

- A. `export`
- B. `import`
- C. `require`
- D. `module`

183. **\*\*How do you export multiple members from a TypeScript module?\*\***

- A.

```
```typescript
```

```
export const a = 1;
```

```
export const b = 2;
```

```
...
```

- B.

```
``typescript
```

```
export { a, b };
```

```
...
```

- C.

```
``typescript
```

```
module.exports = { a, b };
```

```
...
```

- D.

```
``typescript
```

```
export default { a, b };
```

```
...
```

184. **\*\*Which TypeScript feature allows you to use modules in a file?\*\***

- A. ``import`` and ``export``
- B. ``require`` and ``module.exports``
- C. ``include`` and ``exclude``
- D. ``use`` and ``end``

185. **\*\*How do you import all members from a module into a single object in TypeScript?\*\***

- A. ``import * as module from './module';``
- B. ``import { * } from './module';``
- C. ``import module from './module';``
- D. ``import { all } from './module';``

### Topic: native\_ECMAScript\_modules (continued)

186. **\*\*What is the syntax to use native ECMAScript modules in TypeScript?\*\***

- A. ``import { member } from './module';``
- B. ``require('./module');``
- C. ``import module = require('./module');``
- D. ``include('./module');``

187. **\*\*How do you specify module format in `tsconfig.json` for native ECMAScript modules?**

- A. ```"module": "ES6"```
- B. ```"module": "CommonJS"```
- C. ```"module": "AMD"```
- D. ```"module": "UMD"```

188. **\*\*Which of the following is a benefit of using native ECMAScript modules?**

- A. Standardized syntax for imports and exports
- B. Improved performance with dynamic loading
- C. Better support for asynchronous code
- D. Both A and B

189. **\*\*How do you ensure compatibility with native ECMAScript modules in TypeScript?**

- A. Use ``import`` and ``export`` statements
- B. Use ``require`` and ``module.exports``
- C. Use ``include`` and ``exclude``
- D. Use ``import`` with ``require``

190. **\*\*What is the file extension for TypeScript files using native ECMAScript modules?\*\***

- A. `.ts``
- B. `.js``
- C. `.mjs``
- D. `.d.ts``

### Topic: `import_inquirer_ECMAScript_module` (continued)

191. **\*\*How do you import the ``inquirer`` module in a TypeScript file using ECMAScript modules?\*\***

- A. ``import * as inquirer from 'inquirer';``
- B. ``import { inquirer } from 'inquirer';``
- C. ``import inquirer from 'inquirer';``
- D. ``import 'inquirer';``

192. **\*\*Which of the following demonstrates a correct way to import a named export from ``inquirer`` in TypeScript?\*\***

- A. ``import { prompt } from 'inquirer';``
- B. ``import prompt from 'inquirer';``
- C. ``import 'prompt' from 'inquirer';``
- D. ``import { 'prompt' } from 'inquirer';``

193. **\*\*How do you use ``inquirer`` in TypeScript to prompt the user for input?\*\***

- A.

````typescript`

`import { prompt } from 'inquirer';`



```
prompt({ type: 'input', name: 'name', message: 'What is your name?' })
```

```
.then(answers => console.log(answers));
```

```
...
```

- B.

```
```typescript
```

```
import inquirer from 'inquirer';
```

```
inquirer.prompt({ type: 'input', name: 'name', message: 'What is your name?' })
```

```
.then(answers => console.log(answers));
```

```
...
```

- C.

```
```typescript
```

```
import { inquirer } from 'inquirer';
```

```
inquirer.prompt({ type: 'input', name: 'name', message: 'What is your name?' })
```

```
.then(answers => console.log(answers));
```

```
...
```

- D.

```
```typescript
```

```
import 'inquirer';
```

```
inquirer.prompt({ type: 'input', name: 'name', message: 'What is your name?' })
```

```
.then(answers => console.log(answers));
```

```
...
```

194. **\*\*Which method is used to prompt a list of options with `inquirer` in TypeScript?\*\***

- A. ``prompt()``
- B. ``list()``
- C. ``choice()``
- D. ``query()``

195. **\*\*How do you handle the promise returned by `inquirer.prompt()` in TypeScript?\*\***

- A. Using ``.then()`` method
- B. Using ``.catch()`` method
- C. Using ``async/await`` syntax
- D. Both A and C

### Topic: chalk (continued)

196. **\*\*How do you import the `chalk` module in a TypeScript file?\*\***

- A. ``import chalk from 'chalk';``
- B. ``import * as chalk from 'chalk';``
- C. ``import { chalk } from 'chalk';``
- D. ``import 'chalk';``

197. **\*\*Which `chalk` method is used to style text in red?\*\***

- A. ``chalk.red()``
- B. ``chalk.color('red')``
- C. ``chalk.style('red')``
- D. ``chalk.setColor('red')``

198. **\*\*How do you chain multiple styles using `chalk`?\***

- A.

```
````typescript
```

```
chalk.red.bold.underline('Styled text')
```

```
````
```

- B.

```
````typescript
```

```
chalk.red.bold('Styled text').underline()
```

```
````
```

- C.

```
````typescript
```

```
chalk.red.bold('Styled text').underline
```

```
````
```

- D.

```
````typescript
```

```
chalk.red('Styled text').bold.underline()
```

```
````
```

199. **\*\*What does the `chalk.bgGreen()` method do?\***

- A.

Sets the background color to green

- B. Sets the text color to green

- C. Underlines the text

- D. Makes the text bold

200. **\*\*How do you use `chalk` to style a string with multiple colors and styles?\*\***

- A.

```
````typescript
```

```
chalk.red.bold('Red and Bold') + chalk.blue.underline(' Blue and Underline')
```

```
````
```

- B.

```
````typescript
```

```
chalk.red('Red').bold + chalk.blue.underline(' Blue and Underline')
```

```
````
```

- C.

```
````typescript
```

```
chalk.red.bold('Red').blue.underline(' Blue')
```

```
````
```

- D.

```
````typescript
```

```
chalk.red.bold('Red') + chalk.blue(' Blue')
```

```
````
```

### Topic: unions\_literals (continued)

201. **\*\*Which of the following is an example of a union type in TypeScript?\*\***

- A. `let value: string | number;`

- B. `let value: string & number;`

- C. `let value: string | boolean | number;`

- D. `let value: (string | number)[];`

202. **\*\*How do you specify a union type that includes both string literals and number literals?\*\***

- A. ``let value: 'string' | 123;``
- B. ``let value: string | number;``
- C. ``let value: 'string' | 'number';``
- D. ``let value: string & number;``

203. **\*\*Which TypeScript feature allows you to define a variable that can hold either a string or a number?\*\***

- A. Union types
- B. Intersection types
- C. Literal types
- D. Tuple types

204. **\*\*How do you use a union type with a literal type in TypeScript?\*\***

- A.

```
```typescript
```

```
let value: 'hello' | 'world';
```

```
value = 'hello'; // valid
```

```
value = 'world'; // valid
```

```
value = 'other'; // error
```

```
```
```

- B.

```
```typescript
```

```
let value: 'hello' | number;
```

```
value = 'hello'; // valid
```

```
value = 42; // valid
```

```
value = true; // error
```

...

- C.

```
```typescript
```

```
let value: string | 42;
```

```
value = 'hello'; // valid
```

```
value = 42; // valid
```

```
value = true; // error
```

...

- D.

```
```typescript
```

```
let value: 'hello' | number;
```

```
value = 'hello'; // valid
```

```
value = 'world'; // error
```

...

205. **\*\*Which of the following is a valid union type declaration?\*\***

- A. `let value: string | number | boolean;`

- B. `let value: string & number;`

- C. `let value: (string | number)[];`

- D. `let value: string | (number | boolean);`

### Topic: nested\_objects (continued)

221. **\*\*How do you define a nested object type in TypeScript?\*\*** (continued)

- A.

```
```typescript
```

```
type Address = { street: string; city: string; };  
  
type Person = { name: string; address: Address; };  
  
...
```

- B.

```
```typescript  
  
type Address = { street: string; city: string };  
  
type Person = { name: string; address: { street: string; city: string } };  
  
...
```

- C.

```
```typescript  
  
interface Address { street: string; city: string; }  
  
interface Person { name: string; address: Address; }  
  
...
```

- D.

```
```typescript  
  
type Address = { street: string; city: string; };  
  
type Person = { name: string; address: { street: string; city: string } };  
  
...
```

222. **\*\*How do you access a nested property in a TypeScript object?\*\***

- A. ``obj.address.city``
- B. ``obj['address']['city']``
- C. ``obj.address['city']``
- D. All of the above

223. **\*\*How do you define an optional nested property in a TypeScript object type?\*\***

- A. ``type Person = { name: string; address?: { street: string; city: string; }; };``
- B. ``type Person = { name: string; address: { street?: string; city?: string; }; };``
- C. ``type Person = { name: string; address?: { street: string; city: string; }; };``
- D. ``type Person = { name: string; address?: { street: string; city: string; }; };``

224. **\*\*Which of the following represents an optional nested property in a TypeScript interface?\*\***

- A. ``interface Person { name: string; address?: { street: string; city: string; }; }``
- B. ``interface Person { name: string; address: { street?: string; city?: string; }; }``
- C. ``interface Person { name: string; address?: { street: string; city: string; }; }``
- D. ``interface Person { name: string; address: { street: string; city: string; }; }``

225. **\*\*How do you handle default values for nested properties in TypeScript?\*\***

- A. By initializing properties in the constructor
- B. By using default values in type definitions
- C. By using ``null`` or ``undefined`` as default
- D. By providing default values in the object literal

### Topic: interfaces (continued)

226. **\*\*How do you declare an interface in TypeScript?\*\***

- A.

```
```typescript
```

```
interface Person {
```

```
  name: string;
```

```
  age: number;
```

```
}
```



...

- B.

```typescript

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

...

- C.

```typescript

```
class Person {
```

```
    name: string;
```

```
    age: number;
```

```
}
```

...

- D.

```typescript

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

...

227. \*\*Which keyword is used to extend an interface in TypeScript?\*\*

- A. `extends`

- B. `implements`

- C. `inherits`

- D. `inheritsFrom`

228. \*\*How can you extend multiple interfaces in TypeScript?\*\*

- A.

```typescript

```
interface Person extends Contact, Address {
```

```
    name: string;
```

```
}
```

```
...
```

- B.

```
```typescript
```

```
interface Person extends Contact, Address {
```

```
    name: string;
```

```
}
```

```
...
```

- C.

```
```typescript
```

```
interface Person extends Contact & Address {
```

```
    name: string;
```

```
}
```

```
...
```

- D.

```
```typescript
```

```
interface Person implements Contact, Address {
```

```
    name: string;
```

```
}
```

```
...
```

229. **\*\*Which of the following is correct regarding interface merging in TypeScript?\*\***

- A. Interfaces with the same name will be merged automatically.

- B. Interfaces with the same name will override each other.

- C. Only the last interface definition will be used.
- D. Type aliases with the same name will be merged.

230. **\*\*What is the purpose of using interfaces in TypeScript?\*\***

- A. To define the shape of objects
- B. To create classes
- C. To define functions
- D. To perform type assertions

### Topic: classes (continued)

231. **\*\*How do you declare a class in TypeScript?\*\***

- A.

```
```typescript
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;
  }
}
```
```

- B.

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

...

- C.

```typescript

```
function Person(name: string) {
```

```
    this.name = name;
```

```
}
```

...

- D.

```typescript

```
type Person = { name: string };
```

...

232. **\*\*Which keyword is used to define a class property that can only be accessed within the class?\*\***

- A. `private`

- B. `protected`

- C. `public`

- D. `readonly`

233. **\*\*What does the `protected` keyword do in a TypeScript class?\*\***

- A. Allows access to the property in the class and its subclasses.

- B. Restricts access to the property only within the class.

- C. Allows access to the property only within the same package.

- D. Makes the property immutable.

### Topic: generics (continued)

242. **\*\*How do you define a generic function in TypeScript?\*\*** (continued)

- A.

```
```typescript
function identity<T>(value: T): T {
    return value;
}
```
```

- B.

```
```typescript
function identity<T>(value: T): T {
    return value;
}
```
```

- C.

```
```typescript
function identity(value: any): any {
    return value;
}
```
```

- D.

```
```typescript
function identity(value: string): string {
    return value;
}
```
```

243. **\*\*How do you use a generic type with a class in TypeScript?\*\***

- A.

```
```typescript
```

```
class Box<T> {
```

```
    private value: T;
```

```
    constructor(value: T) {
```

```
        this.value = value;
```

```
    }
```

```
    getValue(): T {
```

```
        return this.value;
```

```
    }
```

```
}
```

```
```
```

- B.

```
```typescript
```

```
class Box<T> {
```

```
    value: T;
```

```
    constructor(value: T) {
```

```
        this.value = value;
```

```
    }
```

```
    getValue(): T {
```

```
        return this.value;
```

```
    }
```

```
}
```

```
```
```

- C.

```
```typescript
```

```
class Box<T> {  
  private value: T;  
  
  constructor(value: T) {  
    this.value = value;  
  }  
  
  getValue(): T {  
    return this.value;  
  }  
}
```

```
```
```

- D.

```
```typescript
```

```
class Box<T> {  
  value: T;  
  
  constructor(value: T) {  
    this.value = value;  
  }  
  
  getValue(): T {  
    return this.value;  
  }  
}
```

```
```
```

244. **\*\*How do you constrain a generic type in TypeScript?\*\***

- A.

```
```typescript
```

```
function logLength<T extends { length: number }>(item: T): void {  
    console.log(item.length);  
}
```

```
```
```

- B.

```
```typescript
```

```
function logLength<T>(item: T & { length: number }): void {  
    console.log(item.length);  
}
```

```
```
```

- C.

```
```typescript
```

```
function logLength<T>(item: { length: number }): void {  
    console.log(item.length);  
}
```

```
```
```

- D.

```
```typescript
```

```
function logLength<T extends { length: number }>(item: T): void {  
    console.log(item.length);  
}
```

```
```
```

245. **\*\*What is the purpose of using the `default` keyword in generic types?\*\***

- A. To specify a default type when none is provided



- B. To override the generic type with a specific type
- C. To make the generic type required
- D. To create a new generic type

### Topic: type\_inference (continued)

246. **What is type inference in TypeScript?**

- A. The automatic determination of variable types by the compiler
- B. The manual specification of variable types
- C. The conversion of one type to another
- D. The exclusion of type checking

247. **How does TypeScript infer types for function return values?**

- A. Based on the return statement in the function
- B. Based on the function parameter types
- C. Based on the function name
- D. Based on the function body

248. **What is the result of not specifying a type for a variable in TypeScript?**

- A. TypeScript infers the type based on the assigned value
- B. TypeScript treats the variable as `any`
- C. TypeScript throws an error
- D. TypeScript assigns the type `unknown`

249. **How do you explicitly specify a type for a variable that TypeScript cannot infer?**

- A. Using a type annotation

- B. Using type assertions
- C. Using type inference
- D. Using a type alias

250. **\*\*Which of the following allows TypeScript to infer a type?\*\***

- A. Initializing a variable with a value
- B. Defining a variable without a value
- C. Using a generic type without providing a specific type
- D. Using an empty object as a value

### Topic: tuples (continued)

251. **\*\*How do you declare a tuple type in TypeScript?\*\***

- A.

```
```typescript
```

```
let tuple: [number, string] = [1, 'one'];
```

```
```
```

- B.

```
```typescript
```

```
let tuple: [string, number] = ['one', 1];
```

```
```
```

- C.

```
```typescript
```

```
let tuple: [number, string] = [1, 'one'];
```

```
```
```

- D.

```
``typescript
```

```
let tuple: [string, number] = ['one', 1];
```

```
...
```

252. **\*\*What happens if you try to assign a tuple with more elements than its type definition?\*\***

- A. TypeScript throws an error
- B. The extra elements are ignored
- C. The tuple automatically adjusts its type
- D. TypeScript allows any number of elements

253. **\*\*How do you access individual elements in a TypeScript tuple?\*\***

- A. Using index notation, e.g., `tuple[0]`
- B. Using property names
- C. Using the `get` method
- D. Using a loop

254. **\*\*How can you specify optional elements in a tuple type?\*\***

- A.

```
``typescript
```

```
let tuple: [number, string?] = [1];
```

```
...
```

- B.

```
``typescript
```

```
let tuple: [number, string?] = [1, 'one'];
```

```
...
```

- C.

```typescript

```
let tuple: [number?, string] = ['one'];
```

```

- D.

```typescript

```
let tuple: [number, string?] = [1];
```

```

255. \*\*How do you declare a tuple with mixed types and default values in TypeScript?\*\*

- A.

```typescript

```
let tuple: [number, string, boolean] = [1, 'one', true];
```

```

- B.

```typescript

```
let tuple: [number, string, boolean?] = [1, 'one'];
```

```

- C.

```typescript

```
let tuple: [number, string] = [1, 'one'];
```

```

- D.

```typescript

```
let tuple: [number, string, boolean] = [1, 'one', true];
```

```

### ### Topic: enums (continued)

256. \*\*How do you define an enum in TypeScript?\*\*

- A.

```
```typescript
enum Color {
    Red,
    Green,
    Blue
}
```
```

- B.

```
```typescript
enum Color {
    Red = 1,
    Green = 2,
    Blue = 3
}
```
```

- C.

```
```typescript
enum Color {
    Red = 'RED',
    Green = 'GREEN',
    Blue = 'BLUE'
}
```
```

...

- D.

```typescript

enum Color {

Red = 0,

Green = 1,

Blue = 2

}

...

257. **What is the default value of the first enum member in TypeScript if not explicitly set?**

- A. `0`

- B. `1`

- C. `null`

- D. `undefined`

258. **How do you access an enum member's name and value?**

- A.

```typescript

console.log(Color.Red); // Value

console.log(Color[0]); // Name

...

- B.

```typescript

console.log(Color[0]); // Value

console.log(Color.Red); // Name

...

- C.

```
```typescript
```

```
console.log(Color.Red); // Name
```

```
console.log(Color[0]); // Value
```

...

- D.

```
```typescript
```

```
console.log(Color.Red); // Value
```

```
console.log(Color[0]); // Value
```

...

259. **\*\*How can you create a string-based enum in TypeScript?\*\***

- A.

```
```typescript
```

```
enum Color {
```

```
    Red = 'RED',
```

```
    Green = 'GREEN',
```

```
    Blue = 'BLUE'
```

```
}
```

...

- B.

```
```typescript
```

```
enum Color {
```

```
    Red = 1,
```

```
    Green = 2,
```

```
    Blue = 3
  }
  ...

- C.

```typescript
enum Color {

  Red = 'Red',

  Green = 'Green',

  Blue = 'Blue'

}
...
```

```
- D.

```typescript
enum Color {

  Red = 0,

  Green = 1,

  Blue = 2

}
...
```

260. **\*\*What is the benefit of using enums in TypeScript?\*\***

- A. Provides meaningful names for numeric values
- B. Allows for unlimited numbers of values
- C. Automatically assigns default values
- D. Simplifies the creation of classes



### ### Topic: functions (continued)

261. **\*\*How do you define a function type in TypeScript?\*\***

- A.

```
```typescript
```

```
type Greeting = (name: string) => string;
```

```
```
```

- B.

```
```typescript
```

```
function Greeting(name: string): string {
```

```
    return `Hello, ${
```

```
name}`;
```

```
}
```

```
```
```

- C.

```
```typescript
```

```
type Greeting = string => string;
```

```
```
```

- D.

```
```typescript
```

```
type Greeting = (name: string): string => `Hello, ${name}`;
```

```
```
```

262. **\*\*What does the `void` return type indicate in a TypeScript function?\*\***

- A. The function does not return a value

- B. The function returns an `undefined` value
- C. The function can return any type
- D. The function returns a value of type `null`

263. **\*\*How do you define a function with optional parameters in TypeScript?\*\***

- A.

```
```typescript
function greet(name: string, age?: number): string {
    return `Hello, ${name}`;
}
...
```
```

- B.

```
```typescript
function greet(name: string, age?: number): string {
    return `Hello, ${name}`;
}
...
```
```

- C.

```
```typescript
function greet(name: string, age: number = 30): string {
    return `Hello, ${name}`;
}
...
```
```

- D.

```
```typescript
function greet(name: string, age: number): string {

```

```
    return `Hello, ${name}`;  
  }  
  ...
```

264. **\*\*How do you define a function that accepts a variable number of arguments in TypeScript?\*\***

- A.

```
```typescript  
  
function sum(...numbers: number[]): number {  
  return numbers.reduce((a, b) => a + b, 0);  
}  
...
```

- B.

```
```typescript  
  
function sum(numbers: number[]): number {  
  return numbers.reduce((a, b) => a + b, 0);  
}  
...
```

- C.

```
```typescript  
  
function sum(numbers: number): number {  
  return numbers.reduce((a, b) => a + b, 0);  
}  
...
```

- D.

```
```typescript  
  
function sum(...numbers: number): number {
```

```
    return numbers.reduce((a, b) => a + b, 0);  
  }  
  ...
```

265. **\*\*How can you specify a default parameter value in a TypeScript function?\*\***

- A.

```
```typescript  
function greet(name: string = 'Guest'): string {  
    return `Hello, ${name}`;  
}  
...
```

- B.

```
```typescript  
function greet(name: string = 'Guest'): string {  
    return `Hello, ${name}`;  
}  
...
```

- C.

```
```typescript  
function greet(name: string, age: number = 30): string {  
    return `Hello, ${name}`;  
}  
...
```

- D.

```
```typescript  
function greet(name: string, age?: number): string {
```

```
    return `Hello, ${name}`;  
  }  
  ...
```

### Topic: type\_assertions (continued)

266. \*\*How do you perform a type assertion in TypeScript?\*\*

- A.

```
```typescript  
let value: any = 'hello';  
let length: number = (value as string).length;  
...
```

- B.

```
```typescript  
let value: any = 'hello';  
let length: number = (<string>value).length;  
...
```

- C.

```
```typescript  
let value: any = 'hello';  
let length: number = value.length as string;  
...
```

- D.

```
```typescript  
let value: any = 'hello';  
let length: number = value.length;
```

...

267. **\*\*What is the difference between type assertions and type casting in TypeScript?\*\***

- A. Type assertions are used to inform the compiler about the type of a variable, while type casting converts a variable to a specific type.
- B. Type assertions convert a variable to a specific type, while type casting informs the compiler about the type of a variable.
- C. There is no difference; both terms are used interchangeably.
- D. Type casting is only used in TypeScript, while type assertions are used in JavaScript.

268. **\*\*When should you use type assertions in TypeScript?\*\***

- A. When you are confident about the type of a variable and want to override TypeScript's inferred type.
- B. When you want to check the runtime type of a variable.
- C. When you need to create a new type.
- D. When you want to prevent type checking.

269. **\*\*Can type assertions change the runtime type of a variable?\*\***

- A. No, type assertions only affect compile-time type checking.
- B. Yes, type assertions can change the runtime type.
- C. Type assertions can only affect the value of the variable.
- D. Yes, type assertions modify the variable at runtime.

270. **\*\*How do you use `unknown` in type assertions?\*\***

- A. You cannot use `unknown` in type assertions.
- B.

```typescript

```
let value: unknown = 'hello';

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

...

```

- C.

```
```typescript

let value: unknown = 'hello';

let length: number = <string>value.length;

...

```

- D.

```
```typescript

let value: unknown = 'hello';

let length: number = value.length as string;

...

```

### Topic: modules (continued)

271. **\*\*How do you import a module in TypeScript?\*\***

- A.

```
```typescript

import { MyClass } from './myModule';

...

```

- B.

```
```typescript

import MyClass from './myModule';

...

```

- C.

```
```typescript
```

```
import * as MyClass from './myModule';
```

```
...
```

- D.

```
```typescript
```

```
import { MyClass } from 'myModule';
```

```
...
```

272. **\*\*How do you export a class from a module in TypeScript?\*\***

- A.

```
```typescript
```

```
export class MyClass {}
```

```
...
```

- B.

```
```typescript
```

```
class MyClass {}
```

```
export { MyClass };
```

```
...
```

- C.

```
```typescript
```

```
class MyClass {}
```

```
export default MyClass;
```

```
...
```

- D.

```
```typescript
```

```
export default class MyClass {}
```



...

273. **What is the difference between `export` and `export default` in TypeScript?**

- A. `export` allows exporting multiple values, while `export default` allows exporting a single value.
- B. `export` is used for default exports, while `export default` is used for named exports.
- C. `export` is used to import modules, while `export default` is used to export modules.
- D. There is no difference between `export` and `export default`.

274. **How do you re-export a module in TypeScript?**

- A.

```
``typescript
```

```
export { MyClass } from './myModule';
```

...

- B.

```
``typescript
```

```
import { MyClass } from './myModule';
```

```
export { MyClass };
```

...

- C.

```
``typescript
```

```
export * from './myModule';
```

...

- D.

```
``typescript
```

```
import * as MyModule from './myModule';
```

```
export { MyModule };
```

...

275. **What does the `import * as` syntax do in TypeScript?**

- A. Imports all exported members of a module as a single object.
- B. Imports a single default export from a module.
- C. Imports a module without any members.
- D. Imports a module with a specific member.

### Topic: JSON objects (continued)

276. **How can you parse a JSON string into a JavaScript object in TypeScript?**

- A.

```
```typescript
```

```
const jsonString = '{"name":"John","age":30}';
```

```
const obj = JSON.parse(jsonString);
```

```
```
```

- B.

```
```typescript
```

```
const jsonString = '{"name":"John","age":30}';
```

```
const obj = JSON.stringify(jsonString);
```

```
```
```

- C.

```
```typescript
```

```
const obj = JSON.parse('{"name":"John","age":30}');
```

```
```
```

- D.

```
```typescript
```

```
const obj = JSON.stringify({ name: "John", age: 30 });
```

```
```
```

277. **\*\*What TypeScript type is most commonly used to represent a parsed JSON object?\*\***

- A. ``any``
- B. ``object``
- C. ``unknown``
- D. ``string``

278. **\*\*How do you ensure type safety when working with JSON objects in TypeScript?\*\***

- A. By defining interfaces or types that match the structure of the JSON data.
- B. By using ``any`` type for all JSON objects.
- C. By converting JSON data into a string before processing.
- D. By not using TypeScript for JSON data processing.

279. **\*\*How do you type-assert a JSON object to a specific interface in TypeScript?\*\***

- A.

```
```typescript
```

```
interface User {
```

```
  name: string;
```

```
  age: number;
```

```
}
```

```
const jsonString = '{"name":"John","age":30}';
```

```
const user = JSON.parse(jsonString) as User;
```

```
```
```

- B.

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

const user = JSON.parse('{"name":"John","age":30}');
...

```

- C.

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

const user = JSON.stringify({ name: "John", age: 30 }) as User;
...

```

- D.

```
``typescript
const jsonString: string = '{"name":"John","age":30}';

const user = JSON.parse(jsonString);
...

```

280. **\*\*How can you handle optional properties in a TypeScript interface when dealing with JSON data?\*\***

- A.

```
``typescript
interface User {

```

```
    name: string;  
    age?: number;  
}
```

...

- B.

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

...

- C.

```
```typescript  
interface User {  
    name: string;  
    age: null;  
}
```

...

- D.

```
```typescript  
interface User {  
    name?: string;  
    age?: number;  
}
```

...

### Topic: type\_errors (continued)

281. **\*\*What is a type error in TypeScript?\*\***

- A. An error that occurs when a value does not match the expected type.
- B. An error that occurs when a value is null.
- C. An error that occurs during runtime.
- D. An error that occurs when a variable is uninitialized.

282. **\*\*How does TypeScript report type errors?\*\***

- A. By using the compiler and showing errors in the IDE or terminal.
- B. By logging errors in the browser console.
- C. By throwing runtime exceptions.
- D. By stopping the execution of the program.

283. **\*\*What should you do when encountering a type error related to function parameters?\*\***

- A. Check the function signature and ensure that the arguments passed match the expected types.
- B. Ignore the error if the function works correctly.
- C. Change the function return type to `any`.
- D. Remove the type annotations from the function parameters.

284. **\*\*How can you fix a type error related to object properties in TypeScript?\*\***

- A. Ensure that the object properties are accessed using the correct keys and that they match the expected type.
- B. Use the `any` type for the object.
- C. Remove the property from the object.
- D. Change the object's type to `string`.

285. **\*\*What is the best practice for handling type errors when using external libraries in TypeScript?\*\***

- A. Use type definitions (`@types`) for the external library.
- B. Ignore type errors and rely on runtime checks.
- C. Rewrite the external library in TypeScript.
- D. Use `any` type for all external library imports.

### Topic: strong\_typing (continued)

286. **\*\*What is the advantage of strong typing in TypeScript?\*\***

- A. It helps catch errors at compile time and improves code reliability.
- B. It makes the code run faster.
- C. It reduces the need for code comments.
- D. It allows dynamic typing at runtime.

287. **\*\*How does TypeScript enforce strong typing?\*\***

- A. By providing type annotations and checking types during compilation.
- B. By using runtime type checks.
- C. By converting TypeScript code into JavaScript with dynamic types.
- D. By using type inference only.

288. **\*\*Which of the following is a benefit of using strong typing in TypeScript?\*\***

- A. Increased code maintainability and better refactoring capabilities.
- B. Reduced code size.
- C. Faster execution of code.
- D. Simplified syntax.

289. **\*\*How can you ensure that your TypeScript code adheres to strong typing principles?\*\***

- A. By consistently using type annotations and interfaces.
- B. By avoiding any type assertions.
- C. By only using basic types.
- D. By writing code without any type definitions.

290. **\*\*Which TypeScript feature helps to enforce strong typing across different modules?\*\***

- A. Modules and type definitions
- B. Runtime type checking
- C. Type assertions
- D. Dynamic typing

### Topic: const\_let (continued)

291. **\*\*What is the difference between `const` and `let` in TypeScript?\*\***

- A. `const` creates a read-only reference to a value, while `let` allows reassignment.
- B. `let` creates a read-only reference, while `const` allows reassignment.
- C. `const` and `let` are equivalent in behavior.
- D. `const` is used for function declarations, while `let` is used for variable declarations.

292. **\*\*Can you reassign a value to a variable declared with `const` in TypeScript?\*\***

- A. No, `const` creates a constant reference.
- B. Yes, but only if the variable is an object.
- C. Yes, but only if the variable is a number.
- D. Yes, `const` variables can be reassigned if they are not primitive types.



293. **\*\*How do you declare a block-scoped variable in TypeScript?\*\***

- A. By using ``let`` or ``const``.
- B. By using ``var``.
- C. By using ``function``.
- D. By using ``class``.

294. **\*\*What will happen if you try to reassign a ``const`` variable in TypeScript?**

- A. TypeScript will throw a compilation error.
- B. The reassignment will be ignored at runtime.
- C. The variable will be automatically converted to ``let``.
- D. TypeScript will automatically fix the reassignment.

295. **\*\*Which of the following correctly declares a constant array in TypeScript?**

- A.

```
``typescript
```

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

```
...
```

- B.

```
``typescript
```

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

```
...
```

- C.

```
``typescript
```

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

```
...
```

- D.

```
```typescript
```

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

```
```
```

### Topic: modules (continued)

296. **\*\*How do you import a specific member from a module in TypeScript?\*\***

- A.

```
```typescript
```

```
import { member } from './module';
```

```
```
```

- B.

```
```typescript
```

```
import member from './module';
```

```
```
```

- C.

```
```typescript
```

```
import * as member from './module';
```

```
```
```

- D.

```
```typescript
```

```
import { member } from 'module';
```

```
```
```

297. **\*\*What is the purpose of `export \* from` in TypeScript?\*\***

- A. To re-export all members from a module.
- B. To import all members from a module.
- C. To rename all exported members from a module.
- D. To delete all members from a module.

298. **\*\*How can you use TypeScript to work with CommonJS modules?\*\***

- A. By using ``import`` and ``export`` syntax with module loaders.
- B. By using ``require`` and ``module.exports`` syntax.
- C. By using only ``require`` syntax.
- D. By using ``export default`` syntax only.

299. **\*\*How do you declare a module in TypeScript?\*\***

- A.

```
``typescript
module MyModule {
  export class MyClass {}
}
...

```

- B.

```
``typescript
export module MyModule

{
  export class MyClass {}
}
...

```

- C.

```
```typescript
namespace MyModule {
  export class MyClass {}
}
...

```

- D.

```
```typescript
class MyModule {
  export class MyClass {}
}
...

```

300. **\*\*What is the syntax to import the default export from a module in TypeScript?\*\***

- A.

```
```typescript
import MyClass from './myModule';
...

```

- B.

```
```typescript
import { MyClass } from './myModule';
...

```

- C.

```
```typescript
import * as MyClass from './myModule';
...

```

- D.

```
```typescript
```

```
import MyClass = require('./myModule');
```

```
```
```

### Topic: union\_literals (continued)

301. **\*\*How do you define a union type with literal types in TypeScript?\*\***

- A.

```
```typescript
```

```
type Status = 'success' | 'error' | 'pending';
```

```
```
```

- B.

```
```typescript
```

```
type Status = string | 'success' | 'error';
```

```
```
```

- C.

```
```typescript
```

```
type Status = 'success' | 'failure' | boolean;
```

```
```
```

- D.

```
```typescript
```

```
type Status = 'success' | 'error' | number;
```

```
```
```

302. **\*\*What will the following TypeScript code output?\*\***

```
```typescript
```

```
type Color = 'red' | 'green' | 'blue';
```

```
const color: Color = 'yellow';
```

```
```
```

- A. TypeScript will throw a compilation error.
- B. The code will run successfully, and `color` will be 'yellow'.
- C. `color` will be converted to 'red'.
- D. The code will run with a warning, and `color` will be 'yellow'.

303. **\*\*Which of the following is a valid use of union types in TypeScript?\*\***

- A.

```
```typescript
```

```
function handleInput(input: string | number) {
```

```
  if (typeof input === 'string') {
```

```
    console.log(input.toUpperCase());
```

```
  } else {
```

```
    console.log(input.toFixed(2));
```

```
  }
```

```
}
```

```
```
```

- B.

```
```typescript
```

```
function handleInput(input: string | boolean) {
```

```
  if (typeof input === 'string') {
```

```
    console.log(input.toUpperCase());
```

```
  } else {
```

```
    console.log(input.toFixed(2));  
  }  
}  
...
```

- C.

```
```typescript  
function handleInput(input: number | boolean) {  
  if (typeof input === 'number') {  
    console.log(input.toFixed(2));  
  } else {  
    console.log(input.toUpperCase());  
  }  
}  
}  
...
```

- D.

```
```typescript  
function handleInput(input: string | number) {  
  console.log(input.toUpperCase());  
}  
}  
...
```

304. **\*\*How do you define a union type that includes both object types and literal types in TypeScript?\***

- A.

```
```typescript  
type Result = { success: true } | { error: string } | 'pending';  
...
```

- B.

```
```typescript  
type Result = { success: true } | { error: boolean } | 'pending';  
...`
```

- C.

```
```typescript  
type Result = { success: string } | { error: string } | 'pending';  
...`
```

- D.

```
```typescript  
type Result = { success: true } | { error: string } | number;  
...`
```

305. **\*\*What will happen if you attempt to assign a value not included in a union type to a variable of that type?\*\***

- A. TypeScript will throw a compilation error.
- B. The value will be automatically converted to one of the union types.
- C. The code will run successfully, but a warning will be logged.
- D. The variable will be set to `undefined`.

### Topic: async (continued)

306. **\*\*How do you define an asynchronous function in TypeScript?\*\***

- A.

```
```typescript  
async function fetchData(): Promise<string> {
```



```
    return 'data';  
  }  
  ...
```

- B.

```
```typescript  
function fetchData(): Promise<string> {  
  return new Promise(resolve => resolve('data'));  
}  
...
```

- C.

```
```typescript  
function fetchData(): string {  
  return 'data';  
}  
...
```

- D.

```
```typescript  
async function fetchData(): string {  
  return 'data';  
}  
...
```

307. **\*\*What will the following TypeScript code do?\*\***

```
```typescript  
async function getUser() {  
  return { name: 'John', age: 30 };  
}
```

```
}
```

```
...
```

- A. It will return a promise that resolves to `{ name: 'John', age: 30 }`.
- B. It will return `{ name: 'John', age: 30 }` directly.
- C. It will return a promise that rejects with an error.
- D. It will return an empty promise.

308. **\*\*How do you handle errors in an asynchronous function in TypeScript?\*\***

- A. By using `try` and `catch` blocks within the `async` function.
- B. By checking the error status of the promise.
- C. By using `.catch()` method on the promise.
- D. By handling errors outside the `async` function.

309. **\*\*Which of the following is the correct way to call an `async` function and handle its result?\*\***

- A.

```
```typescript
```

```
async function fetchData() {
```

```
  return 'data';
```

```
}
```

```
fetchData().then(result => console.log(result));
```

```
...
```

- B.

```
```typescript
```

```
async function fetchData() {
```

```
  return 'data';
```

```
}
```

```
console.log(fetchData());
```

```
...
```

- C.

```
```typescript
```

```
function fetchData() {
```

```
    return new Promise(resolve => resolve('data'));
```

```
}
```

```
fetchData().then(result => console.log(result));
```

```
...
```

- D.

```
```typescript
```

```
function fetchData() {
```

```
    return 'data';
```

```
}
```

```
fetchData().then(result => console.log(result));
```

```
...
```

310. **\*\*How do you use `await` inside an `async` function in TypeScript?\*\***

- A. By using `await` to pause the execution until the promise is resolved.
- B. By using `await` to convert a promise to a synchronous result.
- C. By using `await` to handle errors in promises.
- D. By using `await` to convert a synchronous function to asynchronous.

### Topic: tuples (continued)

311. **\*\*How do you define a tuple in TypeScript?\*\***

- A.

```
```typescript
```

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

```
```
```

- B.

```
```typescript
```

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

```
```
```

- C.

```
```typescript
```

```
let tuple: (string, number) = ['hello', 10];
```

```
```
```

- D.

```
```typescript
```

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

```
```
```

312. **\*\*What will happen if you try to assign a value of the wrong type to a tuple element in TypeScript?\*\***

- A. TypeScript will throw a compilation error.
- B. The value will be automatically converted to the correct type.
- C. The code will run, and the tuple will be filled with the default values.
- D. The value will be ignored.

313. **\*\*How do you access tuple elements by index in TypeScript?\*\***

- A.

```
```typescript
```

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

```
let firstElement = tuple[0];
```

```
let secondElement = tuple[1];
```

```
...
```

- B.

```
```typescript
```

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

```
let firstElement = tuple['0'];
```

```
let secondElement = tuple['1'];
```

```
...
```

- C.

```
```typescript
```

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

```
let firstElement = tuple.get(0);
```

```
let secondElement = tuple.get(1);
```

```
...
```

- D.

```
```typescript
```

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

```
let firstElement = tuple.first();
```

```
let secondElement = tuple.second();
```

```
...
```

314. **\*\*How can you use rest elements in tuples in TypeScript?\*\***

- A.

```
```typescript
```

```
let tuple: [string, ...number[]] = ['hello', 1, 2, 3];
```

```
...
```

- B.

```
```typescript
```

```
let tuple: [string, number, ...string[]] = ['hello', 1, 'world'];
```

```
...
```

- C.

```
```typescript
```

```
let tuple: [...string[], number] = ['hello', 'world', 1];
```

```
...
```

- D.

```
```typescript
```

```
let tuple: [string, number, ...boolean[]] = ['hello', 1, true];
```

```
...
```

315. **\*\*How do you specify a tuple with optional elements in TypeScript?\*\***

- A.

```
```typescript
```

```
let tuple: [string, number?] = ['hello'];
```

```
...
```

- B.

```
```typescript
```

```
let
```

```
tuple: [string, number] = ['hello', undefined];
```

```
...
```

- C.

```
```typescript
```

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

```
```
```

- D.

```
```typescript
```

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

```
```
```

Certainly! Continuing from where we left off:

### Topic: JSON Objects (continued)

316. **\*\*Which TypeScript type should you use to represent a JSON object with mixed key types?\*\***

- A. `Record<string, any>`
- B. `{ [key: string]: any }`
- C. `{ [key: number]: any }`
- D. `object`

317. **\*\*How do you ensure that a JSON object adheres to a specific structure in TypeScript?\*\***

- A. By using interfaces or types to define the structure.
- B. By directly using the JSON object without type checking.
- C. By converting the JSON object to a string and validating the string.
- D. By using the `any` type.

318. **\*\*How would you type a JSON object that contains an array of strings?\*\***

- A.

```
```typescript
```

```
type MyObject = { names: string[] };
```

```
```
```

- B.

```
```typescript
```

```
type MyObject = { names: Array<string> };
```

```
```
```

- C.

```
```typescript
```

```
type MyObject = { names: (string)[] };
```

```
```
```

- D.

```
```typescript
```

```
type MyObject = { names: [string] };
```

```
```
```

319. **\*\*What will be the result of the following TypeScript code?\*\***

```
```typescript
```

```
const data: any = { id: 1, name: "Alice" };
```

```
const user: { id: number, name: string } = data;
```

```
```
```

- A. `user` will be correctly typed.

- B. TypeScript will throw an error because `data` is of type `any`.

- C. `user` will be of type `any`.

- D. The code will result in a runtime error.



320. **\*\*How do you define a JSON object with optional properties in TypeScript?\*\***

- A.

```
````typescript
type MyObject = { id: number; name?: string };
...

```

- B.

```
````typescript
type MyObject = { id?: number; name?: string };
...

```

- C.

```
````typescript
type MyObject = { id: number; name: string | undefined };
...

```

- D.

```
````typescript
type MyObject = { id: number; name: string | null };
...

```

### Topic: Type Errors (continued)

321. **\*\*How does TypeScript handle type errors during compilation?\*\***

- A. By providing detailed error messages in the console or IDE.
- B. By ignoring type errors and compiling the code anyway.
- C. By automatically converting types to the correct ones.
- D. By halting the compilation process.

322. **\*\*Which TypeScript feature helps in identifying and fixing type errors early in the development process?\*\***

- A. Type inference
- B. Type assertions
- C. Type guards
- D. Type annotations

323. **\*\*How can you use type guards to handle type errors in TypeScript?\*\***

- A. By using conditional checks to narrow down types.
- B. By casting types explicitly.
- C. By using `any` type to bypass type checking.
- D. By avoiding type annotations.

324. **\*\*What is a common cause of type errors when working with third-party libraries in TypeScript?\*\***

- A. Missing or incorrect type definitions for the library.
- B. Incorrect TypeScript version.
- C. Using TypeScript without a module loader.
- D. Incorrect usage of `any` type.

325. **\*\*How do you resolve type errors related to mismatched function signatures in TypeScript?\*\***

- A. By adjusting the function parameters or return types to match the expected types.
- B. By using the `any` type for all function signatures.
- C. By removing type annotations from the function.
- D. By overriding the function with a different implementation.

### ### Topic: Strong Typing (continued)

326. **\*\*What is the main advantage of strong typing in TypeScript?\*\***

- A. Reduces the likelihood of runtime errors by catching errors at compile time.
- B. Improves the runtime performance of the application.
- C. Simplifies the syntax of the code.
- D. Reduces the need for unit testing.

327. **\*\*How does TypeScript enforce strong typing when dealing with complex data structures?\*\***

- A. By using explicit type annotations and interfaces.
- B. By automatically inferring types based on usage.
- C. By allowing any type of data structure.
- D. By using type assertions only.

328. **\*\*How can you use TypeScript's type system to improve code readability and maintainability?\*\***

- A. By using descriptive type names and clear interfaces.
- B. By avoiding type annotations and using `any`.
- C. By writing less code with fewer type definitions.
- D. By using complex and nested types excessively.

329. **\*\*Which TypeScript feature enhances strong typing by allowing the creation of custom types?\*\***

- A. Type aliases
- B. Type assertions
- C. Type inference
- D. Type guards

330. **\*\*How does TypeScript's strong typing benefit team collaboration on a project?\*\***

- A. By providing clear type definitions and interfaces that help understand and use the code correctly.
- B. By allowing team members to work with any type without constraints.
- C. By reducing the need for documentation.
- D. By automatically generating code based on types.

### Topic: Const and Let (continued)

331. **\*\*Which statement is true about `const` in TypeScript?\*\***

- A. `const` declares a read-only variable that cannot be reassigned.
- B. `const` allows reassignments but not changes to the value.
- C. `const` is similar to `var` but with block scope.
- D. `const` variables must be initialized at the time of declaration.

332. **\*\*What is the correct way to use `let` for a variable that will be reassigned multiple times?\*\***

- A.

```
``typescript
```

```
let counter = 0;
```

```
counter = 1;
```

```
counter = 2;
```

```
...
```

- B.

```
``typescript
```

```
const counter = 0;
```

```
counter = 1;
```

```
...
```

- C.

```
``typescript
var counter = 0;

counter = 1;

...

```

- D.

```
``typescript
let counter: number;

counter = 1;

...

```

333. **\*\*Which of the following correctly demonstrates block scope with `let` in TypeScript?\*\***

- A.

```
``typescript
if (true) {

  let x = 10;

}

console.log(x); // Error: x is not defined

...

```

- B.

```
``typescript
if (true) {

  var x = 10;

}

console.log(x); // Output: 10

...

```

- C.

```
```typescript
```

```
if (true) {
```

```
    const x = 10;
```

```
}
```

```
console.log(x); // Error: x is not defined
```

```
```
```

- D.

```
```typescript
```

```
if (true) {
```

```
    let x = 10;
```

```
}
```

```
console.log(x); // Output: 10
```

```
```
```

334. **\*\*How do `const` and `let` handle reassignments in TypeScript?\*\***

- A. `const` does not allow reassignment, while `let` allows multiple reassignments.
- B. Both `const` and `let` allow reassignment.
- C. `let` does not allow reassignment, while `const` allows multiple reassignments.
- D. Both `const` and `let` do not allow reassignment.

335. **\*\*When should you use `const` instead of `let` in TypeScript?\*\***

- A. When the variable's value should not be reassigned.
- B. When the variable's value will change multiple times.
- C. When the variable needs to be globally accessible.
- D. When the variable is intended for asynchronous operations.

### ### Topic: Modules (continued)

336. **\*\*How can you export multiple items from a TypeScript module?\*\***

- A.

```
``typescript
export { item1, item2 };
...

```

- B.

```
``typescript
export item1, item2;
...

```

- C.

```
``typescript
export default { item1, item2 };
...

```

- D.

```
``typescript
export * from './module';
...

```

337. **\*\*What is the default export syntax in TypeScript?\*\***

- A.

```
``typescript
export default function myFunction() {}
...

```

- B.

```
```typescript  
  
export function myFunction() {}  
  
```
```

- C.

```
```typescript  
  
default export function myFunction() {}  
  
```
```

- D.

```
```typescript  
  
export default myFunction;  
  
```
```

338. **\*\*How do you import everything from a module as a single object in TypeScript?\*\***

- A.

```
```typescript  
  
import * as moduleName from './module';  
  
```
```

- B.

```
```typescript  
  
import moduleName from './module';  
  
```
```

- C.

```
```typescript  
  
import { * } from './module';  
  
```
```



- D.

```
```typescript
```

```
import { moduleName } from './module';
```

```
```
```

339. **What is the purpose of `export =` syntax in TypeScript modules?**

- A. To export a single object, function, or class from a module.
- B. To export multiple named items from a module.
- C. To import items from a CommonJS module.
- D. To define a default export for a module.

340. **How do you handle module resolution when importing a module in TypeScript?**

- A. By configuring `tsconfig.json` with `moduleResolution` settings.
- B. By using relative paths for all imports.
- C. By manually resolving paths in the code.
- D. By avoiding module imports altogether.

### Topic: Native ECMAScript Modules (continued)

341. **How do you import a named export from a native ECMAScript module in TypeScript?**

- A.

```
```typescript
```

```
import { namedExport } from './module';
```

```
```
```

- B.

```
```typescript
```

```
import namedExport from './module';
```

```
```
```

- C.

```
```typescript
```

```
import * as namedExport from './module';
```

```
```
```

- D.

```
```typescript
```

```
import { namedExport as alias } from './module';
```

```
```
```

342. \*\*What is the correct syntax for importing a default export from a native ECMAScript module in TypeScript?\*\*

- A.

```
```typescript
```

```
import defaultExport from './module';
```

```
```
```

- B.

```
```typescript
```

```
import { defaultExport } from './module';
```

```
```
```

- C.

```
```typescript
```

```
import * as defaultExport from './module';
```

...

- D.

```
```typescript
```

```
import defaultExport = require('./module');
```

...

343. \*\*How do you combine named and default exports in a single import statement in TypeScript?\*\*

- A.

```
```typescript
```

```
import defaultExport, { namedExport } from './module';
```

...

- B.

```
```typescript
```

```
import { defaultExport, namedExport } from './module';
```

...

- C.

```
```typescript
```

```
import defaultExport from './module';
```

```
import { namedExport } from './module';
```

...

- D.

```
```typescript
```

```
import { defaultExport as namedExport } from './module';
```

...

344. \*\*What will be the result of the following TypeScript import statement?\*\*

```
```typescript
```

```
import { a, b } from './module';
```

```
```
```

- A. Imports named exports `a` and `b` from `./module`.
- B. Imports `a` and `b` as default exports from `./module`.
- C. Imports the entire module as `a` and `b`.
- D. Throws a syntax error.

345. **\*\*What does `export \* from './module';` do in TypeScript?\*\***

- A. Re-exports all named exports from `./module`.
- B. Exports a default export from `./module`.
- C. Exports only the default export from `./module`.
- D. Imports all named exports from `./module`.

### Topic: Importing Inquirer ECMAScript Module (continued)

346. **\*\*How do you import the `inquirer` module in TypeScript?\*\***

- A.

```
```typescript
```

```
import inquirer from 'inquirer';
```

```
```
```

- B.

```
```typescript
```

```
import * as inquirer from 'inquirer';
```

```
```
```

- C.

```
```typescript
```

```
const inquirer = require('inquirer');
```

```
...
```

- D.

```
```typescript
```

```
import { inquirer } from 'inquirer';
```

```
...
```

347. **What is the purpose of the `inquirer` module in TypeScript?**

- A. To prompt users for input via the command line.
- B. To handle HTTP requests.
- C. To manage application state.
- D. To provide a UI for web applications.

348. **Which method from the `inquirer` module is used to prompt the user with a question?**

- A. `inquirer.prompt()`
- B. `inquirer.ask()`
- C. `inquirer.query()`
- D. `inquirer.request()`

349. **How do you define a prompt with a list of choices using `inquirer` in TypeScript?**

- A.

```
```typescript
```

```
inquirer.prompt({
```

```
  type: 'list',
```

```
  name: 'choice',
```

```
    message: 'Choose an option:',  
    choices: ['Option1', 'Option2']  
  });  
  ...
```

- B.

```
```typescript  
inquirer.prompt({  
  type: 'checkbox',  
  name: 'choice',  
  message: 'Choose options:',  
  choices: ['Option1', 'Option2']  
});  
...
```

- C.

```
```typescript  
inquirer.ask({  
  type: 'list',  
  name: 'choice',  
  message: 'Choose an option:',  
  choices: ['Option1', 'Option2']  
});  
...
```

- D.

```
```typescript  
inquirer.request({  
  type: 'list',
```

```
name: 'choice',  
message: 'Choose an option:',  
choices: ['Option1', 'Option2']  
});  
...
```

350. **What does the `name` property in an `inquirer` prompt configuration represent?**

- A. The key under which the user's input will be stored.
- B. The label displayed to the user.
- C. The type of the input prompt.
- D. The default value of the prompt.

### Topic: Chalk (continued)

351. **How do you use `chalk` to style console output in TypeScript?**

- A.

```
``typescript  
import chalk from 'chalk';  
console.log(chalk.green('Success!'));  
...
```

- B.

```
``typescript  
import * as chalk from 'chalk';  
console.log(chalk.bold('Warning!'));  
...
```

- C.

```
```typescript
```

```
const chalk = require('chalk');  
  
console.log(chalk.red('Error!'));  
...
```

- D.

```
```typescript
```

```
import { chalk } from 'chalk';  
  
console.log(chalk.blue('Info!'));  
...
```

352. **\*\*Which method in `chalk` is used to apply multiple styles to a string?\*\***

- A. `chalk.styles()`
- B. `chalk.combine()`
- C. `chalk`
- D. `chalk.compose()`

353. **\*\*How do you chain multiple styles using `chalk` in TypeScript?\*\***

- A.

```
```typescript
```

```
console.log(chalk.red.bold('Error!'));  
...
```

- B.

```
```typescript
```

```
console.log(chalk.combine(chalk.red(), chalk.bold('Error!')));  
...
```

- C.



```
```typescript
console.log(chalk.red().bold('Error!'));
...

```

- D.

```
```typescript
console.log(chalk.red('Error!').bold());
...

```

354. **\*\*What is the output of the following TypeScript code using `chalk`?\*\***

```
```typescript
import chalk from 'chalk';

console.log(chalk.bgYellow.black('Warning!'));
...

```

- A. The text 'Warning!' with a yellow background and black text.
- B. The text 'Warning!' with a black background and yellow text.
- C. The text 'Warning!' with a yellow foreground and black background.
- D. The text 'Warning!' with a yellow border and black text.

355. **\*\*How do you use `chalk` to style text with a specific color and background?\*\***

- A.

```
```typescript
console.log(chalk.bgBlue.white('Hello World!'));
...

```

- B.

```
```typescript
console.log(chalk.color('blue').background('white')('Hello World!'));

```

...

- C.

```
```typescript
```

```
console.log(chalk.text('Hello World!').background('blue'));
```

...

- D.

```
```typescript
```

```
console.log(chalk.color('white').background('blue')('Hello World!'));
```

...

### ### Topic: Intersection Types (continued)

356. **\*\*How do you define an intersection type in TypeScript?\*\***

- A.

```
```typescript
```

```
type Combined = TypeA & TypeB;
```

...

- B.

```
```typescript
```

```
type Combined = TypeA | TypeB;
```

...

- C.

```
```typescript
```

```
type Combined = TypeA + TypeB;
```

...

- D.

```
```typescript
```

```
type Combined = TypeA, TypeB;
```

```
```
```

357. **\*\*What will be the type of `result` in the following TypeScript code?\*\***

```
```typescript
```

```
type A = { a: number };
```

```
type B = { b: string };
```

```
type C = A & B;
```

```
const result: C = { a: 1, b: 'test' };
```

```
```
```

- A. `{ a: number; b: string }`

- B. `{ a: number }`

- C. `{ b: string }`

- D. `A | B`

358. **\*\*What does the `&` operator represent in TypeScript intersection types?\*\***

- A. A combination of multiple types.

- B. A union of multiple types.

- C. A reference to a single type.

- D. An alias for one type.

359. **\*\*Which of the following is an example of**

**using intersection types to combine interfaces?\*\***

- A.

```
```typescript
```

```
interface A { a: number; }
```

```
interface B { b: string; }
```

```
type C = A & B;
```

```
...
```

- B.

```
```typescript
```

```
interface A { a: number; }
```

```
interface B { b: string; }
```

```
type C = A | B;
```

```
...
```

- C.

```
```typescript
```

```
interface A { a: number; }
```

```
type B = { b: string; }
```

```
type C = A & B;
```

```
...
```

- D.

```
```typescript
```

```
interface A { a: number; }
```

```
type B = A | { b: string; }
```

```
...
```

360. **\*\*How does TypeScript handle type checking for intersection types with optional properties?\*\***

- A. All properties from intersected types are required.
- B. Optional properties are treated as required.

- C. Only properties present in the intersected types are required.
- D. Optional properties remain optional in the resulting type.

### Topic: Any, Unknown, and Never Types (continued)

361. **\*\*What is the difference between ``any`` and ``unknown`` in TypeScript?\*\***

- A. ``any`` allows any operations without type checking, while ``unknown`` requires type checking before use.
- B. ``any`` and ``unknown`` are identical in behavior.
- C. ``unknown`` allows any operations without type checking, while ``any`` requires type checking before use.
- D. ``any`` is more restrictive than ``unknown``.

362. **\*\*Which type should you use if you want to ensure that a variable is never assigned a value?\*\***

- A. ``never``
- B. ``unknown``
- C. ``any``
- D. ``void``

363. **\*\*How do you handle a variable with type ``unknown`` in TypeScript?\*\***

- A. By using type assertions or type checks to determine the actual type.
- B. By assigning it directly without any type checks.
- C. By using it as ``any`` to bypass type checking.
- D. By avoiding its use altogether.

364. **\*\*What is the purpose of the ``never`` type in TypeScript?\*\***

- A. To indicate a value that should never occur.
- B. To represent any possible value.

- C. To indicate an unknown type.
- D. To represent an optional value.

365. **\*\*What will be the result of the following TypeScript code?\*\***

```
``typescript
function throwError(): never {
    throw new Error('An error occurred');
}
``
```

- A. The function never returns a value.
- B. The function returns `void`.
- C. The function returns `any`.
- D. The function returns a value of type `Error`.

### Topic: Explicit Casting (continued)

366. **\*\*How do you perform explicit type casting in TypeScript?\*\***

- A. Using the `as` keyword or angle-bracket syntax.
- B. Using the `cast` keyword.
- C. Using type assertions with `assert`.
- D. Using the `convert` keyword.

### Topic: Const Enum (continued)

380. **\*\*What is the difference between `enum` and `const enum` in TypeScript?\*\***

- A. ``const enum`` values are inlined into the JavaScript output at compile time, while ``enum`` values are accessed via an object at runtime.

- B. ``const enum`` provides additional type safety compared to ``enum``.

- C. ``const enum`` supports runtime evaluation of values, while ``enum`` does not.

- D. ``const enum`` allows for more flexible value assignments compared to ``enum``.

### ### Topic: Interfaces and Classes (continued)

381. **\*\*How do you define an interface with optional properties in TypeScript?\*\***

- A.

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

- B.

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

- C.

```
```typescript
interface Person {
```

```
name: string;
```

```
age: number;
```

```
}
```

```
...
```

- D.

```
```typescript
```

```
interface Person {
```

```
name: string;
```

```
age: number;
```

```
}
```

```
...
```

382. **\*\*What is the syntax for implementing an interface in a TypeScript class?\*\***

- A.

```
```typescript
```

```
class Employee implements Person {
```

```
name: string;
```

```
age: number;
```

```
}
```

```
...
```

- B.

```
```typescript
```

```
class Employee extends Person {
```

```
name: string;
```

```
age: number;
```

```
}
```



...

- C.

```typescript

class Employee uses Person {

    name: string;

    age: number;

}

...

- D.

```typescript

class Employee applies Person {

    name: string;

    age: number;

}

...

383. **\*\*How do you extend an interface in TypeScript?\*\***

- A.

```typescript

interface Employee extends Person {

    employeeId: number;

}

...

- B.

```typescript

interface Employee extends Person {

```
name: string;

employeeId: number;

}
```

- C.

```
```typescript

interface Employee uses Person {

    employeeId: number;

}

```
```

- D.

```
```typescript

interface Employee applies Person {

    employeeId: number;

}

```
```

384. **What is the correct way to define a method in an interface?**

- A.

```
```typescript

interface Greeter {

    greet(name: string): string;

}

```
```

- B.

```
```typescript
```

```
interface Greeter {  
    greet(name: string): void;  
}  
...
```

- C.

```
```typescript
```

```
interface Greeter {  
    greet(name: string);  
}  
...
```

- D.

```
```typescript
```

```
interface Greeter {  
    greet(name: string): string | void;  
}  
...
```

385. **\*\*How do you implement a method defined in an interface in a TypeScript class?\*\***

- A.

```
```typescript
```

```
class FriendlyGreeter implements Greeter {  
    greet(name: string): string {  
        return `Hello, ${name}!`;  
    }  
}  
...
```

- B.

```
```typescript
```

```
class FriendlyGreeter implements Greeter {
```

```
  greet(name: string): void {
```

```
    console.log(`Hello, ${name}!`);
```

```
  }
```

```
}
```

```
```
```

- C.

```
```typescript
```

```
class FriendlyGreeter uses Greeter {
```

```
  greet(name: string) {
```

```
    return `Hello, ${name}!`;
```

```
  }
```

```
}
```

```
```
```

- D.

```
```typescript
```

```
class FriendlyGreeter applies Greeter {
```

```
  greet(name: string) {
```

```
    console.log(`Hello, ${name}!`);
```

```
  }
```

```
}
```

```
```
```

### Topic: Type Inference (continued)

386. **\*\*What is TypeScript's type inference system used for?\*\***

- A. To automatically determine the type of variables based on their values and usage.
- B. To enforce strict type checking throughout the codebase.
- C. To define explicit types for variables and functions.
- D. To convert JavaScript types to TypeScript types.

387. **\*\*How does TypeScript infer the type of a variable when you assign a value to it?\*\***

- A. Based on the value assigned to the variable.
- B. By using type annotations provided by the developer.
- C. Through explicit type declarations.
- D. By analyzing the function's return type.

388. **\*\*What is the default type of a variable if no type is explicitly assigned in TypeScript?\*\***

- A. ``any``
- B. ``unknown``
- C. ``void``
- D. ``never``

389. **\*\*How does TypeScript infer the return type of a function?\*\***

- A. Based on the return statements within the function.
- B. From the function's parameter types.
- C. Through explicit type annotations on the function.
- D. By analyzing the function's name.

390. **\*\*When would you use explicit type annotations in TypeScript?\*\***

- A. To override or specify the types when type inference is not sufficient or clear.
- B. To ensure all variables have a type even if it is obvious.
- C. To allow TypeScript to infer types more accurately.
- D. To simplify code and reduce the need for type checks.

### ### Topic: Generics (continued)

391. **\*\*What is the purpose of generics in TypeScript?\*\***

- A. To create reusable components that can work with a variety of types.
- B. To enforce strict type checking for specific data types.
- C. To define default values for variables.
- D. To limit the types of data that can be used in functions.

392. **\*\*How do you define a generic function in TypeScript?\*\***

- A.

```
```typescript
```

```
function identity<T>(value: T): T {  
    return value;  
}
```

```
```
```

- B.

```
```typescript
```

```
function identity(value: T): T {  
    return value;  
}
```

```
```
```

- C.

```
```typescript
function identity(value: any): any {
    return value;
}
```
```

- D.

```
```typescript
function identity<T>(value: any): T {
    return value;
}
```
```

393. **\*\*How do you use a generic type parameter in a class in TypeScript?\*\***

- A.

```
```typescript
class Box<T> {
    value: T;
    constructor(value: T) {
        this.value = value;
    }
}
```
```

- B.

```
```typescript
class Box<T> {
```

```
    constructor(public value: T) {}  
}  
...
```

- C.

```
```typescript  
class Box {  
    constructor(public value: T) {}  
}  
...
```

- D.

```
```typescript  
class Box<T> {  
    value: any;  
    constructor(value: T) {  
        this.value = value;  
    }  
}  
...
```

394. **\*\*What is the syntax for specifying multiple generic type parameters in TypeScript?\*\***

- A.

```
```typescript  
function combine<T, U>(a: T, b: U): [T, U] {  
    return [a, b];  
}  
...
```



- B.

```
```typescript
```

```
function combine<T, U, V>(a: T, b: U): [T, U] {  
    return [a, b];  
}
```

```
```
```

- C.

```
```typescript
```

```
function combine<T>(a: T, b: T): [T, T] {  
    return [a, b];  
}
```

```
```
```

- D.

```
```typescript
```

```
function combine<T>(a: T, b: any): [T, any] {  
    return [a, b];  
}
```

```
```
```

395. **\*\*How do you constrain generic types in TypeScript?\*\***

- A. By using the `extends` keyword in the generic type parameter.
- B. By defining the type directly in the generic parameter.
- C. By providing a default type for the generic parameter.
- D. By using type assertions within the generic type parameter.

### Topic: Type Guards (continued)

396. **What is a type guard in TypeScript?**

- A. A mechanism to narrow down the type of a variable within a specific scope.
- B. A method for checking the type of a variable at runtime.
- C. A utility to convert one type into another.
- D. A tool for generating type declarations automatically.

397. **How do you use the `typeof` operator as a type guard in TypeScript?**

- A. By checking the type of a variable and narrowing down its type.
- B. By converting the type of a variable to a specific type.
- C. By asserting the type of a variable explicitly.
- D. By defining a type alias for the variable's type.

398. **What is the syntax for using `instanceof` as a type guard in TypeScript?**

- A.

```
``typescript
if (value instanceof Date) {
    // value is of type Date
}
...

```

- B.

```
``typescript
if (value instanceof String) {
    // value is of type string
}
...

```

- C.

```
```typescript
```

```
if (value instanceof Number) {
```

```
    // value is
```

```
of type number
```

```
}
```

```
```
```

- D.

```
```typescript
```

```
if (value instanceof Object) {
```

```
    // value is of type Object
```

```
}
```

```
```
```

399. **\*\*How do you define a user-defined type guard function in TypeScript?\*\***

- A. By defining a function that returns a boolean and uses type predicates.
- B. By using type assertions inside the function.
- C. By specifying the return type as `boolean`.
- D. By using type checks inside the function.

400. **\*\*What is a type predicate in TypeScript?\*\***

- A. A return type of a function that asserts a variable's type.
- B. A way to define default values for types.
- C. A syntax for declaring complex types.
- D. A utility for generating type definitions.

Sure, let's continue from where we left off:

### Topic: Tuples (continued)

401. **\*\*How do you define a tuple with fixed types and lengths in TypeScript?\*\***

- A.

```
``typescript
let tuple: [string, number] = ['hello', 42];
...

```

- B.

```
``typescript
let tuple: [number, string] = [42, 'hello'];
...

```

- C.

```
``typescript
let tuple: [string, number, boolean] = ['hello', 42, true];
...

```

- D.

```
``typescript
let tuple: [number, string, boolean] = [42, 'hello', false];
...

```

402. **\*\*How can you access the elements of a tuple in TypeScript?\*\***

- A. Using index notation, e.g., `tuple[0]`.

- B. Using dot notation, e.g., `tuple.0`.

- C. Using array methods like `map()`.

- D. Using object destructuring only.

403. **\*\*What is the result of trying to assign a value to an out-of-bounds index in a tuple?\*\***

- A. TypeScript will throw an error because the tuple length is fixed.
- B. The value will be assigned without any issues.
- C. TypeScript will automatically resize the tuple.
- D. The value will be assigned but will be ignored at runtime.

404. **\*\*How do you define a tuple with optional elements in TypeScript?\*\***

- A.

```
```typescript
```

```
let tuple: [string, number?] = ['hello'];
```

```
```
```

- B.

```
```typescript
```

```
let tuple: [string?, number] = [undefined, 42];
```

```
```
```

- C.

```
```typescript
```

```
let tuple: [string, number | undefined] = ['hello'];
```

```
```
```

- D.

```
```typescript
```

```
let tuple: [string?, number?] = ['hello', 42];
```

```
```
```

405. **\*\*What is a rest element in a tuple?\*\***

- A. An element that captures all remaining values in a tuple as an array.
- B. An element that is automatically added to the end of a tuple.
- C. An element that is used to define the type of all tuple elements.
- D. An element that is used to exclude specific values from a tuple.

406. **\*\*How do you define a tuple with a rest element in TypeScript?\*\***

- A.

```
```typescript
let tuple: [string, ...number[]] = ['hello', 1, 2, 3];
```
```

- B.

```
```typescript
let tuple: [string, number, ...boolean[]] = ['hello', 42];
```
```

- C.

```
```typescript
let tuple: [string, ...number] = ['hello', 1, 2, 3];
```
```

- D.

```
```typescript
let tuple: [...string[], number] = ['hello', 'world', 42];
```
```

407. **\*\*How do you create a read-only tuple in TypeScript?\*\***

- A. Using the `ReadonlyTuple`` type.

- B. Using the ``readonly`` modifier.
- C. Using the ``const`` keyword.
- D. Using the ``immutable`` keyword.

408. **\*\*What will be the result of trying to push a new element into a read-only tuple?\*\***

- A. TypeScript will throw an error because read-only tuples cannot be modified.
- B. The new element will be added successfully.
- C. The existing elements will be replaced by the new element.
- D. TypeScript will automatically convert the tuple to a regular array.

409. **\*\*How do you specify the types of individual elements in a tuple with TypeScript?\*\***

- A. By listing each element's type in the tuple definition.
- B. By using a type alias with a generic.
- C. By defining a tuple type and then using type inference.
- D. By using object notation with named properties.

410. **\*\*What is the purpose of a tuple in TypeScript?\*\***

- A. To represent a fixed-size collection of elements with different types.
- B. To represent an unordered collection of elements with the same type.
- C. To create an object with dynamic properties.
- D. To define a type with optional properties.

### Topic: Async and Await (continued)

411. **\*\*How do you define an asynchronous function in TypeScript?\*\***

- A. By using the ``async`` keyword before the function declaration.

- B. By using the `await` keyword inside the function body.
- C. By declaring the function with a `Promise` return type.
- D. By defining the function as a generator function.

412. **What does the `await` keyword do in an asynchronous function?**

- A. It pauses the execution of the function until the `Promise` is resolved.
- B. It immediately returns the value of the `Promise` without waiting.
- C. It converts the function into a synchronous function.
- D. It throws an error if the `Promise` is rejected.

413. **What will be the result of the following TypeScript code?**

```
``typescript
async function fetchData(): Promise<number> {
    return 42;
}
...

```

- A. The function returns a `Promise` that resolves to `42`.
- B. The function returns `42` directly.
- C. The function throws an error.
- D. The function returns `Promise<number>`.

414. **How do you handle errors in an `async` function?**

- A. By using a `try` and `catch` block within the function.
- B. By using a `catch` block with the `Promise` returned from the function.
- C. By using `await` to handle exceptions.
- D. By specifying an `error` parameter in the function.



415. \*\*What is the syntax for chaining multiple asynchronous operations using `await`?

- A.

```
``typescript
async function processData() {
  let result1 = await fetchData();
  let result2 = await processResult(result1);
  return result2;
}
...

```

- B.

```
``typescript
async function processData() {
  let result1 = await fetchData().then(processResult);
  return result1;
}
...

```

- C.

```
``typescript
async function processData() {
  let result1 = fetchData();
  let result2 = processResult(result1);
  return result2;
}
...

```

- D.

```
``typescript

function processData() {

    let result1 = await fetchData();

    let result2 = await processResult(result1);

    return result2;

}

...

```

416. **\*\*What will happen if you forget to use `await` in an `async` function when calling another `async` function?\*\***

- A. The `Promise` returned by the called function will be ignored, and the code will execute asynchronously.
- B. The function will not compile due to a type error.
- C. The called function will be executed synchronously.
- D. The `async` function will throw an error.

417. **\*\*How do you ensure that a function always returns a `Promise` in TypeScript?\*\***

- A. By marking the function with the `async` keyword.
- B. By returning a `Promise` object explicitly.
- C. By using the `Promise.resolve()` method.
- D. By using the `Promise.all()` method.

418. **\*\*What is the correct way to use `await` with a `Promise` that might be rejected?\*\***

- A.

```
``typescript

async function handleData() {

```

```
try {  
    let result = await fetchData();  
    return result;  
} catch (error) {  
    console.error(error);  
}  
}
```

...

- B.

```typescript

```
async function handleData() {  
    let result = await fetchData().catch(error => console.error(error));  
    return result;  
}  
}
```

...

- C.

```typescript

```
function handleData() {  
    let result = await fetchData();  
    return result;  
}  
}
```

...

- D.

```typescript

```
async function handleData() {  
    let result = await fetchData();
```

```
    console.error(result);  
  
    return result;  
}  
...
```

419. **\*\*What is the type of a variable that stores the result of an `await` expression?\*\***

- A. The type of the `Promise` that was awaited.
- B. `Promise<any>`
- C. `void`
- D. `undefined`

420. **\*\*How do you handle multiple asynchronous tasks in parallel?\*\***

- A. By using `Promise.all()` to wait for all `Promises` to resolve.
- B. By using `Promise.race()` to wait for the first `Promise` to resolve.
- C. By using `await` on each `Promise` sequentially.
- D. By executing the `Promises` in a loop with `await`.

### Topic: Function Overloads (continued)

421. **\*\*How do you define multiple signatures for a function in TypeScript?\*\***

- A. By using function overloads with different parameter types.
- B. By using default parameters for each function signature.
- C. By defining multiple functions with

the same name but different implementations.

- D. By using type aliases to specify different function types.

422. **\*\*What is the syntax for defining a function overload in TypeScript?\*\***

- A.

```
```typescript
function greet(person: string): string;
function greet(person: string, age: number): string;
function greet(person: string, age?: number): string {
    if (age === undefined) {
        return `Hello, ${person}`;
    } else {
        return `Hello, ${person}. You are ${age} years old.`;
    }
}
```
```

- B.

```
```typescript
function greet(person: string, age?: number): string {
    return `Hello, ${person}`;
}
function greet(person: string, age: number): string {
    return `Hello, ${person}. You are ${age} years old.`;
}
```
```

- C.

```
```typescript
function greet(person: string): string;
```

```
function greet(person: string, age: number): void;
```

```
function greet(person: string, age?: number) {
```

```
    return `Hello, ${person}`;
```

```
}
```

```
...
```

- D.

```
```typescript
```

```
function greet(person: string, age?: number): string;
```

```
function greet(person: string, age: number) {
```

```
    return `Hello, ${person}. You are ${age} years old.`;
```

```
}
```

```
...
```

423. **What will happen if a function has multiple overloads and the call does not match any of them?**

- A. TypeScript will throw a compile-time error.
- B. The function will execute with default values.
- C. The function will execute with the first overload signature.
- D. The function will return `undefined`.

424. **How do you specify a function that can accept multiple types of arguments using overloads?**

- A. By defining different overload signatures for each type combination.
- B. By using `any` type for all parameters.
- C. By using type assertions inside the function body.
- D. By defining a single function with type unions for parameters.

425. **How do you call a function with overloads and specify the correct argument types?**

- A. By passing arguments that match one of the defined overload signatures.
- B. By using type assertions to convert arguments to the expected types.
- C. By specifying argument types explicitly in the function call.
- D. By using a type cast to match the function signature.

426. **\*\*What is the purpose of using function overloads in TypeScript?\*\***

- A. To define multiple ways to call a function based on different parameter types or counts.
- B. To simplify function definitions by combining multiple functions into one.
- C. To automatically handle various types of return values.
- D. To support dynamic typing and runtime type checks.

427. **\*\*How does TypeScript handle the return type of a function with multiple overloads?\*\***

- A. By using the return type specified in the implementation signature.
- B. By using the return type specified in the first overload signature.
- C. By using the return type specified in the last overload signature.
- D. By inferring the return type from the function body.

428. **\*\*How do you handle optional parameters in a function overload?\*\***

- A. By defining overload signatures with and without the optional parameter.
- B. By using default values for optional parameters in the function body.
- C. By using `undefined` as the default value for optional parameters.
- D. By specifying `null` as a default for optional parameters.

429. **\*\*What will be the result of calling a function with overloads where the provided arguments do not match any signature?\*\***

- A. TypeScript will generate a compile-time error.

- B. The function will use default values for missing arguments.
- C. The function will throw a runtime error.
- D. The function will attempt to match the closest overload signature.

430. **What is the correct way to handle different return types in function overloads?**

- A. Define multiple overload signatures with different return types.
- B. Use a union type for the return type in the function implementation.
- C. Specify `void` as the return type in the function implementation.
- D. Use type assertions to handle different return types.

### Topic: Functions (continued)

431. **What is the syntax for defining a function with an optional parameter in TypeScript?**

- A.

```
```typescript
```

```
function greet(name: string, age?: number): string {  
    return `Hello, ${name}`;  
}
```

```
```
```

- B.

```
```typescript
```

```
function greet(name: string, age: number = 0): string {  
    return `Hello, ${name}`;  
}
```

```
```
```

- C.



```
```typescript
```

```
function greet(name: string, age: number | undefined): string {  
    return `Hello, ${name}`;  
}
```

```
```
```

- D.

```
```typescript
```

```
function greet(name: string, age: number | null = null): string {  
    return `Hello, ${name}`;  
}
```

```
```
```

432. **\*\*How do you define a function with default parameters in TypeScript?\*\***

- A.

```
```typescript
```

```
function greet(name: string, age: number = 30): string {  
    return `Hello, ${name}. You are ${age} years old.`;  
}
```

```
```
```

- B.

```
```typescript
```

```
function greet(name: string, age: number | null): string {  
    return `Hello, ${name}. You are ${age || 30} years old.`;  
}
```

```
```
```

- C.

```
```typescript
```

```
function greet(name: string, age?: number): string {  
    return `Hello, ${name}. You are ${age ?? 30} years old.`;  
}
```

```
```
```

- D.

```
```typescript
```

```
function greet(name: string, age: number): string {  
    return `Hello, ${name}. You are ${age ? age : 30} years old.`;  
}
```

```
```
```

433. **What is the purpose of using the `rest` parameter in a function?**

- A. To allow a function to accept an arbitrary number of arguments.
- B. To specify default values for function parameters.
- C. To enforce a fixed number of arguments in a function.
- D. To define optional parameters in a function.

434. **What is the syntax for defining a function with a rest parameter in TypeScript?**

- A.

```
```typescript
```

```
function sum(...numbers: number[]): number {  
    return numbers.reduce((total, num) => total + num, 0);  
}
```

```
```
```

- B.

```typescript

```
function sum(numbers: number[]): number {  
    return numbers.reduce((total, num) => total + num, 0);  
}
```

```

- C.

```typescript

```
function sum(numbers: ...number): number {  
    return numbers.reduce((total, num) => total + num, 0);  
}
```

```

- D.

```typescript

```
function sum(...numbers: Array<number>): number {  
    return numbers.reduce((total, num) => total + num, 0);  
}
```

```

435. **\*\*How do you define a function with multiple optional parameters in TypeScript?\*\***

- A.

```typescript

```
function createProfile(name: string, age?: number, address?: string): string {  
    return `Name: ${name}, Age: ${age}, Address: ${address}`;  
}
```

```

- B.

```typescript

```
function createProfile(name: string, age?: number | undefined, address?: string | null): string {  
    return `Name: ${name}, Age: ${age}, Address: ${address}`;  
}
```

```

- C.

```typescript

```
function createProfile(name: string, age: number | undefined, address: string | undefined): string {  
    return `Name: ${name}, Age: ${age}, Address: ${address}`;  
}
```

```

- D.

```typescript

```
function createProfile(name: string, age: number, address: string = ""): string {  
    return `Name: ${name}, Age: ${age}, Address: ${address}`;  
}
```

```

436. **\*\*What is the difference between a regular function and an arrow function in TypeScript?\*\***

- A. Arrow functions do not have their own `this` context and are more concise.
- B. Regular functions cannot use default or rest parameters.
- C. Arrow functions can only be used as methods of objects.
- D. Regular functions are syntactically different but functionally identical to arrow functions.

437. **\*\*How do you define an arrow function in TypeScript?\*\***

- A.

```
```typescript
```

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

```
```
```

- B.

```
```typescript
```

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

```
```
```

- C.

```
```typescript
```

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

```
```
```

- D.

```
```typescript
```

```
const add = (a: number, b: number) => {
```

```
    return a + b;
```

```
};
```

```
```
```

438. **\*\*How do you specify the return type of a function in TypeScript?\*\***

- A. By using a type annotation after

the parameter list.

- B. By using a type assertion inside the function body.

- C. By defining the return type in the function implementation.

- D. By specifying a type alias for the return type.

439. **\*\*What is the default return type of a function in TypeScript if no return type is explicitly specified?\*\***

- A. ``void``
- B. ``any``
- C. ``undefined``
- D. ``null``

440. **\*\*How do you define a function that accepts another function as a parameter in TypeScript?\*\***

- A.

```
```typescript
function processData(callback: (data: string) => void): void {
    callback('sample data');
}
```
```

- B.

```
```typescript
function processData(callback: void): void {
    callback('sample data');
}
```
```

- C.

```
```typescript
function processData(callback: (data: string) => string): void {
    callback('sample data');
}
```
```

- D.

```
```typescript

function processData(callback: Function): void {

    callback('sample data');

}

```
```

### Topic: Arrays (continued)

441. **\*\*How do you declare an array of numbers in TypeScript?\*\***

- A.

```
```typescript

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

```
```

- B.

```
```typescript

let numbers: Array<number> = [1, 2, 3, 4];

```
```

- C.

```
```typescript

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

```
```

- D.

```
```typescript

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

```
```

442. **\*\*What is the syntax for accessing an element of an array in TypeScript?\*\***

- A. Using index notation, e.g., `array[0]`.
- B. Using dot notation, e.g., `array.0`.
- C. Using array methods like `find()`.
- D. Using object destructuring.

443. **\*\*How do you add a new element to the end of an array in TypeScript?\*\***

- A. By using the `push()` method.
- B. By using the `unshift()` method.
- C. By using the `splice()` method.
- D. By directly assigning a value to an index.

444. **\*\*What will be the result of using the `pop()` method on an array?\*\***

- A. It removes the last element from the array and returns it.
- B. It adds a new element to the end of the array.
- C. It removes the first element from the array and returns it.
- D. It returns the length of the array.

445. **\*\*How do you concatenate two arrays in TypeScript?\*\***

- A. By using the `concat()` method.
- B. By using the `merge()` method.
- C. By using the `push()` method with the spread operator.
- D. By using the `add()` method.

Certainly! Continuing from where we left off:



459. **What will happen if you try to assign a non-enum value to an enum variable?**

- A. TypeScript will throw a compile-time error.
- B. The value will be assigned, but it will not be recognized as part of the enum.
- C. The value will be implicitly converted to a valid enum value.
- D. TypeScript will ignore the invalid value.

460. **How do you create a computed enum member in TypeScript?**

- A. By assigning an expression to the enum member that evaluates to a value.
- B. By using a function to compute the enum value at runtime.
- C. By using the `const` keyword with the enum member.
- D. By using string concatenation in the enum member definition.

461. **What happens when you mix numeric and string values in an enum?**

- A. TypeScript will throw a compile-time error because enums cannot mix numeric and string values.
- B. TypeScript will allow it but will only use the numeric values for operations.
- C. The string values will override the numeric values.
- D. The numeric values will override the string values.

462. **How do you access the numeric value of an enum member?**

- A. By using the enum name and member name, e.g., `Direction.Up`.
- B. By converting the enum member to a number using `Number(Direction.Up)`.
- C. By using a reverse lookup on the enum object.
- D. By using the `toString()` method on the enum member.

463. **How do you define an enum member that starts from a specific value in TypeScript?**

- A. By assigning a specific value to the first member, and subsequent members will auto-increment.

- B. By defining the start value using the ``start`` keyword in the enum definition.
- C. By using a custom initializer for each member.
- D. By setting the start value using the ``initial`` keyword.

464. **\*\*Can you define an enum with mixed literal and computed values?\*\***

- A. Yes, you can mix literal and computed values within an enum.
- B. No, TypeScript requires all values to be either literals or computed.
- C. You can only define enums with literals and then override with computed values.
- D. Mixed literal and computed values are allowed but not recommended.

465. **\*\*How do you use a numeric enum as a key in an object?\*\***

- A. By using the enum member value as a key.
- B. By using the enum name as a key.
- C. By converting the numeric enum to a string.
- D. By using the ``toString()`` method on the enum member.

466. **\*\*How can you use enums to create a set of flags in TypeScript?\*\***

- A. By using a combination of numeric values with bitwise operators.
- B. By defining each flag with a unique string value.
- C. By using an array of enums and combining them.
- D. By creating a separate enum for each flag and combining them.

467. **\*\*What is the purpose of ``const enum`` in TypeScript?\*\***

- A. To provide compile-time inlining of enum values for optimization.
- B. To ensure that enum values are immutable.
- C. To allow enum values to be dynamically computed.

- D. To support the use of string literals in enums.

468. **\*\*How do you handle enum values that need to be compared in TypeScript?\*\***

- A. By using enum member names directly in comparisons.
- B. By comparing the numeric values or string values of the enum members.
- C. By using the ``equals()`` method on enum members.
- D. By using the ``compareTo()`` method for enum members.

469. **\*\*What is the default behavior when an enum member is assigned a string value?\*\***

- A. All other members must be assigned string values or left unassigned.
- B. The default behavior is to automatically assign numeric values to subsequent members.
- C. TypeScript will throw an error if mixed values are used.
- D. String values are treated the same as numeric values for enum operations.

470. **\*\*How do you define an enum where members are assigned to specific string values?\*\***

- A.

```
```typescript
```

```
enum Status {
```

```
    Active = 'ACTIVE',
```

```
    Inactive = 'INACTIVE',
```

```
    Pending = 'PENDING'
```

```
}
```

```
```
```

- B.

```
```typescript
```

```
enum Status {
```

```
Active = 1,  
  
Inactive = 'INACTIVE',  
  
Pending = 'PENDING'  
  
}
```

...

- C.

```
```typescript  
  
enum Status {  
  
    Active = 1,  
  
    Inactive = 2,  
  
    Pending = 3  
  
}  
  
...
```

- D.

```
```typescript  
  
const Status = {  
  
    Active: 'ACTIVE',  
  
    Inactive: 'INACTIVE',  
  
    Pending: 'PENDING'  
  
};  
  
...
```