20 TypeScript Questions for Senior Front-End Developers

Table of Contents

Question 1: Advanced Conditional Types

Question: Explain conditional types in TypeScript and create a utility type that extracts all function property names from an object type.

👉 Jump to Answer

Answer:

Conditional types allow you to create types based on conditions, following the pattern `T extends U ? X : Y`.

References:

- TypeScript Handbook - Conditional Types
- Advanced Types

```
// Generic utility type that extracts only function
property names
type FunctionPropertyNames<T> = {
  // Step 1: Iterate over all keys in T
  [K in keyof T]: T[K] extends Function ? K : never;
  // Step 2: If T[K] is a function, keep the key K,
otherwise return 'never'
}[keyof T];
// Step 3: Index with [keyof T] to extract the union of
all non-never values

// Usage example
interface User {
  id: number;           // Not a function - will be
filtered out
  name: string;         // Not a function - will be
filtered out
  save: () => void;     // Function - will be included
  update: (data: Partial<User>) => Promise<void>;  //
Function - will be included
  delete: () => Promise<boolean>;                  //
Function - will be included
}

// Extract only function property names
type UserFunctions = FunctionPropertyNames<User>;
// Result: 'save' | 'update' | 'delete'

// How it works:
// 1. Mapped type creates: { id: never, name: never,
save: 'save', update: 'update', delete: 'delete' }
// 2. Indexing with [keyof T] extracts: never | never |
'save' | 'update' | 'delete'
// 3. 'never' is eliminated from unions, leaving:
'save' | 'update' | 'delete'
```

Key Points:

- Conditional types enable type-level logic
- Mapped types iterate over properties
- `never` type is used to filter out unwanted properties
- Index access `[keyof T]` extracts the union of values

⬆ Back to Top

Question 2: Template Literal Types

Question: How do template literal types work in TypeScript 4.1+? Create a type-safe event emitter system using template literal types.

👉 Jump to Answer

Answer:

Template literal types allow you to manipulate string literal types at the type level.

References:

- TypeScript 4.1 - Template Literal Types
- TypeScript Release Notes 4.1

```
// Define available event names
type EventNames = 'user:created' | 'user:updated' |
'user:deleted';

// Create listener method names by capitalizing and
```

```
prefixing with 'on'
type ListenerName<T extends string> =
`on${Capitalize<T>}`;
// Example: 'user:created' → 'onUser:created'

// Basic event listeners without type-safe payloads
type EventListeners = {
  // Remap each event name to a listener method name
  [K in EventNames as ListenerName<K>]: (data: any) =>
void;
};
// Result:
// {
//   onUser:created: (data: any) => void;
//   onUser:updated: (data: any) => void;
//   onUser:deleted: (data: any) => void;
// }

// Advanced: Type-safe event emitter with specific
payload types
type EventPayloadMap = {
  'user:created': { id: string; name: string };
// Payload for user creation
  'user:updated': { id: string; changes: Record<string,
any> }; // Payload for updates
  'user:deleted': { id: string };
// Payload for deletion
};

// Create type-safe event emitter interface
type TypedEventEmitter = {
  // Map each event to a listener with the correct
payload type
  [K in keyof EventPayloadMap as ListenerName<K &
string>]: (
    payload: EventPayloadMap[K]  // Each listener gets
its specific payload type
  ) => void;
};

// Usage example:
// const emitter: TypedEventEmitter = {
//   'onUser:created': (payload) =>
console.log(payload.id, payload.name), // ✅ Type-safe
//   'onUser:updated': (payload) =>
console.log(payload.changes),          // ✅ Type-safe
//   'onUser:deleted': (payload) =>
console.log(payload.id),               // ✅ Type-safe
// };
```

Key Points:

- Template literal types manipulate strings at type level
- Key remapping with `as` clause enables property name transformations
- Intrinsic string manipulation utilities: `Capitalize`, `Uppercase`, `Lowercase`, `Uncapitalize`
- Combines with mapped types for powerful patterns

⬆ Back to Top

Question 3: Variance and Type Safety

Question: Explain covariance, contravariance, and invariance in TypeScript. Why are function parameters contravariant?

👉 Jump to Answer

Answer:

References:

- TypeScript FAQ - Covariance and Contravariance
- Understanding Variance in TypeScript

Covariance: Type T is covariant if `A extends B` implies `T<A> extends T<B>`. Arrays in TypeScript are covariant:

```
// Define class hierarchy
class Animal {
  name: string;
}

class Dog extends Animal {
  breed: string;  // Dog has all Animal properties +
breed
}

// Arrays are covariant - you can assign a more
specific array to a more general one
let animals: Animal[] = [];
let dogs: Dog[] = [];

animals = dogs; // ✅ OK - arrays are covariant
```

```
// This works because:
// 1. Dog extends Animal (Dog is a subtype of Animal)
// 2. Dog[] is treated as a subtype of Animal[]
// 3. It's safe to read (every Dog is an Animal)
// 4. But can be unsafe for writes: animals.push(new
Animal()) would add non-Dog to dogs array
```

Contravariance: Type T is contravariant if `A extends B` implies `T<B> extends T<A>`. Function parameters are contravariant:

```
type AnimalHandler = (animal: Animal) => void;  //
Accepts any Animal
type DogHandler = (dog: Dog) => void;           //
Accepts only Dog

// Create handlers
let handleAnimal: AnimalHandler = (animal) =>
console.log(animal.name);  // Works with any Animal
let handleDog: DogHandler = (dog) =>
console.log(dog.breed);              // Needs Dog-
specific property

// Function parameter contravariance
handleDog = handleAnimal; // ✅ OK - can substitute
more general handler
// Why this works:
// 1. handleDog expects to receive a Dog
// 2. handleAnimal can handle any Animal (including
Dog)
// 3. Since Dog extends Animal, handleAnimal can safely
handle Dogs
```

```
// 4. We're substituting a "more capable" handler

// handleAnimal = handleDog; // ❌ Error - cannot
substitute more specific handler
// Why this fails:
// 1. handleAnimal might receive any Animal (not just
Dog)
// 2. handleDog only knows how to work with Dogs
(accesses dog.breed)
// 3. If we passed a Cat to handleAnimal, it would try
to access cat.breed (doesn't exist!)
// 4. This would be unsafe
```

Why parameters are contravariant:

Function parameters are contravariant because you can safely pass a more general handler where a specific handler is expected. The general handler can handle all cases the specific handler could.

```
// Real-world example
interface Cat extends Animal { meow: () => void; }

// This function expects a DogHandler
function processDog(dog: Dog, handler: DogHandler) {
  handler(dog);  // Calls handler with a Dog
}

// We can pass a more general handler
const generalHandler: AnimalHandler = (animal) => {
  console.log(animal.name);  // Only uses Animal
properties
};

processDog(new Dog(), generalHandler);  // ✅ Safe -
generalHandler can handle Dogs
```

Invariance: Type T is invariant if neither `T<A> extends T<B>` nor `T<B> extends T<A>` when `A extends B`.

```
interface Box<T> {
  value: T;
  setValue: (value: T) => void;
}

let animalBox: Box<Animal>;
let dogBox: Box<Dog>;

// Both assignments fail - Box is invariant
// animalBox = dogBox; // Error
// dogBox = animalBox; // Error
```

Key Points:

- Use `strictFunctionTypes` to enforce proper variance checking
- Understanding variance prevents runtime type errors
- Return types are covariant, parameters are contravariant
- Bidirectional properties create invariance

⬆ Back to Top

Question 4: Advanced Generics with Constraints

Question: Create a type-safe deep partial implementation that works with nested objects and arrays.

👉 Jump to Answer

Answer:

References:

- TypeScript Handbook - Generics
- Advanced Generic Constraints

```
/**
 * DeepPartial - Recursively makes all properties
optional
 * Handles various built-in types: Objects, Arrays,
Maps, Sets
 */
type DeepPartial<T> = T extends object  // First, check
if T is an object type
  ? T extends Array<infer U>            // If it's an
array, extract element type U
    ? Array<DeepPartial<U>>             // Return array
with deep partial elements
    : T extends ReadonlyArray<infer U>  // Check for
readonly arrays
    ? ReadonlyArray<DeepPartial<U>>     // Return
readonly array with deep partial elements
    : T extends Map<infer K, infer V>   // Check for
```

```
Map type
    ? Map<K, DeepPartial<V>>            // Make Map
values deep partial (keys stay the same)
    : T extends Set<infer M>            // Check for
Set type
    ? Set<DeepPartial<M>>               // Make Set
values deep partial
    : { [K in keyof T]?: DeepPartial<T[K]> }  // For
regular objects, make all props optional and recurse
  : T;  // If not an object, return as-is (primitive
types)

// Real-world configuration interface
interface Config {
  database: {
    host: string;
    port: number;
    credentials: {
      username: string;
      password: string;
    };
  };
  cache: {
    enabled: boolean;
    ttl: number;
  };
  features: string[];
}

// Using DeepPartial to create a partial configuration
const partialConfig: DeepPartial<Config> = {
  database: {
    // ✅ host and port are optional due to DeepPartial
    credentials: {
      username: 'admin', // ✅ password is now optional!
      // password omitted - this is valid with DeepPartial
    },
  },
  // ✅ cache is completely optional - entire section can be omitted
  features: [], // ✅ features is optional, and its elements are too
};

// This is useful for:
// 1. Configuration overrides (only specify what you want to change)
// 2. Partial updates to deep object structures
// 3. Form data where not all nested fields are required
// 4. API patch requests with nested objects
```

Key Points:

- Recursive types handle nested structures
- Conditional types handle different object types (Array, Map, Set)
- `infer` keyword extracts type parameters
- Handles both mutable and readonly arrays

⬆ Back to Top

Question 5: Type Guards and Narrowing

Question: Explain different types of type guards in TypeScript and create a custom type guard for discriminated unions with error handling.

👉 Jump to Answer

Answer:

References:

- TypeScript Handbook - Narrowing
- Type Guards and Type Predicates

Built-in Type Guards:

```
// typeof guard
function processValue(value: string | number) {
  if (typeof value === 'string') {
    return value.toUpperCase();
  }
  return value.toFixed(2);
}

// instanceof guard
class CustomError extends Error {
  code: number;
}

function handleError(error: Error | CustomError) {
  if (error instanceof CustomError) {
    console.log(error.code);
  }
}

// in operator
interface Cat { meow: () => void; }
interface Dog { bark: () => void; }

function makeSound(animal: Cat | Dog) {
  if ('meow' in animal) {
    animal.meow();
  } else {
    animal.bark();
  }
}
```

Custom Type Guard with Discriminated Union:

```
/**
 * Discriminated Union Pattern for API Results
 * Each type has a unique 'type' property (the
discriminant)
 */

// Success case - contains the actual data
interface Success<T> {
  type: 'success';  // Literal type - discriminant
property
  data: T;          // Generic data payload
}

```

```
// Validation error case - multiple field errors
interface ValidationError {
  type: 'validation_error';  // Different literal type
  errors: Record<string, string[]>;  // Map of field
names to error messages
}

// Server error case - single error message with code
interface ServerError {
  type: 'server_error';  // Another unique literal type
  message: string;       // Error description
  code: number;          // HTTP status code or custom
error code
}

// Union type combining all possible results
type Result<T> = Success<T> | ValidationError |
ServerError;

// Custom type guard - narrows Result to Success
// The 'result is Success<T>' tells TypeScript to
narrow the type
function isSuccess<T>(result: Result<T>): result is
Success<T> {
  return result.type === 'success';
  // After this check returns true, TypeScript knows
result.data exists
}

function isValidationError<T>(result: Result<T>):
result is ValidationError {
  return result.type === 'validation_error';
  // After this check, TypeScript knows result.errors
exists
}

function isServerError<T>(result: Result<T>): result is
ServerError {
  return result.type === 'server_error';
  // After this check, TypeScript knows result.code and
result.message exist
}

// Usage with exhaustive checking
function handleResult<T>(result: Result<T>): string {
  // Check for success case
  if (isSuccess(result)) {
    // ✅ TypeScript knows result is Success<T> here
    // result.data is accessible and type-safe
    return `Success: ${JSON.stringify(result.data)}`;
  }

  // Check for validation error
  if (isValidationError(result)) {
    // ✅ TypeScript knows result is ValidationError
here
    // result.errors is accessible
    return `Validation errors:
${Object.keys(result.errors).join(', ')}`;
  }

  // Check for server error
  if (isServerError(result)) {
    // ✅ TypeScript knows result is ServerError here
    // result.code and result.message are accessible
    return `Server error ${result.code}:
${result.message}`;
  }

  // Exhaustive check - ensures all cases are handled
  // If we add a new type to Result but forget to
handle it,
  // TypeScript will error here because result won't be
assignable to never
  const _exhaustive: never = result;
  return _exhaustive;
}

// Example usage:
const apiResult: Result<{ userId: string }> = {
  type: 'success',
  data: { userId: '123' }
};

console.log(handleResult(apiResult)); // "Success:
{"userId":"123"}"
```

Key Points:

- Type guards narrow types within conditional blocks
- Custom type guards use `is` keyword for user-defined predicates
- Discriminated unions use literal types for type narrowing
- Exhaustive checking ensures all cases are handled

⬆ Back to Top

Question 6: Mapped Types and Key Remapping

Question: Create a utility type that converts all methods of a class to async methods and adds error handling.

👉 Jump to Answer

Answer:

References:

- TypeScript Handbook - Mapped Types
- Key Remapping in Mapped Types

```
type AsyncifyMethods<T> = {
  [K in keyof T as T[K] extends (...args: any[]) => any
    ? K
    : never]: T[K] extends (...args: infer Args) =>
infer Return
    ? Return extends Promise<any>
      ? T[K] // Already async
      : (...args: Args) => Promise<Return>
    : never;
};

// Extract only methods
type MethodsOnly<T> = {
  [K in keyof T as T[K] extends (...args: any[]) => any
? K : never]: T[K];
```

```
};

// Asyncify with error handling
type AsyncWithErrorHandling<T> = {
  [K in keyof T as T[K] extends (...args: any[]) => any
    ? K
    : never]: T[K] extends (...args: infer Args) =>
infer Return
    ? (
        ...args: Args
      ) => Promise<
        | { success: true; data: Awaited<Return> }
        | { success: false; error: Error }
      >
    : never;
};

// Usage
class UserService {
  getUser(id: string): { name: string; email: string }
{
    return { name: 'John', email: 'john@example.com' };
  }

  updateUser(id: string, data: Partial<{ name: string
}>): boolean {
    return true;
  }

  async deleteUser(id: string): Promise<void> {
    // Already async
  }
```

```
}

type AsyncUserService = AsyncifyMethods<UserService>;
// Result:
// {
//   getUser: (id: string) => Promise<{ name: string;
email: string }>;
//   updateUser: (id: string, data: Partial<{ name:
string }>) => Promise<boolean>;
//   deleteUser: (id: string) => Promise<void>;
// }

type SafeUserService =
AsyncWithErrorHandling<UserService>;
// Methods return Promise<{success: true, data: T} |
{success: false, error: Error}>
```

Key Points:

- Key remapping with `as` filters and transforms property names
- `infer` extracts function parameter and return types
- `Awaited<T>` unwraps Promise types
- Conditional types handle already-async methods

⬆ Back to Top

Question 7: Infer Keyword and Type Extraction

Question: Explain the `infer` keyword and create utility types to extract deeply nested types from complex structures.

👉 Jump to Answer

Answer:

The `infer` keyword declares a type variable within a conditional type's extends clause, allowing you to extract types from complex structures.

References:

- TypeScript Handbook - Type Inference in Conditional Types
- Advanced Conditional Types

```
// Extract return type
type ReturnType<T> = T extends (...args: any[]) =>
infer R ? R : never;

// Extract promise value type
type Awaited<T> = T extends Promise<infer U> ?
Awaited<U> : T;

// Extract array element type
type ElementType<T> = T extends (infer U)[] ? U :
never;

// Advanced: Extract nested property types
type NestedPropertyType<
  T,
  Path extends string
```

```
> = Path extends `${infer First}.${infer Rest}`
  ? First extends keyof T
    ? NestedPropertyType<T[First], Rest>
    : never
  : Path extends keyof T
  ? T[Path]
  : never;

interface User {
  profile: {
    address: {
      street: string;
      city: string;
    };
  };
}

type StreetType = NestedPropertyType<User,
'profile.address.street'>; // string

// Extract function parameter types at specific
positions
type FirstParameter<T> = T extends (arg: infer P,
...args: any[]) => any
  ? P
  : never;

type SecondParameter<T> = T extends (
  arg1: any,
  arg2: infer P,
  ...args: any[]
) => any
```

```
  ? P
  : never;

// Extract constructor parameter types
type ConstructorParameters<T extends abstract new
(...args: any) => any> =
  T extends abstract new (...args: infer P) => any ? P
: never;

// Extract instance type from constructor
type InstanceType<T extends abstract new (...args: any)
=> any> =
  T extends abstract new (...args: any) => infer R ? R
: any;

// Advanced: Extract all function types from an object
type FunctionTypes<T> = {
  [K in keyof T]: T[K] extends (...args: infer Args) =>
infer Return
    ? { args: Args; return: Return }
    : never;
};

interface API {
  getUser: (id: string) => Promise<User>;
  updateUser: (id: string, data: Partial<User>) =>
Promise<boolean>;
}

type APIFunctionTypes = FunctionTypes<API>;
// Result:
// {
```

```
//   getUser: { args: [string]; return: Promise<User>
};
//   updateUser: { args: [string, Partial<User>];
return: Promise<boolean> };
// }
```

Key Points:

- `infer` creates type variables within conditional types
- Multiple `infer` declarations can extract different parts
- Recursive types with `infer` handle nested structures
- Combines with template literals for path-based type extraction

⬆ Back to Top

Question 8: Utility Types Deep Dive

Question: Implement `Required`, `Readonly`, `Pick`, and `Omit` utility types from scratch and explain their internal workings.

👉 Jump to Answer

Answer:

References:

- TypeScript Handbook - Utility Types
- Mapped Type Modifiers

```
// Required - makes all properties required
type MyRequired<T> = {
  [K in keyof T]-?: T[K];
};

// Readonly - makes all properties readonly
type MyReadonly<T> = {
  readonly [K in keyof T]: T[K];
};

// Pick - selects specific properties
type MyPick<T, K extends keyof T> = {
  [P in K]: T[P];
};

// Omit - excludes specific properties
type MyExclude<T, U> = T extends U ? never : T;
type MyOmit<T, K extends keyof any> = MyPick<T,
MyExclude<keyof T, K>>;

// Advanced utility types
// Mutable - removes readonly
type Mutable<T> = {
  -readonly [K in keyof T]: T[K];
};

// Optional - makes properties optional
type Optional<T> = {
  [K in keyof T]+?: T[K];
};
```

```
// DeepRequired - makes all nested properties required
type DeepRequired<T> = T extends object
  ? {
      [K in keyof T]-?: DeepRequired<T[K]>;
    }
  : T;

// DeepReadonly - makes all nested properties readonly
type DeepReadonly<T> = T extends object
  ? {
      readonly [K in keyof T]: DeepReadonly<T[K]>;
    }
  : T;

// PickByType - pick properties by value type
type PickByType<T, U> = {
  [K in keyof T as T[K] extends U ? K : never]: T[K];
};

// OmitByType - omit properties by value type
type OmitByType<T, U> = {
  [K in keyof T as T[K] extends U ? never : K]: T[K];
};

// Usage
interface User {
  readonly id: string;
  name?: string;
  email: string;
  age?: number;
  createdAt: Date;
}
```

```
type MutableUser = Mutable<User>; // All readonly
removed
type CompleteUser = MyRequired<User>; // No optional
properties
type UserInfo = MyPick<User, 'name' | 'email'>; // Only
name and email
type UserWithoutDates = OmitByType<User, Date>; //
Exclude Date properties
type StringProperties = PickByType<User, string>; //
Only string properties
```

Key Points:

- `-?` removes optional modifier
- `+?` adds optional modifier
- `-readonly` removes readonly modifier
- `readonly` adds readonly modifier
- Mapped types iterate over properties with transformations
- Conditional types filter properties

⬆️ Back to Top

Question 9: Type Inference in Generic Functions

Question: Explain how TypeScript infers generic types and create a type-safe builder pattern with method chaining.

👉 Jump to Answer

Answer:

References:

- TypeScript Handbook - Generic Functions
- Type Inference

```
/**
 * Type-safe QueryBuilder using generics and method
chaining
 * Demonstrates type inference and fluent API pattern
 */
class QueryBuilder<T> {
  // Store filter predicates
  private filters: Array<(item: T) => boolean> = [];
  // Optional sort function
  private sortFn?: (a: T, b: T) => number;
  // Optional limit for result count
  private limitValue?: number;

  /**
   * Add a filter condition
   * K extends keyof T ensures 'key' is a valid
property of T
   * TypeScript infers the type of 'value' parameter in
predicate based on T[K]
   */
  where<K extends keyof T>(
    key: K,                          // Must be a
valid key of T
```

```
    predicate: (value: T[K]) => boolean  // Predicate
receives the correct type for that key
  ): QueryBuilder<T> {
    // Store a filter that extracts the key and applies
the predicate
    this.filters.push((item) => predicate(item[key]));
    return this;  // Return this for method chaining
  }

  /**
   * Sort by a specific property
   * Type-safe: can only sort by existing properties of
T
   */
  sort<K extends keyof T>(
    key: K,                    // Must be a valid key
of T
    order: 'asc' | 'desc' = 'asc'  // Optional order
parameter with literal types
  ): QueryBuilder<T> {
    this.sortFn = (a, b) => {
      const aVal = a[key];     // TypeScript knows
a[key] is type T[K]
      const bVal = b[key];     // TypeScript knows
b[key] is type T[K]

      // Compare values (works with numbers, strings,
etc.)
      const result = aVal < bVal ? -1 : aVal > bVal ? 1
: 0;

      // Reverse result if descending order
      return order === 'asc' ? result : -result;
    };
    return this;  // Return this for chaining
  }

  /**
   * Limit the number of results
   */
  limit(count: number): QueryBuilder<T> {
    this.limitValue = count;
    return this;  // Return this for chaining
  }

  /**
   * Execute the query and return
filtered/sorted/limited results
   */
  execute(data: T[]): T[] {
    // Step 1: Apply all filters
    // Item must pass ALL filters (every returns true
only if all predicates return true)
    let result = data.filter((item) =>
      this.filters.every((filter) => filter(item))
    );

    // Step 2: Apply sorting if specified
    if (this.sortFn) {
      result = result.sort(this.sortFn);
    }

    // Step 3: Apply limit if specified
    if (this.limitValue !== undefined) {
      result = result.slice(0, this.limitValue);
    }

    return result;
  }
}

// Example usage with type inference
interface User {
  id: string;
  name: string;
  age: number;
  email: string;
}

const users: User[] = [
  { id: '1', name: 'Alice', age: 30, email:
'alice@example.com' },
  { id: '2', name: 'Bob', age: 25, email:
'bob@example.com' },
  { id: '3', name: 'Alex', age: 35, email:
'alex@example.com' },
];

// TypeScript infers all types from User interface
const result = new QueryBuilder<User>()
  .where('age', (age) => age > 20)          // ✅ age is
inferred as number
  .where('name', (name) => name.startsWith('A'))  // ✅
name is inferred as string
  .sort('age', 'desc')                      // ✅ 'age'
must be keyof User
```

```
  .limit(10)                                    // ✅ Simple
number parameter
  .execute(users);

console.log(result); // [{ id: '3', name: 'Alex', ...
}, { id: '1', name: 'Alice', ... }]
```

// Advanced builder with type transformation

class TypedBuilder<T extends Record<string, any>> {

private data: Partial = {};

set(key: K, value: T[K]): TypedBuilder {

this.data[key] = value;

return this;

}

// Transform builder to return different type

map(fn: (data: Partial) => U): U {

return fn(this.data);

}

// Type-safe build requiring all properties

build(this: TypedBuilder & { data: T }): T {

return this.data;

}

// Partial build allowing incomplete data

buildPartial(): Partial {

return this.data;

```
}
}

// Usage
interface User {
id: string;
name: string;
age: number;
email: string;
}

const users: User[] = [
{ id: '1', name: 'Alice', age: 30, email: 'alice@example.com' },
{ id: '2', name: 'Bob', age: 25, email: 'bob@example.com' },
];

const result = new QueryBuilder()
.where('age', (age) => age > 20)
.where('name', (name) => name.startsWith('A'))
.sort('age', 'desc')
.limit(10)
.execute(users);

const user = new TypedBuilder()
.set('id', '123')
.set('name', 'John')
.set('age', 30)
.set('email', 'john@example.com')
.buildPartial(); // Partial
```

```
**Key Points:**
- Generic type parameters are inferred from arguments
- `keyof T` constrains keys to object properties
- Method chaining returns `this` type for fluent API
- Conditional `this` types enable type-safe builders
- Type inference flows through method chains

**[⬆ Back to Top](#table-of-contents)**


---


## Question 10: Module Resolution and Declaration Merging

**Question:** Explain TypeScript's module resolution strategies and declaration merging. How do you augment existing types from third-party libraries?

**[👉 Jump to Answer](#answer-10)**

<a id="answer-10"></a>
**Answer:**

**References:**
- [TypeScript Module Resolution](https://www.typescriptlang.org/docs/handbook/module-resolution.html)
- [Declaration Merging](https://www.typescriptlang.org/docs/handbook/declaration-merging.html)
```

~~~
- [Augmenting Global Types](https://www.typescriptlang.org/docs/handbook/declaration-files/templates/global-modifying-module-d-ts.html)

**Module Resolution Strategies:**

```typescript
// Classic Resolution (deprecated)
// import { x } from './moduleB'
// Looks for: ./moduleB.ts, ./moduleB.d.ts

// Node Resolution (common)
// import { x } from 'moduleB'
// Looks for:
// 1. node_modules/moduleB.ts
// 2. node_modules/moduleB/package.json (types field)
// 3. node_modules/moduleB/index.ts
// 4. node_modules/@types/moduleB/index.d.ts

// Node16/NodeNext Resolution (modern)
// Respects package.json "type": "module" and "exports" field
~~~

Declaration Merging:

```
// Interface merging
interface User {
  id: string;
  name: string;
}
```

```
interface User {
  email: string;
}

// Merged result:
// interface User {
//   id: string;
//   name: string;
//   email: string;
// }

// Namespace and class merging
class Album {
  label!: Album.AlbumLabel;
}

namespace Album {
  export interface AlbumLabel {
    name: string;
  }
}

// Function and namespace merging
function buildQuery(query: string): string {
  return query;
}

namespace buildQuery {
  export function parse(query: string): object {
    return JSON.parse(query);
  }
}
```

```
}

const query = buildQuery('SELECT * FROM users');
const parsed = buildQuery.parse('{"key": "value"}');
```

Augmenting Third-Party Libraries:

```
// Augment Express Request
declare global {
  namespace Express {
    interface Request {
      user?: {
        id: string;
        email: string;
      };
    }
  }
}

// Augment Window object
declare global {
  interface Window {
    gtag: (
      command: string,
      targetId: string,
      config?: Record<string, any>
    ) => void;
    dataLayer: any[];
  }
}

// Augment Array prototype (not recommended in
production)
declare global {
  interface Array<T> {
    last(): T | undefined;
  }
}

Array.prototype.last = function () {
  return this[this.length - 1];
};

// Module augmentation for libraries
import 'axios';

declare module 'axios' {
  export interface AxiosRequestConfig {
    retry?: number;
    retryDelay?: number;
  }
}

// Usage
import axios from 'axios';

axios.get('/api/users', {
  retry: 3,
  retryDelay: 1000,
});
```

tsconfig.json settings:

```
{
  "compilerOptions": {
    "moduleResolution": "node16",
    "module": "esnext",
    "target": "es2020",
    "esModuleInterop": true,
    "resolveJsonModule": true,
    "paths": {
      "@/*": ["./src/*"],
      "@components/*": ["./src/components/*"]
    },
    "baseUrl": ".",
    "typeRoots": ["./node_modules/@types",
"./src/types"]
  }
}
```

Key Points:

- Module resolution affects import behavior
- Declaration merging combines multiple declarations
- Global augmentation modifies global namespace
- Module augmentation extends existing modules
- Use `.d.ts` files for type declarations

⬆ Back to Top

Question 11: Advanced Type Narrowing with Control Flow Analysis

Question: How does TypeScript's control flow analysis work? Demonstrate complex narrowing scenarios including discriminated unions and exhaustive checking.

👉 Jump to Answer

Answer:

References:

- TypeScript Control Flow Analysis
- Discriminated Unions
- Assertion Functions

```
// Discriminated Unions
interface Circle {
  kind: 'circle';
  radius: number;
}

interface Rectangle {
  kind: 'rectangle';
  width: number;
  height: number;
}
```

```
interface Triangle {
  kind: 'triangle';
  base: number;
  height: number;
}

type Shape = Circle | Rectangle | Triangle;

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case 'circle':
      // TypeScript knows shape is Circle
      return Math.PI * shape.radius ** 2;
    case 'rectangle':
      // TypeScript knows shape is Rectangle
      return shape.width * shape.height;
    case 'triangle':
      // TypeScript knows shape is Triangle
      return (shape.base * shape.height) / 2;
    default:
      // Exhaustive check
      const _exhaustive: never = shape;
      throw new Error(`Unhandled shape: 
${_exhaustive}`);
  }
}

// Control flow analysis with assertions
function processValue(value: string | null | 
undefined): string {
  if (!value) {
    throw new Error('Value is required');
  }
  // TypeScript knows value is string here
  return value.toUpperCase();
}

// Type predicates with unknown
function isString(value: unknown): value is string {
  return typeof value === 'string';
}

function isNumber(value: unknown): value is number {
  return typeof value === 'number';
}

function processUnknown(value: unknown): string |
number {
  if (isString(value)) {
    return value.toUpperCase();
  }
  if (isNumber(value)) {
    return value.toFixed(2);
  }
  throw new Error('Invalid value type');
}

// Narrowing with truthiness checks
function processArray<T>(arr: T[] | null | undefined):
T[] {
  // Narrows to T[] | null | undefined
  if (arr == null) {
    return [];
  }
  // Narrows to T[]
  return arr.filter((item) => item != null);
}

// Narrowing with property checks
interface Dog {
  type: 'dog';
  bark: () => void;
}

interface Cat {
  type: 'cat';
  meow: () => void;
}

type Pet = Dog | Cat;

function makeSound(pet: Pet): void {
  if ('bark' in pet) {
    pet.bark(); // TypeScript knows pet is Dog
  } else {
    pet.meow(); // TypeScript knows pet is Cat
  }
}

// Advanced: Narrowing with assertion functions
function assertIsString(value: unknown): asserts value 
is string {
  if (typeof value !== 'string') {
    throw new Error('Value must be a string');
  }
}
```

```typescript
function processUnknownValue(value: unknown): string {
  assertIsString(value);
  // TypeScript knows value is string after assertion
  return value.toUpperCase();
}

// Assertion function for non-null
function assertNonNull<T>(value: T | null | undefined): 
asserts value is T {
  if (value == null) {
    throw new Error('Value is null or undefined');
  }
}

// Narrowing with instanceof and class hierarchies
class NetworkError extends Error {
  statusCode: number;
  constructor(message: string, statusCode: number) {
    super(message);
    this.statusCode = statusCode;
  }
}

class ValidationError extends Error {
  fields: Record<string, string[]>;
  constructor(message: string, fields: Record<string, 
string[]>) {
    super(message);
    this.fields = fields;
  }
}
```

```
function handleError(error: Error | NetworkError |
ValidationError): void {
  if (error instanceof NetworkError) {
    console.log(`Network error: ${error.statusCode}`);
  } else if (error instanceof ValidationError) {
    console.log(`Validation errors:
${Object.keys(error.fields)}`);
  } else {
    console.log(`Generic error: ${error.message}`);
  }
}
```

Key Points:

- Control flow analysis tracks type narrowing through code paths
- Discriminated unions use literal types for narrowing
- Exhaustive checking ensures all cases are handled
- Assertion functions modify type in calling scope
- Type predicates enable custom narrowing logic

⬆ Back to Top

Question 12: Performance and Compilation Optimization

Question: What TypeScript compiler options and patterns improve compilation performance? Explain project references and incremental compilation.

👉 Jump to Answer

Answer:

References:

- TypeScript Compiler Options
- Project References
- Performance Best Practices

Compiler Optimization Options:

```
// tsconfig.json
{
  "compilerOptions": {
    // Incremental compilation
    "incremental": true,
    "tsBuildInfoFile": "./.tsbuildinfo",

    // Skip type checking of declaration files
    "skipLibCheck": true,

    // Skip default library declaration file check
    "skipDefaultLibCheck": true,

    // Don't emit comments
    "removeComments": true,

    // Faster builds with less checking
    "isolatedModules": true,
    // Disable source maps in development
    "sourceMap": false,

    // Faster module resolution
    "moduleResolution": "bundler",

    // Only emit declaration files
    "declaration": true,
    "emitDeclarationOnly": true
  }
}
```

Project References for Monorepos:

```
// Root tsconfig.json
{
  "files": [],
  "references": [
    { "path": "./packages/common" },
    { "path": "./packages/client" },
    { "path": "./packages/server" }
  ]
}

// packages/common/tsconfig.json
{
  "compilerOptions": {
    "composite": true,
    "declaration": true,
    "outDir": "dist",
    "rootDir": "src"
  },
  "include": ["src/**/*"]
}

// packages/client/tsconfig.json
{
  "compilerOptions": {
    "composite": true,
    "outDir": "dist"
  },
  "references": [
    { "path": "../common" }
  ]
}
```

Performance Patterns:

```
// 1. Use type imports to avoid runtime dependencies
import type { User } from './types';

// 2. Avoid complex conditional types in hot paths
type Slow<T> = T extends { [K in keyof T]: infer U } ?
U : never;

// Better: Use simpler types
type Fast<T> = T[keyof T];

// 3. Use const assertions for literal types
const config = {
  apiUrl: 'https://api.example.com',
  timeout: 5000,
} as const;

// Instead of defining explicit type
type Config = {
  readonly apiUrl: 'https://api.example.com';
  readonly timeout: 5000;
};

// 4. Use interfaces over type aliases for object types
// Interfaces are cached better by the compiler
interface User {
  id: string;
  name: string;
}

// 5. Avoid excessive union types
// Slow: type Slow = A | B | C | D | E | ... | Z;
// Better: Group related types
type FastGroup1 = A | B | C;
type FastGroup2 = D | E | F;

// 6. Use discriminated unions instead of complex 
conditionals
type ApiResponse<T> =
  | { status: 'success'; data: T }
  | { status: 'error'; error: Error };

// 7. Leverage satisfies operator for type checking 
without widening
const routes = {
  home: '/',
  about: '/about',
  contact: '/contact',
} satisfies Record<string, string>;

// routes still has literal types, not string
```

Build Performance Tips:

```
# Use tsc with --build flag for project references
tsc --build

# Use --incremental for faster rebuilds
tsc --incremental

# Use --watch for development
tsc --watch

# Use --diagnostics to identify slow compilation
tsc --diagnostics

# Clear cache if needed
tsc --build --clean
```

Key Points:

- `skipLibCheck` significantly improves compilation speed
- Project references enable independent package compilation
- Incremental compilation reuses previous build information
- `isolatedModules` ensures each file can be compiled independently
- Use `type` imports for type-only dependencies

- Interfaces are faster than type aliases for objects

⬆ Back to Top

Question 13: Decorators and Metadata Reflection

Question: Explain TypeScript decorators and metadata reflection. Implement a validation decorator system using reflect-metadata.

👉 Jump to Answer

Answer:

References:

- TypeScript Decorators
- Reflect Metadata
- TC39 Decorators Proposal

```
// Enable decorators and metadata
// tsconfig.json: "experimentalDecorators": true, 
"emitDecoratorMetadata": true

import 'reflect-metadata';

// Metadata keys
const VALIDATION_METADATA = 
Symbol('validation:metadata');
const REQUIRED_METADATA = Symbol('required:metadata');
```

```
// Validation decorator
function Validate(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  const originalMethod = descriptor.value;

  descriptor.value = function (...args: any[]) {
    // Get parameter types from metadata
    const paramTypes = Reflect.getMetadata(
      'design:paramtypes',
      target,
      propertyKey
    );

    // Get validation rules
    const validations =
      Reflect.getMetadata(VALIDATION_METADATA, target, 
propertyKey) || {};

    // Validate each parameter
    args.forEach((arg, index) => {
      const validation = validations[index];
      if (validation) {
        validation.forEach((rule: any) => {
          if (!rule.validate(arg)) {
            throw new Error(`Validation failed: 
${rule.message}`);
          }
        });
      }
    });

    return originalMethod.apply(this, args);
  };

  return descriptor;
}

// Parameter decorators
function Min(min: number) {
  return function (
    target: any,
    propertyKey: string | symbol,
    parameterIndex: number
  ) {
    const validations =
      Reflect.getMetadata(VALIDATION_METADATA, target, 
propertyKey) || {};

    if (!validations[parameterIndex]) {
      validations[parameterIndex] = [];
    }

    validations[parameterIndex].push({
      validate: (value: number) => value >= min,
      message: `Value must be at least ${min}`,
    });

    Reflect.defineMetadata(
      VALIDATION_METADATA,
      validations,
      target,
      propertyKey
    );
  };
}

function Max(max: number) {
  return function (
    target: any,
    propertyKey: string | symbol,
    parameterIndex: number
  ) {
    const validations =
      Reflect.getMetadata(VALIDATION_METADATA, target, 
propertyKey) || {};

    if (!validations[parameterIndex]) {
      validations[parameterIndex] = [];
    }

    validations[parameterIndex].push({
      validate: (value: number) => value <= max,
      message: `Value must be at most ${max}`,
    });

    Reflect.defineMetadata(
      VALIDATION_METADATA,
      validations,
      target,
      propertyKey
    );
  };
```

```
}

function Required() {
  return function (
    target: any,
    propertyKey: string | symbol,
    parameterIndex: number
  ) {
    const requiredParams =
      Reflect.getMetadata(REQUIRED_METADATA, target, 
propertyKey) || [];
    requiredParams.push(parameterIndex);
    Reflect.defineMetadata(
      REQUIRED_METADATA,
      requiredParams,
      target,
      propertyKey
    );
  };
}

// Property decorator
function MinLength(length: number) {
  return function (target: any, propertyKey: string) {
    let value: string;

    const getter = () => value;
    const setter = (newValue: string) => {
      if (newValue.length < length) {
        throw new Error(
          `${propertyKey} must be at least ${length} 
characters`
```

```
        );
      }
      value = newValue;
    };

    Object.defineProperty(target, propertyKey, {
      get: getter,
      set: setter,
      enumerable: true,
      configurable: true,
    });
  };
}

// Usage
class UserService {
  @Validate
  createUser(@Required() @Min(1) id: number, 
@Required() name: string) {
    return { id, name };
  }

  @Validate
  updateAge(@Min(0) @Max(150) age: number) {
    return { age };
  }
}

class User {
  @MinLength(3)
  username!: string;
  @MinLength(8)
  password!: string;
}

// Test
const service = new UserService();
try {
  service.createUser(1, 'John'); // OK
  service.createUser(-1, 'John'); // Error: Validation 
failed
  service.updateAge(200); // Error: Value must be at 
most 150
} catch (error) {
  console.error(error);
}

const user = new User();
user.username = 'ab'; // Error: username must be at 
least 3 characters
```

Key Points:

- Decorators are experimental feature requiring `experimentalDecorators`
- `emitDecoratorMetadata` enables runtime type information
- `reflect-metadata` library enables metadata storage and retrieval
- Four types: class, method, property, parameter decorators
- Decorators execute at class definition time, not instantiation
- Multiple decorators compose from bottom to top

⬆ Back to Top

Question 14: Type-Level Programming with Recursive Types

Question: Create a type-level JSON path validator that ensures type-safe access to nested object properties using string paths.

👉 Jump to Answer

Answer:

References:

- TypeScript Recursive Types
- Template Literal Types for Path Validation

```
// Type-level path validation
type PathValue<T, Path extends string> = Path extends 
keyof T
  ? T[Path]
  : Path extends `${infer K}.${infer Rest}`
  ? K extends keyof T
    ? PathValue<T[K], Rest>
    : never
  : never;

// Generate all valid paths
type Paths<T, Prefix extends string = ''> = T extends 
object
  ? {
      [K in keyof T]: K extends string
        ? T[K] extends object
          ?
              | `${Prefix}${K}`
              | Paths<T[K], `${Prefix}${K}.`>
          : `${Prefix}${K}`
        : never;
    }[keyof T]
  : never;

// Type-safe get function
function get<T, P extends Paths<T>>(
  obj: T,
  path: P
): PathValue<T, P> {
  const keys = (path as string).split('.');
  let result: any = obj;

  for (const key of keys) {
    result = result?.[key];
  }

  return result;
}

// Type-safe set function
type SetPath<T, Path extends string, Value> = Path
extends keyof T
  ? Omit<T, Path> & { [K in Path]: Value }
  : Path extends `${infer K}.${infer Rest}`
  ? K extends keyof T
    ? Omit<T, K> & { [P in K]: SetPath<T[K], Rest,
Value> }
      : never
    : never;

function set<T, P extends Paths<T>, V extends 
PathValue<T, P>>(
  obj: T,
  path: P,
  value: V
): SetPath<T, P, V> {
  const keys = (path as string).split('.');
  const result = JSON.parse(JSON.stringify(obj));
  let current = result;

  for (let i = 0; i < keys.length - 1; i++) {
    current = current[keys[i]];
  }

  current[keys[keys.length - 1]] = value;
  return result;
}

// Usage
interface Config {
  server: {
    host: string;
    port: number;
    ssl: {
      enabled: boolean;
      cert: string;
    };
  };
  database: {
    host: string;
    port: number;
  };
}

const config: Config = {
  server: {
    host: 'localhost',
    port: 3000,
    ssl: {
      enabled: true,
      cert: '/path/to/cert',
    },
  },
  database: {
    host: 'localhost',
    port: 5432,
  },
};

// Type-safe access
const host = get(config, 'server.host'); // string
const sslEnabled = get(config, 'server.ssl.enabled'); 
// boolean
const dbPort = get(config, 'database.port'); // number

// get(config, 'server.invalid'); // Error: Type error
// get(config, 'server.port.invalid'); // Error: Type 
error

// Type-safe update
```

```
const updated = set(config, 'server.port', 4000); // OK
// const invalid = set(config, 'server.port',
'invalid'); // Error: Type mismatch

// Advanced: Array path support
type ArrayPath<T> = T extends any[]
  ? `${number}` | `${number}.${Paths<T[number]>}`
  : never;

type PathsWithArray<T, Prefix extends string = ''> = T
extends any[]
  ? ArrayPath<T>
  : T extends object
  ? {
      [K in keyof T]: K extends string
        ? T[K] extends any[]
          ?
            | `${Prefix}${K}`
            | `${Prefix}${K}.${ArrayPath<T[K]>}`
          : T[K] extends object
          ?
            | `${Prefix}${K}`
            | PathsWithArray<T[K], `${Prefix}${K}.`>
          : `${Prefix}${K}`
        : never;
    }[keyof T]
  : never;

interface User {
  id: string;
  profile: {
    name: string;
    contacts: Array<{
      type: string;
      value: string;
    }>;
  };
}

const user: User = {
  id: '123',
  profile: {
    name: 'John',
    contacts: [
      { type: 'email', value: 'john@example.com' },
    ],
  },
};

// Valid paths with array support
type UserPaths = PathsWithArray<User>;
// Results in: 'id' | 'profile' | 'profile.name' |
'profile.contacts' |
//             'profile.contacts.0' |
'profile.contacts.0.type' | ...
```

Key Points:

- Recursive types enable complex type-level computations
- Template literal types build dynamic string types
- `infer` keyword extracts parts of string literal types
- Type-level path validation prevents runtime errors

- Combines mapped types, conditional types, and template literals

⬆ Back to Top

Question 15: Advanced Async Patterns and Type Safety

Question: Create a type-safe async queue with concurrency control and proper error handling using TypeScript generics.

👉 Jump to Answer

Answer:

References:

- TypeScript Async/Await
- Promise Types
- Generic Constraints with Promises

```
/**
 * Type-safe async task queue with concurrency control
 * Demonstrates advanced TypeScript patterns with 
promises and generics
 */

// Task is a function that returns a promise
type Task<T> = () => Promise<T>;
```

```
// Configuration options for the queue
type QueueOptions = {
  concurrency?: number;  // Max number of tasks running
simultaneously
  timeout?: number;      // Max time for a task (0 = no
timeout)
  retries?: number;      // Number of retry attempts on
failure
};

// Discriminated union for task results (similar to
Promise.allSettled)
type TaskResult<T> =
  | { status: 'fulfilled'; value: T }      // Success
case with value
  | { status: 'rejected'; reason: Error }; // Failure
case with error

/**
 * AsyncQueue - Manages concurrent execution of async
tasks
 */
class AsyncQueue {
  // Internal queue storing pending tasks with their
promise resolvers
  private queue: Array<{
    task: Task<any>;                    // The actual
task to execute
    resolve: (value: any) => void;      // Promise
resolver for success
    reject: (reason: any) => void;      // Promise
rejector for failure
```

```
  }> = [];

  private running = 0;          // Number of currently 
running tasks
  private concurrency: number;  // Max concurrent tasks
  private timeout: number;      // Task timeout in ms
  private retries: number;      // Retry attempts

  constructor(options: QueueOptions = {}) {
    // Use nullish coalescing to provide defaults
    this.concurrency = options.concurrency ?? 1;
    this.timeout = options.timeout ?? 0;
    this.retries = options.retries ?? 0;
  }

  async add<T>(task: Task<T>): Promise<T> {
    return new Promise<T>((resolve, reject) => {
      this.queue.push({ task, resolve, reject });
      this.process();
    });
  }

  async addBatch<T extends readonly Task<any>[]>(
    tasks: T
  ): Promise<{
    [K in keyof T]: T[K] extends Task<infer R> ? 
TaskResult<R> : never;
  }> {
    const results = await Promise.allSettled(
      tasks.map((task) => this.add(task))
    );
    return results.map((result) =>
      result.status === 'fulfilled'
        ? { status: 'fulfilled' as const, value: 
result.value }
        : { status: 'rejected' as const, reason: 
result.reason }
    ) as any;
  }

  private async process(): Promise<void> {
    if (this.running >= this.concurrency || 
this.queue.length === 0) {
      return;
    }

    const item = this.queue.shift();
    if (!item) return;

    this.running++;

    try {
      const result = await 
this.executeWithRetry(item.task);
      item.resolve(result);
    } catch (error) {
      item.reject(error);
    } finally {
      this.running--;
      this.process();
    }
  }
  private async executeWithRetry<T>(
    task: Task<T>,
    attempt = 0
  ): Promise<T> {
    try {
      if (this.timeout > 0) {
        return await this.withTimeout(task(), 
this.timeout);
      }
      return await task();
    } catch (error) {
      if (attempt < this.retries) {
        await this.delay(Math.pow(2, attempt) * 100);
        return this.executeWithRetry(task, attempt + 
1);
      }
      throw error;
    }
  }

  private withTimeout<T>(promise: Promise<T>, timeout: 
number): Promise<T> {
    return Promise.race([
      promise,
      new Promise<T>((_, reject) =>
        setTimeout(() => reject(new Error('Task 
timeout')), timeout)
      ),
    ]);
  }

  private delay(ms: number): Promise<void> {
    return new Promise((resolve) => setTimeout(resolve, 
ms));
  }

  async waitForAll(): Promise<void> {
    while (this.queue.length > 0 || this.running > 0) {
      await this.delay(10);
    }
  }

  clear(): void {
    this.queue = [];
  }

  get size(): number {
    return this.queue.length;
  }

  get activeCount(): number {
    return this.running;
  }
}

// Advanced: Type-safe event-driven queue
type QueueEvents<T> = {
  taskStart: (task: Task<T>) => void;
  taskComplete: (result: T) => void;
  taskError: (error: Error) => void;
  queueEmpty: () => void;
};

class EventDrivenQueue<T> {
  private queue: AsyncQueue;
  private listeners: Partial<QueueEvents<T>> = {};

  constructor(options?: QueueOptions) {
    this.queue = new AsyncQueue(options);
  }

  on<E extends keyof QueueEvents<T>>(
    event: E,
    listener: QueueEvents<T>[E]
  ): void {
    this.listeners[event] = listener;
  }

  async add(task: Task<T>): Promise<T> {
    this.listeners.taskStart?.(task);

    try {
      const result = await this.queue.add(task);
      this.listeners.taskComplete?.(result);

      if (this.queue.size === 0 && 
this.queue.activeCount === 0) {
        this.listeners.queueEmpty?.();
      }

      return result;
    } catch (error) {
      this.listeners.taskError?.(error as Error);
      throw error;
    }
  }
```

```
}

// Usage
const queue = new AsyncQueue({ concurrency: 3, timeout: 
5000, retries: 2 });

// Single task
const result = await queue.add(async () => {
  const response = await 
fetch('https://api.example.com/data');
  return response.json();
});

// Batch tasks with type inference
const tasks = [
  async () => 'string result',
  async () => 42,
  async () => ({ data: 'object' }),
] as const;

const results = await queue.addBatch(tasks);
// Type: [
//   TaskResult<string>,
//   TaskResult<number>,
//   TaskResult<{ data: string }>
// ]

results.forEach((result) => {
  if (result.status === 'fulfilled') {
    console.log(result.value);
  } else {
    console.error(result.reason);
    }
});

// Event-driven queue
const eventQueue = new EventDrivenQueue<string>({ 
concurrency: 2 });

eventQueue.on('taskStart', (task) => console.log('Task 
started'));
eventQueue.on('taskComplete', (result) => 
console.log('Task completed:', result));
eventQueue.on('taskError', (error) => 
console.error('Task failed:', error));
eventQueue.on('queueEmpty', () => console.log('Queue is 
empty'));

await eventQueue.add(async () => 'Hello World');
```

Key Points:

- Generic types preserve type information through async operations
- `Promise.allSettled` provides type-safe batch processing
- Discriminated unions for result types
- Type inference works with `as const` for tuple types
- Event emitter pattern with type-safe event names and handlers

⬆ Back to Top

Question 16: Abstract Classes vs Interfaces

Question: Explain the differences between abstract classes and interfaces in TypeScript. When should you use each? Provide examples showing their unique capabilities.

👉 Jump to Answer

Answer:

References:

- TypeScript Interfaces
- Abstract Classes
- When to Use Interfaces vs Types

Key Differences:

```
// Interface - contract only
interface Logger {
  log(message: string): void;
  error(message: string): void;
}

// Abstract class - contract + implementation
abstract class BaseLogger {
  protected prefix: string;

  constructor(prefix: string) {
    this.prefix = prefix;
  }

  // Abstract method - must be implemented
  abstract log(message: string): void;

  // Concrete method - can be inherited
  protected formatMessage(message: string): string {
    return `[${this.prefix}] ${message}`;
  }

  // Concrete method with default behavior
  error(message: string): void {
    this.log(`ERROR: ${message}`);
  }
}

// Implementing interface
class ConsoleLogger implements Logger {
  log(message: string): void {
    console.log(message);
  }

  error(message: string): void {
    console.error(message);
  }
}

// Extending abstract class
class FileLogger extends BaseLogger {
  private filePath: string;

  constructor(prefix: string, filePath: string) {
    super(prefix);
    this.filePath = filePath;
  }
  log(message: string): void {
    const formatted = this.formatMessage(message);
    // Write to file logic
    console.log(`Writing to ${this.filePath}: 
${formatted}`);
  }
}

// Multiple interface implementation
interface Serializable {
  serialize(): string;
}

interface Comparable<T> {
  compareTo(other: T): number;
}

class User implements Serializable, Comparable<User> {
  constructor(public id: string, public name: string) 
{}

  serialize(): string {
    return JSON.stringify(this);
  }

  compareTo(other: User): number {
    return this.name.localeCompare(other.name);
  }
}

// Cannot extend multiple abstract classes
```

```
abstract class Entity {
  abstract save(): void;
}

abstract class Timestamped {
  createdAt: Date = new Date();
  abstract touch(): void;
}

// Error: Can only extend one class
// class Post extends Entity, Timestamped {} // Error

// Solution: Mix interfaces with abstract class
interface ITimestamped {
  createdAt: Date;
  touch(): void;
}

abstract class BaseEntity implements ITimestamped {
  createdAt: Date = new Date();

  abstract save(): void;

  touch(): void {
    this.createdAt = new Date();
  }
}

class Post extends BaseEntity {
  constructor(public title: string) {
    super();
  }
  save(): void {
    console.log('Saving post:', this.title);
  }
}
```

When to Use Each:

```
// Use Interface when:
// 1. Defining data shape
interface UserDTO {
  id: string;
  email: string;
  name: string;
}

// 2. Multiple inheritance needed
interface Readable {
  read(): string;
}

interface Writable {
  write(data: string): void;
}

interface ReadWrite extends Readable, Writable {}

// 3. Declaration merging needed
interface Window {
  customProperty: string;
}
```

```
// 4. Structural typing (duck typing)
interface Point {
  x: number;
  y: number;
}

function distance(p1: Point, p2: Point): number {
  return Math.sqrt((p2.x - p1.x) ** 2 + (p2.y - p1.y)
** 2);
}

// Any object with x and y works
distance({ x: 0, y: 0 }, { x: 3, y: 4 }); // OK

// Use Abstract Class when:
// 1. Shared implementation needed
abstract class HttpClient {
  protected baseUrl: string;

  constructor(baseUrl: string) {
    this.baseUrl = baseUrl;
  }

  abstract request<T>(endpoint: string): Promise<T>;

  // Shared implementation
  protected buildUrl(endpoint: string): string {
    return `${this.baseUrl}${endpoint}`;
  }

  // Template method pattern
  async get<T>(endpoint: string): Promise<T> {
    return this.request<T>(endpoint);
  }
}

class RestClient extends HttpClient {
  async request<T>(endpoint: string): Promise<T> {
    const url = this.buildUrl(endpoint);
    const response = await fetch(url);
    return response.json();
  }
}

// 2. Constructor logic needed
abstract class Component {
  protected id: string;

  constructor() {
    this.id = Math.random().toString(36);
  }

  abstract render(): string;
}

// 3. Access modifiers needed
abstract class Repository<T> {
  protected items: T[] = [];

  abstract findById(id: string): T | undefined;

  // Protected method only for subclasses
  protected addItem(item: T): void {
    this.items.push(item);
  }
}

// 4. Mix of required and optional methods
abstract class Validator<T> {
  // Required
  abstract validate(value: T): boolean;

  // Optional (default implementation)
  validateAsync(value: T): Promise<boolean> {
    return Promise.resolve(this.validate(value));
  }
}
```

Comparison Table:

Feature	Interface	Abstract Class
Multiple inheritance	✅ Yes	❌ No
Implementation	❌ No	✅ Yes (partial)
Constructor	❌ No	✅ Yes
Access modifiers	❌ No	✅ Yes
Declaration merging	✅ Yes	❌ No
Compiled output	❌ No (erased)	✅ Yes (JS class)

Feature	Interface	Abstract Class
Structural typing	✅ Yes	❌ No (nominal)

Key Points:

- Interfaces are compile-time only contracts
- Abstract classes provide runtime inheritance structure
- Interfaces enable multiple inheritance
- Abstract classes enable code reuse through implementation
- Use interfaces for pure contracts, abstract classes for shared behavior

⬆ Back to Top

Question 17: Nominal Typing in TypeScript

Question: TypeScript uses structural typing. How can you implement nominal (name-based) typing? Provide examples using branded types.

👉 Jump to Answer

Answer:

References:

- Structural vs Nominal Typing
- Branded Types Pattern
- Newtype Pattern in TypeScript

Problem with Structural Typing:

```
type UserId = string;
type OrderId = string;

function getUser(id: UserId): void {
  console.log('Getting user:', id);
}

const orderId: OrderId = 'order-123';
getUser(orderId); // No error! But semantically wrong
```

Solution 1: Branded Types with Intersection

```
/**
 * Branded Types Pattern - Add compile-time type safety 
without runtime overhead
 * Creates nominal types from structural types
 */

// Generic brand utility - adds a phantom type property
// K is the base type (e.g., string), T is the brand 
identifier
type Brand<K, T> = K & { __brand: T };
// The __brand property doesn't exist at runtime - it's 
purely for type checking

// Create distinct branded types from the same base 
type (string)
type UserId = Brand<string, 'UserId'>;      // string + 
UserId brand
```

```
type OrderId = Brand<string, 'OrderId'>;  // string + OrderId brand
type Email = Brand<string, 'Email'>;      // string + Email brand

// Even though all three are strings at runtime, TypeScript treats them as incompatible

// Constructor functions - the only way to create branded types
// This enforces that creation goes through validation/processing
function createUserId(id: string): UserId {
  // Could add validation here
  return id as UserId;  // Type assertion to add the brand
}

function createOrderId(id: string): OrderId {
  // Could add validation here
  return id as OrderId;  // Type assertion to add the brand
}

function createEmail(email: string): Email {
  // Validation before branding
  if (!email.includes('@')) {
    throw new Error('Invalid email');
  }
  return email as Email;  // Only branded if valid
}
```

```
// Type-safe functions that only accept specific 
branded types
function getUser(id: UserId): void {
  console.log('Getting user:', id);
  // id is guaranteed to be created via createUserId
}

function getOrder(id: OrderId): void {
  console.log('Getting order:', id);
  // id is guaranteed to be created via createOrderId
}

// Usage examples
const userId = createUserId('user-123');      // ✅ 
Creates UserId
const orderId = createOrderId('order-456');   // ✅ 
Creates OrderId

getUser(userId);    // ✅ OK - correct brand
// getUser(orderId);   // ❌ Error: Type 'OrderId' is 
not assignable to type 'UserId'

// This prevents accidental misuse:
// getUser('user-123');   // ❌ Error: string is not 
assignable to UserId
// Must use constructor:
getUser(createUserId('user-123'));   // ✅ OK

// Why this works:
// - At runtime: all are just strings, no overhead
// - At compile-time: TypeScript sees them as different 
types
```

```
// - Cannot accidentally mix them up
// - Forces use of constructor functions (good for 
validation)
```

Solution 2: Opaque Types with Unique Symbol

```
declare const __brand: unique symbol;

type Brand<T, TBrand> = T & { [__brand]: TBrand };

type USD = Brand<number, 'USD'>;
type EUR = Brand<number, 'EUR'>;
type GBP = Brand<number, 'GBP'>;

// Constructor functions with validation
function usd(amount: number): USD {
  if (amount < 0) {
    throw new Error('Amount cannot be negative');
  }
  return amount as USD;
}

function eur(amount: number): EUR {
  if (amount < 0) {
    throw new Error('Amount cannot be negative');
  }
  return amount as EUR;
}

function gbp(amount: number): GBP {
  if (amount < 0) {
    throw new Error('Amount cannot be negative');
  }
  return amount as GBP;
}

// Type-safe currency operations
function addUSD(a: USD, b: USD): USD {
  return (a + b) as USD;
}

function convertUSDToEUR(amount: USD, rate: number): 
EUR {
  return (amount * rate) as EUR;
}

// Usage
const price1 = usd(100);
const price2 = usd(50);
const totalUSD = addUSD(price1, price2); // OK

const priceEUR = eur(75);
// const invalid = addUSD(price1, priceEUR); // Error: 
Type 'EUR' is not assignable to type 'USD'

const convertedEUR = convertUSDToEUR(price1, 0.85); // 
OK
```

Solution 3: Class-Based Nominal Types

```
class UserId {
  private constructor(private readonly value: string) 
```

```
{}

  static create(value: string): UserId {
    if (!value) {
      throw new Error('UserId cannot be empty');
    }
    return new UserId(value);
  }

  toString(): string {
    return this.value;
  }

  equals(other: UserId): boolean {
    return this.value === other.value;
  }
}

class OrderId {
  private constructor(private readonly value: string) 
{}

  static create(value: string): OrderId {
    if (!value.startsWith('ORD-')) {
      throw new Error('OrderId must start with ORD-');
    }
    return new OrderId(value);
  }

  toString(): string {
    return this.value;
  }
```

```
}

function processUser(id: UserId): void {
  console.log('Processing user:', id.toString());
}

const userId = UserId.create('user-123');
const orderId = OrderId.create('ORD-456');

processUser(userId); // OK
// processUser(orderId); // Error: Type 'OrderId' is 
not assignable to type 'UserId'
```

Advanced: Refined Types with Runtime Validation

```
declare const validatedBrand: unique symbol;

type Validated<T, TBrand> = T & { [validatedBrand]: 
TBrand };

type PositiveNumber = Validated<number, 
'PositiveNumber'>;
type NonEmptyString = Validated<string, 
'NonEmptyString'>;
type SafeHTML = Validated<string, 'SafeHTML'>;

function createPositiveNumber(value: number): 
PositiveNumber {
  if (value <= 0) {
    throw new Error('Number must be positive');
  }
  return value as PositiveNumber;
}

function createNonEmptyString(value: string): 
NonEmptyString {
  if (value.trim().length === 0) {
    throw new Error('String cannot be empty');
  }
  return value as NonEmptyString;
}

function sanitizeHTML(html: string): SafeHTML {
  // Sanitization logic
  const sanitized = html
    .replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*
<\/script>/gi, '')
    .replace(/on\w+="[^"]*"/g, '');
  return sanitized as SafeHTML;
}

// Usage
function calculateArea(width: PositiveNumber, height: 
PositiveNumber): PositiveNumber {
  return createPositiveNumber(width * height);
}

function renderHTML(html: SafeHTML): void {
  document.body.innerHTML = html;
}

const width = createPositiveNumber(10);
const height = createPositiveNumber(20);
const area = calculateArea(width, height); // OK

// const invalid = calculateArea(width, -5); // Error: 
Type 'number' is not assignable
```

Key Points:

- Branded types add compile-time type safety
- Unique symbols prevent accidental type compatibility
- Constructor functions enforce creation patterns
- Class-based approach provides runtime protection
- Combines with validation for refined types
- No runtime overhead for brand-based approaches

⬆ Back to Top

Question 18: TypeScript Performance in Large Codebases

Question: What are the common TypeScript performance bottlenecks in large applications? How do you diagnose and fix them?

👉 Jump to Answer

Answer:

References:

- TypeScript Performance Wiki
- Analyzing Build Performance
- TSConfig Reference

Diagnosis Tools:

```
# Generate build performance report
tsc --extendedDiagnostics

# Generate trace for analysis
tsc --generateTrace ./trace

# Analyze with @typescript/analyze-trace
npx @typescript/analyze-trace ./trace
```

Common Bottlenecks:

1. Complex Type Inference

```
// Slow - deep inference with unions
type SlowUnion = A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P;

interface A { type: 'a'; data: string; }
interface B { type: 'b'; data: number; }
// ... many more

function process(value: SlowUnion) {
  // TypeScript must check all 16 types
  if (value.type === 'a') {
    return value.data.toUpperCase();
  }
  // ...
}

// Fast - use discriminated unions with indexed access
type FastLookup = {
  a: { data: string };
  b: { data: number };
  // ...
};

type FastUnion = {
  [K in keyof FastLookup]: { type: K } & FastLookup[K];
}[keyof FastLookup];
```

2. Recursive Type Depth

```
// Slow - unlimited recursion
type DeepPartial<T> = {
  [K in keyof T]?: T[K] extends object ? 
DeepPartial<T[K]> : T[K];
};

// Fast - limited recursion depth
type DeepPartialLimited<T, Depth extends number = 5> = 
Depth extends 0
  ? T
  : {
      [K in keyof T]?: T[K] extends object
        ? DeepPartialLimited<T[K], Prev<Depth>>
        : T[K];
```

```
    };

type Prev<T extends number> = [-1, 0, 1, 2, 3, 4, 5, 6,
7, 8, 9, 10][T];
```

3. Type Instantiation

```
// Slow - creates many type instantiations
function map<T, U>(arr: T[], fn: (item: T) => U): U[] {
  return arr.map(fn);
}

// Fast - use constraints to reduce instantiations
function mapConstrained<T extends string | number, U
extends string | number>(
  arr: T[],
  fn: (item: T) => U
): U[] {
  return arr.map(fn);
}
```

4. Module Organization

```
// Slow - barrel exports force checking all modules
// index.ts
export * from './module1';
export * from './module2';
// ... 100 more modules

// Fast - explicit exports
// index.ts
```

```
export { SpecificType1, SpecificType2 } from 
'./module1';
export { SpecificType3 } from './module2';

// Or import directly
// import { SpecificType1 } from 
'./components/module1';
```

5. Type Checking Configuration

```
// tsconfig.json - Optimized for large codebases
{
  "compilerOptions": {
    // Skip type checking of node_modules
    "skipLibCheck": true,

    // Faster module resolution
    "moduleResolution": "bundler",

    // Isolated modules for parallel checking
    "isolatedModules": true,

    // Incremental builds
    "incremental": true,
    "tsBuildInfoFile": "./.tsbuildinfo",

    // Composite projects for monorepos
    "composite": true,

    // Disable unused checks if not needed
    "noUnusedLocals": false,
    "noUnusedParameters": false,

    // Exclude large files
    "exclude": [
      "node_modules",
      "dist",
      "**/*.spec.ts",
      "**/*.test.ts"
    ]
  }
}
```

6. Type Assertions vs Type Guards

```
// Slow - type guard called repeatedly
function isString(value: unknown): value is string {
  return typeof value === 'string';
}

function processItems(items: unknown[]) {
  items.forEach(item => {
    if (isString(item)) { // Called N times
      console.log(item.toUpperCase());
    }
  });
}

// Fast - filter once with type assertion
function processItemsFast(items: unknown[]) {
  const strings = items.filter((item): item is string
=>
```

```
    typeof item === 'string'
  );
  
  strings.forEach(str => {
    console.log(str.toUpperCase());
  });
}
```

7. Avoid `any` Type Propagation

```
// Slow - any propagates through code
function processData(data: any) {
  return data.map((item: any) => item.value);
}

// Fast - use generics
function processDataTyped<T extends { value: any }>
(data: T[]) {
  return data.map(item => item.value);
}

// Or unknown with type guards
function processDataSafe(data: unknown) {
  if (!Array.isArray(data)) {
    throw new Error('Expected array');
  }
  
  return data.map(item => {
    if (typeof item === 'object' && item !== null && 
'value' in item) {
      return item.value;
    }
    throw new Error('Invalid item');
  });
}
```

Performance Best Practices:

```
// 1. Use interfaces over type aliases for objects
interface User { // Faster
  id: string;
  name: string;
}

type UserType = { // Slower for complex types
  id: string;
  name: string;
};

// 2. Prefer const assertions over explicit types
const config = {
  apiUrl: 'https://api.example.com',
  timeout: 5000,
} as const; // Inferred, no extra type checking

// 3. Use type imports
import type { User } from './types'; // Erased at 
compile time

// 4. Limit union size
// Instead of: type Status = 'pending' | 'processing' | 
... (50 literals)
```

```
type Status = `${StatusGroup}_${StatusType}`;
type StatusGroup = 'order' | 'payment' | 'shipping';
type StatusType = 'pending' | 'completed' | 'failed';

// 5. Cache complex type computations
type ExpensiveType<T> = /* complex computation */;

// Don't repeat:
// const a: ExpensiveType<MyType> = ...;
// const b: ExpensiveType<MyType> = ...;

// Instead:
type CachedExpensive = ExpensiveType<MyType>;
const a: CachedExpensive = /* ... */;
const b: CachedExpensive = /* ... */;
```

Key Points:

- Use `--extendedDiagnostics` to identify bottlenecks
- Limit type recursion depth
- Use project references for monorepos
- Avoid barrel exports in large projects
- Use `skipLibCheck` to skip node_modules type checking
- Cache complex type computations
- Use interfaces for object types (better caching)

⬆ Back to Top

Question 19: TypeScript 5.x New Features

Question: What are the most significant features introduced in TypeScript 5.0+? Provide practical examples of const type parameters, decorators, and the satisfies operator.

👉 Jump to Answer

Answer:

References:

- TypeScript 5.0 Release Notes
- TypeScript 5.1 Release Notes
- TypeScript 5.2 Release Notes
- Satisfies Operator

1. Const Type Parameters (TypeScript 5.0)

```
// Before: Type widening issues
function createTuple<T>(values: readonly T[]) {
  return values;
}

const result1 = createTuple(['a', 'b', 'c']);
// Type: string[] - too wide!

// After: Const type parameters
function createTupleConst<const T>(values: readonly 
T[]) {
  return values;
}

const result2 = createTupleConst(['a', 'b', 'c']);
// Type: readonly ["a", "b", "c"] - exact!

// Practical example: Route definitions
function defineRoutes<const T extends readonly Route[]>
(routes: T) {
  return routes;
}

interface Route {
  path: string;
  method: 'GET' | 'POST' | 'PUT' | 'DELETE';
}

const routes = defineRoutes([
  { path: '/users', method: 'GET' },
  { path: '/users/:id', method: 'GET' },
  { path: '/users', method: 'POST' },
] as const);

// Type: readonly [
//   { path: "/users"; method: "GET" },
//   { path: "/users/:id"; method: "GET" },
//   { path: "/users"; method: "POST" }
// ]

type RoutePaths = typeof routes[number]['path'];
// Type: "/users" | "/users/:id"
```

2. Decorators (TypeScript 5.0 - Stage 3)

```
// New decorator syntax (no experimentalDecorators 
needed)
function log<T extends { new (...args: any[]): {} }>(
  target: T,
  context: ClassDecoratorContext
) {
  return class extends target {
    constructor(...args: any[]) {
      super(...args);
      console.log(`Creating instance of 
${context.name}`);
    }
  };
}

function logMethod(
  target: Function,
  context: ClassMethodDecoratorContext
) {
  const methodName = String(context.name);
  
  return function (this: any, ...args: any[]) {
    console.log(`Calling ${methodName} with:`, args);
    const result = target.call(this, ...args);
    console.log(`${methodName} returned:`, result);
    return result;
  };
}

@log
```

```
class UserService {
  @logMethod
  getUser(id: string) {
    return { id, name: 'John' };
  }
}

// Auto-accessor decorators
function validate(target: any, context: 
ClassAccessorDecoratorContext) {
  return {
    get() {
      return target.get.call(this);
    },
    set(value: any) {
      if (typeof value !== 'string' || value.length < 
3) {
        throw new Error('Invalid value');
      }
      target.set.call(this, value);
    },
  };
}

class User {
  @validate
  accessor name: string = '';
}
```

3. Satisfies Operator (TypeScript 4.9+)

```typescript
/**
 * Satisfies Operator - Validates types without widening them
 * Best of both worlds: type checking + precise inference
 */

// Define allowed color formats
type Colors = 'red' | 'green' | 'blue';

// ❌ Problem: Type annotation loses specific types
const colors1: Record<Colors, string | [number, number, number]> = {
  red: '#FF0000',              // Known to be string, but...
  green: [0, 255, 0],          // Known to be tuple, but...
  blue: '#0000FF',             // Known to be string, but...
};

// TypeScript forgets the specific types!
// colors1.red.toUpperCase();
// ❌ Error: Property 'toUpperCase' does not exist on type 'string | [number, number, number]'
// Even though we can see red is a string, TypeScript can't!

// ✅ Solution: satisfies operator
const colors2 = {
  red: '#FF0000',
  green: [0, 255, 0],
  blue: '#0000FF',
} satisfies Record<Colors, string | [number, number, 
number]>;
// Validates structure but preserves precise types

// Now TypeScript knows the exact type of each 
property!
colors2.red.toUpperCase();    // ✅ OK! Type is string 
(not string | tuple)
colors2.green[0];             // ✅ OK! Type is 
[number, number, number]
colors2.blue.toLowerCase();   // ✅ OK! Type is string

// Comparison:
// - Without satisfies: All values typed as `string | 
[number, number, number]`
// - With satisfies: Each value keeps its specific type 
(string or tuple)

// Also catches errors:
// const invalid = {
//   red: '#FF0000',
//   green: [0, 255, 0],
//   purple: '#800080',  // ❌ Error: purple is not in 
Colors union
// } satisfies Record<Colors, string | [number, number, 
number]>;

// Practical example: API routes
type Route = {
  path: string;
  method: 'GET' | 'POST';
  handler: (req: any) => any;
};

const apiRoutes = {
  getUsers: {
    path: '/users',
    method: 'GET',
    handler: (req) => [{ id: 1, name: 'John' }],
  },
  createUser: {
    path: '/users',
    method: 'POST',
    handler: (req) => ({ id: 2, name: req.body.name }),
  },
} satisfies Record<string, Route>;

// Type checking enforced, but specific types preserved
apiRoutes.getUsers.handler({}); // Returns: { id: 
number; name: string; }[]
apiRoutes.createUser.path; // Type: "/users"
```

4. Supporting Paths in tsconfig.json (TypeScript 5.0)

```
{
  "extends": "@tsconfig/node18/tsconfig.json",
  "compilerOptions": {
    "paths": {
      "@/*": ["./src/*"],
      "@components/*": ["./src/components/*"],
      "@utils/*": ["./src/utils/*"]
    }
  }
}
```

5. Better Inference for Type Parameters (TypeScript 5.1)

```
// Before: Required explicit type parameter
function createMap<K, V>(entries: [K, V][]) {
  return new Map(entries);
}

const map1 = createMap<string, number>([['a', 1], ['b', 
2]]);

// After: Inferred from usage
function createMapImproved<K, V>(entries: [K, V][]) {
  return new Map(entries);
}

const map2 = createMapImproved([
  ['a' as const, 1],
  ['b' as const, 2],
]);
// Type: Map<"a" | "b", number>
```

6. Enum Improvements (TypeScript 5.0)

```
// Before: Non-const enum
enum Direction {
  Up,
  Down,
  Left,
  Right,
}

// After: Const enum with preserveConstEnums
const enum FastDirection {
  Up,
  Down,
  Left,
  Right,
}

// Completely inlined at compile time
const direction = FastDirection.Up; // Compiles to: 
const direction = 0;
```

7. Using Declaration in for-await-of (TypeScript 5.2)

```
// Automatically disposes resources
class FileHandle {
  constructor(private path: string) {}

  [Symbol.dispose]() {
    console.log(`Closing file: ${this.path}`);
  }

  read() {
    return 'file contents';
  }
}
```

```
function processFiles() {
  using file = new FileHandle('/path/to/file');
  console.log(file.read());
  // Automatically calls Symbol.dispose when leaving
scope
}
```

Key Points:

- Const type parameters preserve literal types
- New decorator syntax is simpler and standardized
- `satisfies` operator combines validation with precise types
- TypeScript 5.x improves inference and performance
- Use `satisfies` to validate without losing type information
- New decorators work without `experimentalDecorators`

⬆ Back to Top

Question 20: TypeScript Anti-Patterns and Best Practices

Question: What are the most common TypeScript anti-patterns you should avoid in production code? Provide examples of problematic code and their solutions.

👉 Jump to Answer

Answer:

References:

- [TypeScript Do's and Don'ts]
- [TypeScript Best Practices]
- [Strict Mode in TypeScript]
- [TypeScript Style Guide]

Anti-Pattern 1: Excessive Use of `any`

```
// ❌ Bad - `any` disables type checking
function processData(data: any): any {
  return data.map((item: any) => item.value);
  // Problems:
  // - No compile-time safety
  // - data might not be an array (runtime error)
  // - item might not have a value property
  // - Return type is unknown to callers
}

// ✅ Good - Generic with constraints
function processData<T extends { value: unknown }>
(data: T[]): unknown[] {
  return data.map(item => item.value);
  // Benefits:
  // - data is guaranteed to be an array
  // - Each item is guaranteed to have a value property
  // - Type inference works for callers
  // - Still flexible (works with any object that has
value)
}

// ✅ Better - Specific typing when possible
interface DataItem {
  value: string;
  metadata?: Record<string, any>;
}

function processDataTyped(data: DataItem[]): string[] {
  return data.map(item => item.value);
  // Benefits:
  // - Maximum type safety
  // - Autocomplete for DataItem properties
  // - Clear contract for consumers
  // - Easier to refactor
}

// Real-world example:
// ❌ Don't do this
const response: any = await fetch('/api/users');
const users: any = await response.json();

// ✅ Do this instead
interface User {
  id: string;
  name: string;
  email: string;
}

const response = await fetch('/api/users');
const users: User[] = await response.json();  // Type assertion with validation
```

Anti-Pattern 2: Type Assertions Over Type Guards

```
// ❌ Bad
function getUser(data: unknown) {
  const user = data as User;
  return user.name.toUpperCase();
}

// ✅ Good
function getUserSafe(data: unknown): string {
  if (
    typeof data === 'object' &&
    data !== null &&
    'name' in data &&
    typeof data.name === 'string'
  ) {
    return data.name.toUpperCase();
  }
  throw new Error('Invalid user data');
}

// ✅ Better with type guard
function isUser(data: unknown): data is User {
  return (
    typeof data === 'object' &&
    data !== null &&
    'id' in data &&
    'name' in data &&
    typeof (data as any).id === 'string' &&
    typeof (data as any).name === 'string'
  );
}

function getUserWithGuard(data: unknown): string {
  if (isUser(data)) {
    return data.name.toUpperCase();
  }
  throw new Error('Invalid user data');
}
```

Anti-Pattern 3: Non-Null Assertions

```
// ❌ Bad
function getFirstUser(users: User[] | undefined) {
  return users!.[0]!.name!.toUpperCase();
}

// ✅ Good
function getFirstUserSafe(users: User[] | undefined):
string | undefined {
  return users?.[0]?.name?.toUpperCase();
}

// ✅ Better with proper error handling
function getFirstUserOrThrow(users: User[] |
undefined): string {
  if (!users || users.length === 0) {
    throw new Error('No users found');
  }

  const user = users[0];
  if (!user.name) {
    throw new Error('User has no name');
  }

  return user.name.toUpperCase();
}
```

Anti-Pattern 4: Ignoring Strict Mode

```
// ❌ Bad tsconfig.json
{
  "compilerOptions": {
    "strict": false,
    "noImplicitAny": false,
    "strictNullChecks": false
  }
}

// ✅ Good tsconfig.json
{
  "compilerOptions": {
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "strictBindCallApply": true,
    "strictPropertyInitialization": true,
    "noImplicitThis": true,
    "alwaysStrict": true
  }
}
```

Anti-Pattern 5: Interface Naming with "I" Prefix

```
// ❌ Bad (C# style, not TypeScript convention)
interface IUser {
  id: string;
  name: string;
}

class UserImpl implements IUser {
  constructor(public id: string, public name: string) 
{}
}

// ✅ Good (TypeScript convention)
interface User {
  id: string;
  name: string;
}

class UserModel implements User {
  constructor(public id: string, public name: string) 
{}
}

// Or simply
class User {
  constructor(public id: string, public name: string) 
{}
}
```

Anti-Pattern 6: Overusing Enums

```
// ❌ Bad
enum Status {
  Pending,
  Active,
  Completed,
}

// Issues: Non-const enums generate runtime code
// Can't use as key in mapped types

// ✅ Good - Use union types
type Status = 'pending' | 'active' | 'completed';

// ✅ Or const object with as const
const Status = {
  Pending: 'pending',
  Active: 'active',
  Completed: 'completed',
} as const;

type Status = typeof Status[keyof typeof Status];

// Benefits: Tree-shakeable, works in mapped types, no runtime code
```

Anti-Pattern 7: Not Handling Promise Rejections

```
// ❌ Bad
async function fetchUser(id: string) {
  const response = await fetch(`/api/users/${id}`);
  return response.json();
}

// ✅ Good
async function fetchUserSafe(
  id: string
): Promise<User | { error: string }> {
  try {
    const response = await fetch(`/api/users/${id}`);
    
    if (!response.ok) {
      return { error: `HTTP ${response.status}` };
    }
    
    const data = await response.json();
    
    if (isUser(data)) {
      return data;
    }
    
    return { error: 'Invalid response format' };
  } catch (error) {
    return {
      error: error instanceof Error ? error.message : 
'Unknown error',
    };
  }
}
```

Anti-Pattern 8: Mutable Global State

```
// ❌ Bad
let currentUser: User | null = null;

export function setCurrentUser(user: User) {
  currentUser = user;
}

export function getCurrentUser() {
  return currentUser;
}

// ✅ Good - Use readonly state management
class UserStore {
  private _currentUser: Readonly<User> | null = null;

  get currentUser(): Readonly<User> | null {
    return this._currentUser;
  }

  setCurrentUser(user: User): void {
    this._currentUser = Object.freeze({ ...user });
  }
}

export const userStore = new UserStore();
```

Anti-Pattern 9: Deep Nesting

```
// ❌ Bad
function processOrder(order: Order | null) {
  if (order) {
    if (order.customer) {
      if (order.customer.address) {
        if (order.customer.address.zipCode) {
          return
validateZipCode(order.customer.address.zipCode);
        }
      }
    }
  }
  throw new Error('Invalid order');
}

// ✅ Good - Early returns
function processOrderClean(order: Order | null):
boolean {
  if (!order) {
    throw new Error('Order is required');
  }

  if (!order.customer) {
    throw new Error('Customer is required');
  }

  if (!order.customer.address) {
    throw new Error('Address is required');
  }

  const { zipCode } = order.customer.address;
  if (!zipCode) {
    throw new Error('Zip code is required');
  }
  
  return validateZipCode(zipCode);
}

// ✅ Better - Optional chaining with validation
function processOrderBetter(order: Order | null): 
boolean {
  const zipCode = order?.customer?.address?.zipCode;
  
  if (!zipCode) {
    throw new Error('Invalid order: missing zip code');
  }
  
  return validateZipCode(zipCode);
}
```

Anti-Pattern 10: Ignoring Discriminated Unions

```
// ❌ Bad
interface ApiResponse {
  success: boolean;
  data?: any;
  error?: string;
}

function handleResponse(response: ApiResponse) {
  if (response.success) {
    // data might still be undefined!
    return response.data.value;
  }
}

// ✅ Good - Discriminated union
type ApiResponse<T> =
  | { success: true; data: T }
  | { success: false; error: string };

function handleResponseSafe(response: ApiResponse<{
value: string }>) {
  if (response.success) {
    // TypeScript knows data exists
    return response.data.value;
  } else {
    // TypeScript knows error exists
    console.error(response.error);
  }
}
```

Best Practices Summary:

1. **Always enable strict mode** in `tsconfig.json`
2. **Prefer unknown over any** for truly unknown types
3. **Use type guards** instead of type assertions
4. **Leverage optional chaining** (`?.`) and nullish coalescing (`??`)
5. **Use discriminated unions** for complex type narrowing
6. **Avoid non-const enums** - use union types or const objects
7. **Handle all promise rejections** explicitly

8. **Use readonly** for immutability
9. **Prefer early returns** over deep nesting
10. **Write custom type guards** for complex validation

Key Points:

- Strict mode catches more errors at compile time
- Type guards provide runtime safety
- Discriminated unions enable exhaustive checking
- Optional chaining simplifies null checks
- Avoid type assertions unless absolutely necessary
- Always handle promise rejections explicitly

⬆ Back to Top

Summary

These 20 questions cover advanced TypeScript concepts essential for senior front-end developers:

1. Advanced conditional types
2. Template literal types
3. Variance (covariance, contravariance)
4. Deep generics with constraints
5. Type guards and narrowing
6. Mapped types and key remapping

7. Infer keyword and type extraction
8. Utility types implementation
9. Type inference in generics
10. Module resolution and declaration merging
11. Control flow analysis
12. Performance optimization
13. Decorators and metadata
14. Recursive type-level programming
15. Async patterns with type safety
16. Abstract classes vs interfaces
17. Nominal typing (branded types)
18. Performance in large codebases
19. TypeScript 5.x features
20. Anti-patterns and best practices

Each question demonstrates practical applications and real-world scenarios that senior developers encounter daily.