50 TypeScript Interview Questions with Answers

1. What is TypeScript and how does it differ from JavaScript?

Answer: TypeScript is a strongly-typed superset of JavaScript that adds static typing and other features like interfaces, enums, and generics. JavaScript is dynamically typed. TypeScript code is transpiled to JavaScript for execution in browsers or Node.js.

Frequency: Very High (90%)
Companies: Microsoft, Google, Facebook, Amazon, Airbnb

2. Explain the `any` type in TypeScript.

Answer: The `any` type is a special type that represents any JavaScript value with no constraints. It allows you to opt-out of type checking for variables. While it provides flexibility, it defeats the purpose of TypeScript's type safety.

```
// Using any type
let variable: any = 10;
variable = "string";  // No error
variable = true;      // No error
```

Frequency: High (80%)
Companies: Microsoft, Google, Uber, Twitter

3. What are TypeScript interfaces?

Answer: Interfaces define contracts in your code and provide explicit naming of types. They describe the shape that objects must conform to.

```
interface User {
  id: number;
  name: string;
  email?: string; // Optional property
}

// Object must conform to the interface
const user: User = {
  id: 1,
  name: "John"
};
```

Frequency: Very High (95%)
Companies: Microsoft, Amazon, Facebook, LinkedIn, Slack

4. What is the difference between `interface` and `type` in TypeScript?

Answer: Both define types, but interfaces are extendable and can be merged if declared multiple times (declaration merging). Types are more flexible for complex types and can use unions, mapped types, and conditional types.

```
// Interface
interface Animal {
  name: string;
}
interface Animal { // Valid - merges with previous declaration
  age: number;
}

// Type
type Person = {
  name: string;
};
// type Person = { age: number; } // Error - duplicate identifier
```

Frequency: Very High (85%)
Companies: Microsoft, Google, Facebook, Airbnb, Uber

5. Explain TypeScript generics.

Answer: Generics allow you to create reusable components that work with a variety of types rather than a single one, providing type safety while maintaining flexibility.

```
// Generic function
function identity<T>(arg: T): T {
  return arg;
}

// Usage
const num = identity<number>(5); // Type of 'num' is number
const str = identity("hello");   // Type inference, 'str' is string
```

Frequency: Very High (90%)
Companies: Microsoft, Google, Amazon, Facebook, Netflix

6. What is the `never` type in TypeScript?

Answer: The `never` type represents values that never occur. It's used for functions that always throw exceptions or never return (infinite loops).

```
// Function returning never
function throwError(message: string): never {
  throw new Error(message);
}

// Function with infinite loop
function infiniteLoop(): never {
  while (true) {}
}
```

Frequency: Medium (60%)
Companies: Microsoft, Facebook, Uber, Twitter

7. Explain union and intersection types in TypeScript.

Answer:

- Union types (`A | B`) allow a value to be one of several types
- Intersection types (`A & B`) combine multiple types into one

```
// Union type
type StringOrNumber = string | number;
let variable: StringOrNumber = "hello";
variable = 42; // Also valid

// Intersection type
type Employee = { id: number; name: string };
type Manager = { employees: Employee[] };
type ManagerEmployee = Employee & Manager;
// Must have all properties from both types
```

Frequency: High (75%)
Companies: Microsoft, Google, Amazon, Stripe, Shopify

8. What are TypeScript decorators?

Answer: Decorators are special declarations that can be attached to class declarations, methods, properties, or parameters. They use the form `@expression` , where expression must evaluate to a function that will be called at runtime.

```
// Method decorator
function log(target: any, key: string, descriptor: PropertyDescriptor) {
  const original = descriptor.value;
  descriptor.value = function(...args: any[]) {
    console.log(`Calling ${key} with`, args);
    return original.apply(this, args);
  };
  return descriptor;
}

class Calculator {
  @log
  add(a: number, b: number): number {
    return a + b;
  }
}
```

Frequency: Medium (55%)

Companies: Microsoft, Angular teams, NestJS users, Google

9. What are tuple types in TypeScript?

Answer: Tuples are arrays with a fixed number of elements whose types are known but need not be the same.

```
// Define a tuple type
let tuple: [string, number];
tuple = ["hello", 10]; // OK
// tuple = [10, "hello"]; // Error - incorrect order
// tuple = ["hello", 10, true]; // Error - too many elements
```

Frequency: Medium (65%)

Companies: Microsoft, Google, Airbnb, Shopify

10. Explain the `readonly` modifier in TypeScript.

Answer: The `readonly` modifier prevents properties from being changed after initialization.

```
interface Point {
  readonly x: number;
  readonly y: number;
}

const p: Point = { x: 10, y: 20 };
// p.x = 5; // Error: Cannot assign to 'x' because it is a read-only property
```

Frequency: Medium (60%)

Companies: Microsoft, Facebook, Amazon, Twitter

11. What are index signatures in TypeScript?

Answer: Index signatures allow you to create objects with flexible property names but consistent value types.

```typescript
interface Dictionary<T> {
  [key: string]: T;
}

const stringDict: Dictionary<string> = {
  key1: "value1",
  key2: "value2"
};

// Can add any string-keyed property with string value
stringDict.newKey = "newValue";
```

Frequency: Medium (50%)
Companies: Microsoft, Google, Airbnb, LinkedIn

12. What is the `keyof` operator in TypeScript?

Answer: The `keyof` operator takes an object type and produces a string or numeric literal union of its keys.

```typescript
interface User {
  id: number;
  name: string;
  email: string;
}

// KeyOfUser = "id" | "name" | "email"
type KeyOfUser = keyof User;

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

const user: User = { id: 1, name: "John", email: "john@example.com" };
const userName = getProperty(user, "name"); // Type safe
// const invalid = getProperty(user, "age"); // Error: "age" is not in keyof User
```

Frequency: Medium-High (70%)
Companies: Microsoft, Facebook, Google, Stripe, Shopify

13. What are conditional types in TypeScript?

Answer: Conditional types select one of two possible types based on a condition expressed as a type relationship test.

```
type IsString<T> = T extends string ? true : false;

// Usage
type Yes = IsString<string>; // type is true
type No = IsString<number>;  // type is false

// More practical example
type ArrayElementType<T> = T extends (infer U)[] ? U : never;
type Item = ArrayElementType<string[]>; // Item is string
```

Frequency: Medium (55%)
Companies: Microsoft, Facebook, Google, Uber, Airbnb

14. What is the `unknown` type in TypeScript?

Answer: The `unknown` type is a type-safe alternative to `any`. Variables of type `unknown` can hold any value, but you cannot perform operations on an `unknown` value without first performing a type check.

```
function processValue(val: unknown): string {
  // Can't use val directly
  // return val.toString(); // Error

  // Need to check type first
  if (typeof val === "string") {
    return val.toUpperCase(); // OK
  }
  return String(val);
}
```

Frequency: Medium-High (65%)
Companies: Microsoft, Google, Facebook, Amazon, Stripe

15. What are namespaces in TypeScript?

Answer: Namespaces (previously called "internal modules") are a TypeScript-specific way to organize code. They group related functionality and help prevent naming collisions.

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

  export function calculateDistance(p1: Point, p2: Point): number {
    // Implementation
    return Math.sqrt(Math.pow(p2.x - p1.x, 2) + Math.pow(p2.y - p1.y, 2));
  }
}

// Usage
const point: Geometry.Point = { x: 0, y: 10 };
```

Frequency: Low-Medium (40%)

Companies: Microsoft, Enterprise companies with large codebases

16. What is the `infer` keyword used for in TypeScript?

Answer: The `infer` keyword is used within conditional types to infer (extract) a type from another type. It allows you to capture and use types within a conditional type.

```
// Extract the return type of a function
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;

function greet(): string {
  return "Hello";
}

type GreetReturn = ReturnType<typeof greet>; // string
```

Frequency: Medium (50%)

Companies: Microsoft, Facebook, Google, Stripe, Advanced TypeScript users

17. What are mapped types in TypeScript?

Answer: Mapped types allow you to create new types based on existing ones by transforming properties in some way.

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

type Optional<T> = {
  [K in keyof T]?: T[K];
};

interface User {
  id: number;
  name: string;
}

// All properties are readonly
const readonlyUser: Readonly<User> = { id: 1, name: "John" };
// readonlyUser.id = 2; // Error: Cannot assign to 'id' because it is a read-only prope

// All properties are optional
const partialUser: Optional<User> = { id: 1 }; // name is optional
```

Frequency: Medium-High (65%)

Companies: Microsoft, Facebook, Google, Amazon, Airbnb

18. What are declaration merging in TypeScript?

Answer: Declaration merging is when the compiler merges two or more declarations with the same name into a single definition. This applies mainly to interfaces, namespaces, and enums.

```typescript
// Interface merging
interface Box {
  height: number;
}
interface Box {
  width: number;
}
// Equivalent to:
// interface Box {
//   height: number;
//   width: number;
// }

const box: Box = { height: 5, width: 6 }; // Must have both properties
```

Frequency: Medium (45%)

Companies: Microsoft, Google, Advanced TypeScript projects

19. What are the differences between `abstract class` and `interface` in TypeScript?

Answer:

- Abstract classes can have implementations for some methods, while interfaces cannot
- Classes can implement multiple interfaces but extend only one abstract class
- Abstract classes can have constructors and protected members
- Interfaces are pure types with no implementation details

```typescript
// Abstract class
abstract class Animal {
  protected name: string;

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

  abstract makeSound(): void; // Must be implemented by subclasses

  move(): void {
    console.log(`${this.name} moves`);
  }
}

// Interface
interface Shape {
  area(): number;
  perimeter(): number;
}
```

Frequency: Medium-High (70%)

Companies: Microsoft, Google, Amazon, Facebook, Enterprise applications

20. What is the `Partial<T>` utility type in TypeScript?

Answer: `Partial<T>` is a built-in utility type that makes all properties of a type optional.

```typescript
interface User {
  id: number;
  name: string;
  email: string;
}

// All properties are optional
function updateUser(user: User, updates: Partial<User>): User {
  return { ...user, ...updates };
}

const user: User = { id: 1, name: "John", email: "john@example.com" };
// Only need to provide properties to update
const updatedUser = updateUser(user, { name: "Jack" });
```

Frequency: High (75%)

Companies: Microsoft, Google, Facebook, Amazon, Most TypeScript projects

21. How do enums work in TypeScript?

Answer: Enums allow defining a set of named constants. TypeScript provides both numeric and string-based enums.

```typescript
// Numeric enum (auto-incremented)
enum Direction {
  Up,     // 0
  Down,   // 1
  Left,   // 2
  Right   // 3
}

// String enum
enum MediaTypes {
  JSON = "application/json",
  XML = "application/xml"
}

// Usage
function move(direction: Direction) {
  console.log(`Moving ${Direction[direction]}`);
}

move(Direction.Up); // "Moving Up"
```

Frequency: High (80%)

Companies: Microsoft, Google, Facebook, Amazon, Enterprise applications

22. What are the differences between `extends` and `implements` in TypeScript?

Answer:

- `extends` is used for inheritance between classes or to constrain generic type parameters
- `implements` is used by a class to implement an interface

```typescript
// Using extends for class inheritance
class Animal {
  move() { console.log("Moving"); }
}

class Dog extends Animal {
  bark() { console.log("Woof"); }
}

// Using implements for interface implementation
interface Printable {
  print(): void;
}

class Document implements Printable {
  print() { console.log("Printing document"); }
}

// Using extends with generic constraints
function getLength<T extends { length: number }>(item: T): number {
  return item.length;
}
```

Frequency: High (75%)

Companies: Microsoft, Google, Amazon, Facebook, Twitter

23. What is the purpose of the `Record<K, T>` utility type?

Answer: `Record<K, T>` constructs a type with a set of properties of type `T` with keys of type `K`. It's useful for mapping properties of one type to another.

```typescript
// Create a type with string keys and number values
type NumberRecord = Record<string, number>;

const metrics: NumberRecord = {
  requests: 100,
  errors: 5,
  timeMs: 350
};

// With union type keys
type UserRole = "admin" | "user" | "guest";
const permissions: Record<UserRole, string[]> = {
  admin: ["read", "write", "delete"],
  user: ["read", "write"],
  guest: ["read"]
};
```

Frequency: Medium-High (65%)

Companies: Microsoft, Google, Facebook, Advanced TypeScript users

24. What is the difference between `private` and `#private` fields in TypeScript?

Answer:

- `private` is a TypeScript access modifier enforced only at compile-time
- `#private` is a JavaScript private field (ECMAScript 2020+) enforced at runtime

```
class Person {
  private tsPrivate: string; // TypeScript private - compile time only
  #jsPrivate: string;        // JavaScript private - runtime enforced

  constructor(name: string) {
    this.tsPrivate = name;
    this.#jsPrivate = name;
  }

  greet() {
    console.log(`Hello, ${this.tsPrivate} and ${this.#jsPrivate}`);
  }
}

const person = new Person("John");
// Both show compile-time errors in TypeScript
// person.tsPrivate; // Error
// person.#jsPrivate; // Error

// But at runtime, after compilation:
// person["tsPrivate"] would work (no runtime enforcement)
// person["#jsPrivate"] would fail (runtime enforcement)
```

Frequency: Medium (45%)

Companies: Microsoft, Google, Modern TypeScript codebases

25. What is the `satisfies` operator in TypeScript?

Answer: The `satisfies` operator (introduced in TypeScript 4.9) ensures a value satisfies a type while preserving the specific literal type of the value. It validates an expression against a type without changing the expression's inferred type.

```ts
type RGB = [red: number, green: number, blue: number];
type Color = RGB | string;

const palette = {
  red: "#FF0000",
  green: [0, 255, 0],
  blue: "rgb(0, 0, 255)"
} satisfies Record<string, Color>;

// Type is preserved:
palette.red.toUpperCase();   // OK - string methods available
palette.green[1];            // OK - tuple element access

// Without satisfies:
// const palette: Record<string, Color> = { ... }
// palette.red.toUpperCase(); // Error - Color doesn't have toUpperCase
// palette.green[1];          // Error - Color doesn't have indexed access
```

Frequency: Medium (50% and growing)

Companies: Microsoft, Google, Projects using newer TypeScript versions

26. What are discriminated unions in TypeScript?

Answer: A discriminated union is a pattern where you use a property (the discriminant) to narrow down the type of an object to a specific variant.

```typescript
// Define types with a common discriminant property
interface Square {
  kind: "square";
  size: number;
}

interface Rectangle {
  kind: "rectangle";
  width: number;
  height: number;
}

interface Circle {
  kind: "circle";
  radius: number;
}

type Shape = Square | Rectangle | Circle;

// Type narrowing with discriminant
function calculateArea(shape: Shape): number {
  switch (shape.kind) {
    case "square":
      return shape.size * shape.size; // TypeScript knows it's a Square
    case "rectangle":
      return shape.width * shape.height; // TypeScript knows it's a Rectangle
    case "circle":
      return Math.PI * shape.radius * shape.radius; // TypeScript knows it's a Circle
    default:
      // Exhaustiveness check
      const _exhaustiveCheck: never = shape;
      throw new Error(`Unhandled shape: ${_exhaustiveCheck}`);
  }
}
```

Frequency: High (70%)

Companies: Microsoft, Facebook, Google, Airbnb, Most TypeScript projects

27. What is the purpose of `Pick<T, K>` and `Omit<T, K>` utility types?

Answer:

- `Pick<T, K>` constructs a type by picking the set of properties `K` from `T`
- `Omit<T, K>` constructs a type by picking all properties from `T` except for those in `K`

```
interface User {
  id: number;
  name: string;
  email: string;
  password: string;
}

// Only includes 'id' and 'name'
type UserBasicInfo = Pick<User, "id" | "name">;

// Includes all except 'password'
type UserPublicInfo = Omit<User, "password">;

const userBasic: UserBasicInfo = { id: 1, name: "John" };
const userPublic: UserPublicInfo = { id: 1, name: "John", email: "john@example.com" };
```

Frequency: High (75%)
Companies: Microsoft, Google, Facebook, Amazon, Most TypeScript projects

28. What is a type guard in TypeScript?

Answer: A type guard is a function or expression that performs a runtime check to guarantee the type of a value within its scope. This helps TypeScript narrow down types.

```typescript
// User-defined type guard function
function isString(value: unknown): value is string {
  return typeof value === "string";
}

function process(value: string | number) {
  if (isString(value)) {
    // TypeScript knows 'value' is a string here
    return value.toUpperCase();
  }
  // TypeScript knows 'value' is a number here
  return value.toFixed(2);
}

// Built-in type guards
function handleValue(value: string | number | null) {
  if (typeof value === "string") {
    // string type guard
    console.log(value.toUpperCase());
  } else if (typeof value === "number") {
    // number type guard
    console.log(value.toFixed(2));
  } else if (value === null) {
    // null check
    console.log("Value is null");
  }
}
```

Frequency: High (80%)

Companies: Microsoft, Google, Facebook, LinkedIn, Most TypeScript projects

29. What are lookup types in TypeScript?

Answer: Lookup types (indexed access types) allow you to extract the type of a property from another type using the square bracket notation.

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

// Extract the type of the 'name' property
type NameType = Person["name"]; // string

// Extract the type of the 'address' property
type Address = Person["address"]; // { street: string; city: string; }

// Extract type using multiple properties
type NameOrAge = Person["name" | "age"]; // string | number

// Extract nested property type
type City = Person["address"]["city"]; // string
```

Frequency: Medium-High (60%)

Companies: Microsoft, Google, Facebook, Advanced TypeScript users

30. What is the purpose of the `strictNullChecks` compiler option?

Answer: When enabled, `strictNullChecks` makes `null` and `undefined` their own types, requiring explicit checks for them before using methods or properties that might not exist.

```
// With strictNullChecks disabled
function getLength(str: string) {
  return str.length; // No error
}
getLength(null); // Runtime error!

// With strictNullChecks enabled
function getLengthSafe(str: string | null | undefined) {
  // Must check before accessing properties
  if (str === null || str === undefined) {
    return 0;
  }
  return str.length; // Safe now
}
// Or using optional chaining
function getLengthModern(str: string | null | undefined) {
  return str?.length ?? 0;
}
```

Frequency: Medium-High (65%)

Companies: Microsoft, Google, Facebook, Amazon, Modern TypeScript projects

31. How do optional chaining (`?.`) and nullish coalescing (`??`) operators work in TypeScript?

Answer:

- Optional chaining (`?.`) allows safe access to properties of potentially null/undefined objects
- Nullish coalescing (`??`) provides a default value when the left-hand expression is null or undefined

```typescript
interface User {
  name: string;
  address?: {
    street?: string;
    city?: string;
  };
}

function getUserCity(user: User): string {
  // Before optional chaining
  /*
  if (user && user.address && user.address.city) {
    return user.address.city;
  }
  return "Unknown";
  */

  // With optional chaining and nullish coalescing
  return user.address?.city ?? "Unknown";
}

const user1: User = { name: "John" };
console.log(getUserCity(user1)); // "Unknown"

const user2: User = { name: "Jane", address: { city: "New York" } };
console.log(getUserCity(user2)); // "New York"
```

Frequency: Very High (85%)
Companies: Microsoft, Google, Facebook, Amazon, Most modern JavaScript/TypeScript projects

32. What is a const assertion in TypeScript?

Answer: A const assertion (`as const`) is a special kind of type assertion that makes literal expressions completely immutable and tells TypeScript to infer the most specific literal type possible.

```
// Without const assertion
const colors = ["red", "green", "blue"];
// Type is string[] - mutable array of strings

// With const assertion
const colorsConst = ["red", "green", "blue"] as const;
// Type is readonly ["red", "green", "blue"] - immutable tuple of specific strings

// Object with const assertion
const settings = {
  theme: "dark",
  fontSize: 16,
  showSidebar: true
} as const;
// All properties are readonly with their literal types
// Type is { readonly theme: "dark"; readonly fontSize: 16; readonly showSidebar: true
```

Frequency: Medium-High (65%)

Companies: Microsoft, Google, Facebook, Stripe, Modern TypeScript projects

33. What is the `ReturnType<T>` utility type and how is it used?

Answer: `ReturnType<T>` extracts the return type from a function type `T`. It's useful for reusing the return type of a function without redefining it.

```typescript
// Define a function
function createUser(name: string, age: number) {
  return {
    id: Date.now(),
    name,
    age,
    isActive: true
  };
}

// Extract the return type
type User = ReturnType<typeof createUser>;
// Equivalent to:
// type User = {
//   id: number;
//   name: string;
//   age: number;
//   isActive: boolean;
// }

// Now we can use this type elsewhere
function updateUser(user: User, changes: Partial<User>): User {
  return { ...user, ...changes };
}
```

Frequency: Medium-High (70%)

Companies: Microsoft, Google, Facebook, Airbnb, Advanced TypeScript projects

34. What are TypeScript template literal types?

Answer: Template literal types (introduced in TypeScript 4.1) allow you to create new string literal types by concatenating other string literal types using the template literal syntax.

```typescript
// Base string literal types
type EventName = "click" | "focus" | "blur";
type ElementType = "button" | "input" | "form";

// Create new string literal types through concatenation
type ElementEvent = `${ElementType}_${EventName}`;
// Expands to: "button_click" | "button_focus" | "button_blur" |
//             "input_click" | "input_focus" | "input_blur" |
//             "form_click" | "form_focus" | "form_blur"

// Practical example
function handleEvent(event: ElementEvent, callback: () => void) {
  console.log(`Handling ${event}`);
  callback();
}

handleEvent("button_click", () => console.log("Button clicked"));
// handleEvent("link_click", () => {}); // Error: "link_click" is not assignable to Ele
```

Frequency: Medium (55% and growing)
Companies: Microsoft, Facebook, Projects using newer TypeScript versions

35. What are ambient declarations in TypeScript?

Answer: Ambient declarations are type declarations for code that exists outside your TypeScript project, typically in JavaScript libraries. They're defined using the `declare` keyword and are often found in `.d.ts` files.

```typescript
// Example of ambient declarations in a .d.ts file
declare module "external-library" {
  export function doSomething(value: string): number;

  export interface Options {
    debug?: boolean;
    timeout?: number;
  }

  export class Helper {
    constructor(options?: Options);
    process(data: any): Promise<any>;
  }
}

// In your TypeScript code
import { doSomething, Helper } from "external-library";
// TypeScript now knows the types even though they're defined externally
```

Frequency: Medium (50%)
Companies: Microsoft, Projects integrating with JavaScript libraries

36. How do you handle dynamic property access with TypeScript?

Answer: Dynamic property access presents challenges in TypeScript because the compiler needs to know which properties exist. You can handle this with indexed access types, type assertions, or the `keyof` operator.

```typescript
// Option 1: Index signature
interface Dictionary {
  [key: string]: any;
}
const dict: Dictionary = { a: 1, b: "string" };
const value = dict["dynamicKey"]; // OK

// Option 2: Type assertion (use carefully)
interface User {
  name: string;
  age: number;
}
const user: User = { name: "John", age: 30 };
function getProp(obj: any, key: string): any {
  return obj[key];
}
const name = getProp(user, "name");

// Option 3: Type-safe with keyof
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}
const age = getProperty(user, "age"); // Strongly typed as number
// const invalid = getProperty(user, "invalid"); // Error
```

Frequency: Medium (60%)

Companies: Microsoft, Google, Facebook, Airbnb, Enterprise applications

37. What are non-null assertion operators in TypeScript?

Answer: The non-null assertion operator (`!`) tells the TypeScript compiler that a variable is definitely not null or undefined, even if the type system thinks it might be.

```
function getValue(): string | null {
  // Implementation may or may not return null
  return Math.random() > 0.5 ? "value" : null;
}

// Option 1: Check for null
const value = getValue();
if (value !== null) {
  console.log(value.toUpperCase()); // Safe
}

// Option 2: Non-null assertion (use with caution)
const forcedValue = getValue()!; // Assert it's not null
console.log(forcedValue.toUpperCase()); // May cause runtime error if null

// Common use case with DOM access
const element = document.getElementById("my-element")!; // We're certain it exists
element.textContent = "Hello"; // OK with non-null assertion
```

Frequency: Medium-High (65%)
Companies: Microsoft, Google, Facebook, Web application developers

38. How do you use type narrowing in TypeScript?

Answer: Type narrowing is the process of refining types from broader to more specific within conditional blocks. Key techniques include typeof guards, instanceof checks, truthiness checks, equality narrowing, the "in" operator, user-defined type guards, and discriminated unions.

```
// Type narrowing using type guards
function process(value: string | number | boolean) {
  // Narrow with typeof
  if (typeof value === "string") {
    console.log(value.toUpperCase()); // value is string here
  } else if (typeof value === "number") {
    console.log(value.toFixed(2));    // value is number here
  } else {
    console.log(value ? "Yes" : "No"); // value is boolean here
  }
}
```

```
function example(value: string | number) {
  if (typeof value === "string") {
    // TypeScript knows value is string here
    return value.toUpperCase();
  }
  // TypeScript knows value is number here
  return value + 1;
}
```

Frequency: High (75%)
Companies: Microsoft, Google, Facebook, Amazon, Stripe

39. What's the difference between `unknown` and `any` types?

Answer: Both can hold any value, but `unknown` is type-safe because you must perform type checking before using properties or methods on an `unknown` value, whereas `any` bypasses all type checking.

```
// With any type
function processAny(value: any) {
  value.toString(); // No error - but might fail at runtime
  value.someNonExistentMethod(); // No error - but will fail at runtime
  return value * 2; // No error - might not make sense
}

// With unknown type (safer)
function processUnknown(value: unknown) {
  // value.toString(); // Error: Object is of type 'unknown'

  // Must check type first
  if (typeof value === "string") {
    return value.toUpperCase(); // OK now
  } else if (typeof value === "number") {
    return value * 2; // OK now
  }
  return String(value);
}
```

Frequency: High (75%)

Companies: Microsoft, Google, Facebook, Amazon, Stripe

40. What is the TypeScript module resolution strategy?

Answer: Module resolution is the process of determining what file is referenced by an import statement. TypeScript has two strategies: Classic (older, simpler) and Node (follows Node.js rules). Most projects use the Node strategy.

```
// In tsconfig.json
{
  "compilerOptions": {
    "moduleResolution": "node", // or "classic"
    // ...
  }
}
```

Frequency: Medium (45%)

Companies: Microsoft, Enterprise projects with complex setups

41. How do you work with third-party libraries that don't have TypeScript definitions?

Answer: You can either create your own type definitions or use the DefinitelyTyped repository (@types packages).

```
// Option 1: Install declaration files from DefinitelyTyped
// npm install --save-dev @types/lodash

// Option 2: Create declaration file (e.g., declarations.d.ts)
declare module 'untyped-module' {
  export function someFunction(arg: string): number;
  export class SomeClass {
    constructor(options?: {[key: string]: any});
    method(): void;
  }
}
```

Frequency: High (70%)

Companies: All companies using TypeScript with JS libraries

42. What are parameter properties in TypeScript classes?

Answer: Parameter properties are a shorthand way to define and initialize class members in the constructor. They allow you to create and initialize a class property in one place.

```
// Without parameter properties
class Person {
  private name: string;
  private age: number;

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

// With parameter properties
class PersonShort {
  constructor(
    private name: string,
    private age: number
  ) {}
  // name and age are automatically created as private properties
}
```

Frequency: Medium (60%)

Companies: Microsoft, Google, Projects following TypeScript best practices

43. What is the `Required<T>` utility type?

Answer: `Required<T>` is the opposite of `Partial<T>`. It creates a type where all properties of `T` are required (non-optional).

```typescript
interface Config {
  host?: string;
  port?: number;
  secure?: boolean;
}

// All properties are required
type RequiredConfig = Required<Config>;

const partialConfig: Config = { host: "localhost" }; // OK
// const requiredConfig: RequiredConfig = { host: "localhost" }; // Error: missing port
const requiredConfig: RequiredConfig = {
  host: "localhost",
  port: 8080,
  secure: false
}; // OK
```

Frequency: Medium (55%)
Companies: Microsoft, Google, Facebook, Modern TypeScript projects

44. What is the purpose of `strictFunctionTypes` compiler option?

Answer: The `strictFunctionTypes` option performs more strict checking of function parameter types for functions that are assigned to a variable or passed as an argument, particularly focusing on contravariance and covariance for function types.

```
// Without strictFunctionTypes
interface Animal { name: string; }
interface Dog extends Animal { bark(): void; }

let animalCallback: (a: Animal) => void;
let dogCallback: (d: Dog) => void;

// This would be allowed without strictFunctionTypes (unsafe)
// animalCallback = dogCallback;

// With strictFunctionTypes, the above assignment is an error
// because Dog is more specific than Animal, and function parameters
// are contravariant
```

Frequency: Low-Medium (40%)
Companies: Microsoft, Projects with strict TypeScript configurations

45. What are branded types in TypeScript?

Answer: Branded types (or nominal types) are a pattern to create types that are structurally identical but treated as different for improved type safety. This is done by adding a unique property as a "brand".

```typescript
// Create branded types
type UserId = string & { readonly _brand: unique symbol };
type OrderId = string & { readonly _brand: unique symbol };

// Create type-safe factory functions
function createUserId(id: string): UserId {
  return id as UserId;
}

function createOrderId(id: string): OrderId {
  return id as OrderId;
}

// Usage
function processUser(userId: UserId) {
  console.log(`Processing user ${userId}`);
}

const userId = createUserId("user-123");
const orderId = createOrderId("order-456");

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

Frequency: Low-Medium (35%)

Companies: Financial services, Safety-critical applications

46. What is the `ThisType<T>` utility type?

Answer: `ThisType<T>` is a marker interface that doesn't add any members but affects the type of `this` in methods of object literals.

```typescript
// Define interface for methods and interface for this context
interface Methods {
  double(): number;
  greet(): string;
}

interface Context {
  value: number;
  name: string;
}

// Use ThisType to combine them
const obj: Methods & ThisType<Context> = {
  double() {
    return this.value * 2; // 'this' has type Context
  },
  greet() {
    return `Hello, ${this.name}`; // 'this' has type Context
  }
};

// Usage (with call to provide the 'this' context)
const context: Context = { value: 10, name: "John" };
console.log(obj.double.call(context)); // 20
console.log(obj.greet.call(context)); // "Hello, John"
```

Frequency: Low (30%)

Companies: Advanced TypeScript users, Library authors

47. How do you handle circular type dependencies in TypeScript?

Answer: Circular type dependencies occur when two or more types reference each other. TypeScript can handle simple circularities, but complex ones require interfaces (which are lazily evaluated) or careful type structuring.

```typescript
// Simple circular reference
interface Person {
  name: string;
  friends: Person[]; // Self-reference is fine
}

// More complex circular dependency between types
interface Employee {
  name: string;
  department: Department;
}

interface Department {
  name: string;
  head: Employee;
  members: Employee[];
}

// Type aliases might need a workaround with interfaces
// or indirection
```

Frequency: Medium (50%)

Companies: Microsoft, Enterprise applications with complex domain models

48. What is the `asserts` modifier in type predicates?

Answer: The `asserts` modifier (added in TypeScript 3.7) allows you to create assertion functions that tell the type system a condition must be true for the execution to continue.

```typescript
// Assert function with type predicate
function assertIsString(value: unknown): asserts value is string {
  if (typeof value !== "string") {
    throw new Error("Value must be a string");
  }
}

function processValue(value: unknown) {
  assertIsString(value);
  // TypeScript now knows value is a string
  console.log(value.toUpperCase());
}

// Assertion function with condition
function assert(condition: boolean, message: string): asserts condition {
  if (!condition) {
    throw new Error(message);
  }
}

function divide(a: number, b: number): number {
  assert(b !== 0, "Division by zero");
  // TypeScript knows b is not 0 here
  return a / b;
}
```

Frequency: Low-Medium (40%)

Companies: Microsoft, Projects with strict error handling

49. What is the purpose of `as const` assertions for objects?

Answer: When applying `as const` to an object literal, it recursively marks all its properties as `readonly` and all array literals as `readonly` tuples, ensuring the most specific literal types are inferred.

```typescript
// Regular object - wide types
const config = {
  server: {
    host: "localhost",
    port: 8080
  },
  timeout: 1000,
  active: true
};
// config.server.port = 3000; // OK
// config.active = false;     // OK

// With const assertion - narrow, specific types
const configConst = {
  server: {
    host: "localhost",
    port: 8080
  },
  timeout: 1000,
  active: true
} as const;
// configConst.server.port = 3000; // Error - readonly
// configConst.active = false;     // Error - readonly

// Useful for derived types
type ServerConfig = typeof configConst.server;
// { readonly host: "localhost"; readonly port: 8080; }
```

Frequency: Medium-High (65%)

Companies: Microsoft, Google, Facebook, Modern TypeScript projects

50. What are Function Overloads in TypeScript?

Answer: Function overloads allow you to define multiple function types for the same function to handle different parameter types and return values more precisely.

```typescript
// Function overload signatures
function process(value: string): string[];
function process(value: number): number;
function process(value: boolean): boolean;
// Implementation signature must be compatible with all overloads
function process(value: string | number | boolean): string[] | number | boolean {
  if (typeof value === "string") {
    return value.split("");
  } else if (typeof value === "number") {
    return value * 2;
  } else {
    return !value;
  }
}

// Usage
const result1 = process("hello"); // Type: string[]
const result2 = process(42);      // Type: number
const result3 = process(true);    // Type: boolean
```

Frequency: High (75%)

Companies: Microsoft, Google, Facebook, Amazon, Enterprise applications