1. What is Routing Guard in Angular?

- Routing Guards control navigation to and from routes in an Angular application.
- Common types include CanActivate, CanDeactivate, Resolve, and CanLoad.
- Guards are implemented using classes that implement a specific guard interface.
- Example: To protect a route, use CanActivate.

```
@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  canActivate(): boolean {
    return !!localStorage.getItem('token'); // Allow only if token exists
  }
}
const routes: Routes = [{ path: 'dashboard', component:
DashboardComponent, canActivate: [AuthGuard] }];
```

2. What is a Module, and what does it contain?

- A module is a container for components, directives, pipes, and services.
- Every Angular app has at least one module, the AppModule.
- Modules group related functionalities and can be lazy-loaded for optimization.
- Example: Import and declare components in NgModule.

```
@NgModule({
  declarations: [AppComponent, DashboardComponent],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

3. What's the difference between an Angular Component and a Module?

- A module organizes the structure of an app, while a component handles the UI and logic.
- Modules are defined with @NgModule, and components are defined with @Component.
- Components belong to a module and require their module to run.
- Example: A UserModule manages user-related components, such as UserProfileComponent.

4. How would you protect a component from being activated through the router?

- Use a CanActivate guard.
- Implement logic to check user permissions or authentication.
- Attach the guard to the desired route.
- Example:

```
@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  canActivate(route: ActivatedRouteSnapshot): boolean {
    return route.queryParams['role'] === 'admin';
  }
}
```

5. What are Observables?

- Observables are data streams used for handling asynchronous events.
- Provided by RxJS, they support operations like mapping, filtering, and merging.
- Observables are lazy and execute only upon subscription.
- Example: Fetching data with HttpClient.

```
this.http.get('/api/users').subscribe(data => console.log(data));
```

6. How would you run unit tests in Angular?

- Angular uses Karma as the default test runner.
- Write test cases using Jasmine framework.
- Run tests with the ng test command.
- Example: A basic component test.

```
it('should create the app', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

7. What is the difference between ngIf vs ngIf [hidden]?

- *ngIf removes or adds elements to the DOM.
- [hidden] only hides elements via display: none.
- *ngIf is better for performance when toggling large sections.
- Example:

```
<div *ngIf="isVisible">Visible</div>
<div [hidden]="!isVisible">Hidden</div>
```

8. What is Interpolation in Angular?

- Interpolation binds data from a component to the template.
- Uses {{ expression }} syntax to display dynamic values.
- Interpolation can evaluate simple expressions.
- Example:

```
<h1>Hello {{ username }}</h1>
```

9. You have an HTML response I want to display. How do I do that?

- Use [innerHTML] to bind HTML safely.
- Sanitize the response if needed to prevent XSS attacks.
- Use DomSanitizer for trusted content.
- Example:

```
this.safeHtml = this.sanitizer.bypassSecurityTrustHtml(responseHtml);
```

10. What is the difference between @Component and @Directive in Angular?

- @Component creates UI elements; @Directive modifies behavior or appearance.
- Components must have a template; directives don't.
- Use @Directive for attribute and structural modifications.
- Example:

```
@Directive({
  selector: '[highlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

11. What is an Observer in Angular?

- An observer listens to observable streams.
- It responds to next, error, or complete notifications.
- Observers must implement Observer<T> interface.
- Example:

```
const observer = {
  next: (value) => console.log(value),
  error: (err) => console.error(err),
  complete: () => console.log('Complete')
};
observable$.subscribe(observer);
```

12. What is the difference between Structural and Attribute directives in Angular?

- Structural directives change the DOM layout, e.g., *ngIf, *ngFor.
- Attribute directives change the appearance/behavior, e.g., [ngClass].
- Structural directives begin with an asterisk.
- Example:

```
<div *ngIf="isVisible">Visible</div>
<div [ngClass]="{'highlight': isActive}"></div>
```

13. What is a bootstrapping module in Angular?

- The root module bootstraps the Angular app.
- It declares the entry component for application rendering.
- Usually defined as AppModule.

- Example:

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

14. What is the purpose of the base href tag?

- Specifies the base URL for resolving relative paths in the app.
- Enables Angular's routing to interpret paths correctly.
- Located in index.html as <base href="/">.
- Example: For a deployment under /app/, set <base href="/app/">.

15. What is the equivalent of ngShow and ngHide in Angular?

- Use [hidden] or *ngIf for show/hide functionality.
- [hidden] retains elements in the DOM.
- *ngIf removes or adds elements dynamically.
- Example:

```
<div [hidden]="!isVisible">Hidden</div>
<div *ngIf="isVisible">Visible</div>
```

16. What is an Observable?

- An observable is a stream of asynchronous data that emits multiple values over time.
- It provides powerful operators like map, filter, and merge.
- Observables are lazy and require a subscription to begin execution.
- Example: Using HttpClient to fetch data as an observable.

```
this.http.get('/api/data').subscribe(response => console.log(response));
```

17. What is the minimum definition of a Component?

- A component requires a selector, template or template URL, and metadata.
- Defined with the @Component decorator.
- At least one root component must exist in an Angular application.
- Example:

```
@Component({
  selector: 'app-hello',
  template: '<h1>Hello, World!</h1>'
})
export class HelloComponent {}
```

18. What are the differences between AngularJS (Angular 1.x) and Angular (Angular 2.x and beyond)?

- Angular uses TypeScript, whereas AngularJS uses JavaScript.
- Angular has a component-based architecture; AngularJS uses controllers and scope.
- Angular provides better performance with AOT and Ivy Renderer.
- Example: Angular uses modules (@NgModule), whereas AngularJS uses modules with dependency injection.

19. What is a Custom Component? Why would you use it?

- A custom component is a user-defined, reusable UI building block.
- It encapsulates template, style, and logic for specific functionality.
- Used to maintain modularity and code reuse in applications.
- Example:

```
@Component({
  selector: 'app-user-card',
  template: '<div>{{ user.name }}</div>'
})
export class UserCardComponent {
  @Input() user: any;
}
```

20. What is a Service, and when will you use it?

- Services handle reusable logic, such as data fetching and business logic.
- They provide dependency injection for components.
- Services are shared across components for better maintainability.
- Example: Creating a user service for API calls.

```
@Injectable({ providedIn: 'root' })
export class UserService {
  constructor(private http: HttpClient) {}
  getUsers() {
    return this.http.get('/api/users');
  }
}
```

21. Explain how Custom Elements work internally.

- Custom elements are web components defined with Angular components.
- Angular creates these as native browser elements via @angular/elements.
- They encapsulate functionality, making them reusable in non-Angular projects.
- Example: Convert a component into a custom element.

```
const userCardElement = createCustomElement(UserCardComponent, { injector });
customElements.define('user-card', userCardElement);
```

22. When would you use Lazy Loading in Angular?

- Lazy loading loads modules only when required, improving app performance.
- Used for large apps to reduce initial load time.
- Implemented using Angular's router with loadChildren.
- Example:

```
const routes: Routes = [
  { path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule) }
];
```

23. What is a Parameterized Pipe?

- A pipe that takes arguments to transform data dynamically.
- Examples include date, currency, and slice pipes.
- Custom parameterized pipes can be created.
- Example:

```
<p>{{ price | currency:'USD':true }}</p>
```

24. What is Bazel in Angular?

- Bazel is a build tool for efficient, incremental builds in Angular.
- It provides fine-grained control over build outputs.
- Enables faster builds and tests in large-scale projects.
- Example: Configuring Bazel for Angular CLI.

```
ng new my-app --collection=@angular/bazel
```

25. How do you categorize data binding types in Angular?

- Interpolation: {{ expression }} for one-way binding from component to DOM.
- Property binding: [property]="value" for dynamic DOM updates.
- Event binding: (event)="handler" for handling user actions.
- Two-way binding: [(ngModel)] to bind both property and event.
- Example:

```
<input [(ngModel)]="username">
```

26. What is Multicasting in RxJS?

- Multicasting shares a single observable execution among multiple subscribers.
- Achieved using subjects like BehaviorSubject or ReplaySubject.
- Reduces redundant operations and improves efficiency.
- Example:

```
const subject = new BehaviorSubject(0);
subject.subscribe(value => console.log('Subscriber A:', value));
subject.subscribe(value => console.log('Subscriber B:', value));
subject.next(1);
```

27. What are the utility functions provided by RxJS?

- Common utilities
  include of, from, merge, concat, combineLatest, and forkJoin.
- These functions create observables and manipulate data streams.
- Widely used for complex asynchronous tasks.
- Example: Combine multiple streams.

```
combineLatest([stream1, stream2]).subscribe(([a, b]) => console.log(a, b));
```

28. Do I always need a Routing Module in Angular?

- Routing modules are optional but recommended for modularity in apps with routing.
- For small apps, routes can be defined in AppModule.
- A separate routing module simplifies route management in large apps.
- Example:

```
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AdminRoutingModule {}
```

29. What are the ways to control AOT compilation?

- Enable AOT explicitly with ng build --aot.
- Use angular.json to set AOT as default.
- Optimize build pipelines by controlling template and metadata precompilation.
- Example: Configuring AOT in angular.json.

```
"build": {
  "options": {
    "aot": true
  }
}
```

30. What is Router Outlet in Angular?

- RouterOutlet is a placeholder to load routed components dynamically.
- It acts as a viewport in a template for routing.
- Supports named outlets for rendering multiple routes.
- Example:

```
<router-outlet></router-outlet>
```

31. Explain Lazy Loading in Angular.

- Lazy loading is a technique to load modules only when their routes are accessed.
- It helps reduce the initial load time of the application.
- Configured using the loadChildren property in the route definitions.
- Example:

```
const routes: Routes = [
  { path: 'feature', loadChildren: () =>
import('./feature/feature.module').then(m => m.FeatureModule) }
];
```

32. How to inject the base href in Angular?

- Angular uses the APP_BASE_HREF token to configure the base path.
- You can set it explicitly in the providers array.
- Alternatively, it can be dynamically determined using document.baseURI.
- Example:

```
providers: [{ provide: APP_BASE_HREF, useValue: '/my-app/' }]
```

33. What is Protractor in Angular?

- Protractor is an end-to-end testing framework for Angular applications.
- It automates interaction with web elements for UI testing.
- Works seamlessly with Angular's elements like ng-model and ng-repeat.
- Example:

```
it('should display the title', async () => {
  await browser.get('/');
  const title = await element(by.css('h1')).getText();
  expect(title).toBe('My App');
});
```

34. What is the Activated Route in Angular?

- ActivatedRoute provides access to the route's parameters, data, and query parameters.
- It is an injectable service available for the current route.
- It enables reactive programming with observables for route changes.
- Example:

```
constructor(private route: ActivatedRoute) {
  this.route.params.subscribe(params => console.log(params['id']));
}
```

35. What is the purpose of a Wildcard route?

- Wildcard routes handle undefined or fallback paths.
- They are typically defined as the last route in the configuration.
- Prevents users from accessing undefined routes.
- Example:

```
const routes: Routes = [
  { path: '**', component: PageNotFoundComponent }
];
```

36. What is Router State in Angular?

- Router state refers to the state of the router at a given point in navigation.
- It includes the activated route tree and snapshot of the route.
- Provides information like parameters, data, and URLs.
- Example: Accessing RouterStateSnapshot.

```
constructor(private router: Router) {
  console.log(this.router.routerState.snapshot.url);
}
```

37. What are Custom Elements in Angular?

- Custom elements are Angular components transformed into native web components.
- They are compatible with any framework or plain HTML.
- Created using @angular/elements package.
- Example:

```
const customEl = createCustomElement(MyComponent, { injector });
customElements.define('my-component', customEl);
```

38. What is the difference between Promise and Observable in Angular?

- Promises are eager and resolve once, while observables are lazy and emit multiple values.
- Observables support operators for transformation and composition.
- Observables are cancellable, but promises are not.
- Example: Observables for real-time data.

```
this.data$.subscribe(value => console.log(value));
```

39. What happens if you use a <script> tag inside the template?

- Angular sanitizes and removes <script> tags for security reasons.
- It prevents cross-site scripting (XSS) vulnerabilities.
- Use [innerHTML] with sanitized content for dynamic script embedding.
- Example: Using DomSanitizer for safe HTML.

```
this.trustedHtml =
this.sanitizer.bypassSecurityTrustHtml('<div>Hello</div>');
```

40. What is Angular Ivy Renderer?

- Ivy is Angular's next-generation rendering engine introduced in Angular 9.
- It optimizes bundle sizes and improves compilation speed.
- Offers better debugging with human-readable code.
- Example: Enabling Ivy in angular.json.

```
"angularCompilerOptions": {
  "enableIvy": true
}
```
41. What is Subscribing in Angular?

- Subscribing starts the execution of an observable.
- It listens for data emissions (next), errors, or completion of the observable.
- Required for consuming asynchronous streams like HTTP requests.
- Example:

```
this.http.get('/api/data').subscribe(data => console.log(data));
```
42. What are dynamic components?

- Dynamic components are created and loaded at runtime.
- Useful for modular and reusable UI elements.
- Achieved using ComponentFactoryResolver or ViewContainerRef.
- Example:

```
const factory = this.resolver.resolveComponentFactory(MyComponent);
this.container.createComponent(factory);
```
43. What is the option to choose between Inline and External template files?

- Inline templates are defined directly in the template property.
- External templates are specified with templateUrl and loaded as separate files.
- Inline templates are suitable for small, simple components.
- Example:

```
@Component({
  selector: 'app-inline',
  template: '<h1>Hello Inline!</h1>'
})
export class InlineComponent {}
```
44. Why does Incremental DOM have a low memory footprint?

- It directly manipulates the DOM without creating intermediate structures.
- Only updates changed elements, minimizing overhead.
- Eliminates the need for virtual DOM reconciliation.
- Example: Angular Ivy uses Incremental DOM for efficient rendering.

45. How do you perform Error Handling for HttpClient in Angular?

- Use catchError operator to handle HTTP errors.
- Optionally, log errors using a service for monitoring.
- Show user-friendly error messages in the UI.
- Example:

```
this.http.get('/api/data').pipe(
  catchError(err => {
    console.error('Error occurred:', err);
    return of([]); // Return fallback data
  })
).subscribe();
```
46. How do you perform error handling in Observable in Angular?

- Use the catchError operator in the observable pipeline to intercept errors.
- Combine with retry or retryWhen for automatic retries on failure.
- Gracefully handle errors by providing fallback values or logging them.
- Example:

```
this.data$.pipe(
  catchError(err => {
    console.error('Error:', err);
    return of([]); // Provide fallback data
  })
).subscribe(data => console.log(data));
```
47. What is Angular Universal?

- Angular Universal enables server-side rendering (SSR) for Angular apps.
- Improves SEO and performance by rendering pages on the server.
- Uses @nguniversal/express-engine for Node.js server integration.
- Example:

```
ng add @nguniversal/express-engine
npm run build:ssr
npm run serve:ssr
```
48. What is TestBed in Angular?

- TestBed is a testing utility that creates an Angular testing module.
- Configures components, services, and modules for testing.
- Allows mocking dependencies and simulating real scenarios.
- Example:

```
TestBed.configureTestingModule({
  declarations: [MyComponent],
  providers: [MyService]
}).compileComponents();
```
49. What is Redux, and how does it relate to an Angular app?

- Redux is a predictable state management library for JavaScript apps.
- It centralizes app state in a single store, making debugging easier.
- Integrated into Angular via libraries like ngrx/store.
- Example:

```
store.dispatch({ type: '[User] Load', payload: userId });
store.select('user').subscribe(data => console.log(data));
```
50. What is the use of Codelyzer in Angular?

- Codelyzer is a static analysis tool for Angular TypeScript projects.
- Enforces Angular coding standards and best practices.
- Works as a TSLint plugin to provide linting rules for Angular.
- Example:

```
npm install --save-dev codelyzer
ng lint
```
51. What's new in Angular 6, and why should we upgrade to it?

- Introduced ng update for automated dependency updates.
- Added RxJS 6 with improved operators and better performance.
- Included support for Angular Elements to create custom web components.
- Example: Angular CLI workspace feature for managing multiple projects.

52. Can you explain the difference between Promise and Observable in Angular? In what scenario can we use each case?

- Promises handle single asynchronous events, while observables handle streams.
- Observables support operators for chaining and transformation.
- Use Promises for simple, one-time HTTP requests and Observables for streams like real-time updates.
- Example:

```
const observable$ = this.http.get('/api').subscribe(data =>
console.log(data));
```
53. What is the difference between declarations, providers, and imports in NgModule?

- Declarations: Components, directives, and pipes that belong to this module.
- Providers: Services available throughout the app via dependency injection.
- Imports: Other modules whose exported components and services are needed.
- Example:

```
@NgModule({
  declarations: [MyComponent],
  imports: [CommonModule],
  providers: [MyService]
})
export class MyModule {}
```
54. Why should ngOnInit be used if we already have a constructor?

- Constructor is for dependency injection and initialization of the object.
- ngOnInit is specifically designed for component initialization logic.
- Ensures properties are set and bindings are available.
- Example:

```
ngOnInit() {
  this.loadData();
}
```
55. Why would you use a spy in a test?

- Spies intercept and mock the behavior of dependencies.
- Used to test components in isolation without actual service calls.
- Captures method calls and arguments for verification.
- Example:

```
const spy = spyOn(service, 'getData').and.returnValue(of(mockData));
```
56. What is AOT?

- AOT (Ahead-of-Time) compiles Angular HTML and TypeScript code during build time.
- Reduces runtime errors and improves app performance.
- Produces optimized and smaller JavaScript bundles.
- Example: Enable AOT with Angular CLI:

```
ng build --aot
```
57. Explain the difference between Constructor and ngOnInit.

- Constructor is a TypeScript feature for dependency injection and initializing the object.
- ngOnInit is an Angular lifecycle hook for initializing the component.
- Component bindings and inputs are not guaranteed in the constructor.
- Example: Use constructor for injection, and ngOnInit for setup logic.

58. What is the difference between @Component and @Directive in Angular?

- @Component is used to create UI elements with templates and styles.
- @Directive adds behavior to existing DOM elements.
- Components are self-contained, while directives work on other elements.
- Example:

```
@Directive({
  selector: '[highlight]'
})
export class HighlightDirective {
  @HostListener('mouseenter') onMouseEnter() {
    this.highlight('yellow');
  }
}
```
59. What is the difference between Promise and Observable in Angular?

- Promises are resolved once, Observables emit multiple values over time.
- Observables provide operators for data transformation.
- Observables are lazy, while Promises are eager.
- Example: Real-time chat using Observables vs fetching user data with Promises.

60. What is Incremental DOM, and how is it different from Virtual DOM?

- Incremental DOM updates the DOM directly without creating a virtual representation.
- Reduces memory overhead and processing time by skipping reconciliation.
- Angular Ivy uses Incremental DOM for faster and more efficient rendering.
- Example: Incremental DOM optimizes partial updates rather than full-tree changes.

61. Angular 9: What are some new features in Angular 9?

- Introduction of Ivy as the default rendering engine.
- Enhanced debugging with stack traces and better error messages.
- Improved type-checking for templates in development mode.
- Example: Smaller bundle sizes with Ivy, enabling faster loads.

62. Do I need to bootstrap custom elements?

- No, Angular custom elements are bootstrapped automatically by the browser.
- They are self-contained and registered with customElements.define().
- Angular handles their lifecycle and change detection internally.
- Example:

```
const el = createCustomElement(MyComponent, { injector });
customElements.define('my-element', el);
```
63. Are there any pros/cons (especially performance-wise) in using local storage to replace cookie functionality?

- Pros: Larger storage capacity, faster client-side access, no need for server communication.
- Cons: Not sent automatically with HTTP requests, less secure than HTTP-only cookies.
- Suitable for non-sensitive, client-side data storage.
- Example: Storing user preferences.

```
localStorage.setItem('theme', 'dark');
```
64. What are the lifecycle hooks for components and directives in Angular?

- ngOnChanges: Called when input properties change.
- ngOnInit: Invoked after the component is initialized.
- ngOnDestroy: Triggered before the component is destroyed.
- Example:

```
ngOnDestroy() {
  this.subscription.unsubscribe();
}
```
65. How to detect a route change in Angular?

- Use the Router service’s events observable.
- Filter for NavigationStart or NavigationEnd events.
- Can also use ActivatedRoute for specific route parameters.
- Example:

```
this.router.events.pipe(
  filter(event => event instanceof NavigationStart)
).subscribe(() => console.log('Route changed'));
```
66. What are the advantages of AOT compilation?

- Eliminates runtime template errors by catching them during build time.
- Reduces app bundle size with optimized code.
- Improves application performance by compiling templates ahead of time.
- Example: Production build with AOT.

```
ng build --prod --aot
```
67. Explain the purpose of Service Workers in Angular.

- Service workers enable Progressive Web App (PWA) features like offline access and caching.
- Handle background tasks and push notifications.
- Implemented using the @angular/service-worker package.
- Example:

```
ng add @angular/pwa
```
68. What is the difference between Incremental DOM and Virtual DOM?

- Incremental DOM updates DOM directly and skips intermediate representations.
- Virtual DOM creates a virtual representation for reconciliation before DOM updates.
- Incremental DOM is more memory-efficient and faster for large apps.
- Example: Angular Ivy uses Incremental DOM for rendering.

69. Why do we need the compilation process in Angular?

- Converts templates into JavaScript for rendering efficiency.
- Enables dependency injection, directives, and component behaviors.
- Ensures app is optimized for performance during the build process.
- Example: JIT compilation for dynamic builds, AOT for production.

70. What is the need for SystemJS in Angular?

- SystemJS is a module loader that dynamically loads JavaScript modules.
- Used in earlier versions of Angular for module resolution.
- Replaced by Webpack in modern Angular CLI for better bundling.
- Example: Angular CLI-generated apps use Webpack by default.

71. Why should we use Bazel for Angular builds?

- Bazel enables incremental builds and efficient dependency management.
- Allows parallel builds for large-scale applications.
- Facilitates universal builds across multiple languages.
- Example: Bazel's support for distributed builds in CI pipelines.

72. How do you create an application to use Webpack?

- Install Webpack and dependencies like loaders and plugins.
- Configure webpack.config.js with entry points and output.
- Replace Angular CLI commands with Webpack-specific scripts.
- Example:

```
webpack --config webpack.config.js
```

73. What is Upgrade in Angular?

- Upgrade facilitates running AngularJS (1.x) and Angular (2+) side by side.
- Helps transition legacy apps incrementally to modern Angular.
- Achieved using @angular/upgrade library.
- Example: Upgrade an AngularJS service.

```
upgradeAdapter.upgradeNg1Provider('myService');
```

74. What is Reactive Programming, and how does it relate to Angular?

- Reactive programming focuses on asynchronous data streams.
- RxJS implements reactive programming for Angular applications.
- Enables event-driven architectures with operators for data transformation.
- Example: Using switchMap for HTTP calls.

```
this.search$.pipe(
  switchMap(term => this.http.get(`/search?q=${term}`))
).subscribe(results => console.log(results));
```

75. Name some security best practices in Angular.

- Use Angular's built-in sanitization for user inputs (DomSanitizer).
- Avoid dynamic templates and <script> tags in bindings.
- Implement route guards and authentication mechanisms.
- Example: Use HttpInterceptor to append secure headers.

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  const secureReq = req.clone({ headers: req.headers.set('Authorization', 'Bearer token') });
  return next.handle(secureReq);
}
```

76. When is a lazy-loaded module loaded?

- A lazy-loaded module is loaded when its associated route is accessed.
- It is not part of the main bundle, reducing the initial load time.
- Configured using loadChildren in the route definition.
- Example:

```
const routes: Routes = [
  { path: 'feature', loadChildren: () =>
import('./feature/feature.module').then(m => m.FeatureModule) }
];
```

77. Why would you use Renderer methods instead of using native element methods?

- Renderer methods provide a platform-independent way to manipulate DOM.
- Helps maintain compatibility across different environments like SSR.
- Prevents direct DOM access, ensuring secure interactions.
- Example:

```
this.renderer.setStyle(element, 'background-color', 'blue');
```

78. What is Ivy Renderer in Angular?

- Ivy is Angular's new rendering engine introduced in Angular 9.
- It compiles components into efficient JavaScript code for faster rendering.
- Supports incremental DOM for memory-efficient updates.
- Example: Ivy reduces the size of unused components in tree-shaking.

79. What does detectChanges do in Angular Jasmine tests?

- detectChanges triggers change detection for the component.
- Ensures updates to bindings and the DOM are reflected during tests.
- Used with ComponentFixture in Angular test setups.
- Example:

```
fixture.detectChanges();
expect(component.title).toBe('Test Title');
```

80. What is the difference between a pure and impure pipe in Angular?

- A pure pipe runs only when input values change.
- An impure pipe runs on every change detection cycle, even for unrelated changes.
- Use pure pipes for performance-critical scenarios.
- Example:

```
@Pipe({ name: 'filter', pure: false })
export class FilterPipe {
  transform(items: any[], searchText: string): any[] { /* filtering logic */ }
}
```

81. Why would you use lazy loading for modules in an Angular app?

- Reduces the initial load time by deferring module loading.
- Improves performance for apps with many features.
- Helps organize and maintain large codebases.
- Example: Lazy load a module with route configuration as follows:

```
{ path: 'admin', loadChildren: () =>
import('./admin/admin.module').then(m => m.AdminModule) }
```

82. What are the mapping rules between Angular component and custom element?

- Component selector becomes the custom element tag name.
- Inputs and outputs map to properties and events of the custom element.
- Styles and encapsulation are retained within the shadow DOM.
- Example:

```
@Component({ selector: 'app-my-element', template: `<p>Hello!</p>` })
export class MyElement {}
customElements.define('app-my-element',
createCustomElement(MyElement, { injector }));
```

83. Name and explain some Angular Module Loading examples.

- **Eager Loading**: Modules are loaded at the app's startup.
- **Lazy Loading**: Modules are loaded when their route is accessed.
- **Preloading**: Modules are loaded in the background after the app initializes.

- Example: Configure preloading with PreloadAllModules:

```
RouterModule.forRoot(routes, { preloadingStrategy: PreloadAllModules })
```

84. What is Zone in Angular?

- Zone.js is a library for intercepting and keeping track of asynchronous operations.
- It ensures Angular detects changes triggered by async tasks.
- Facilitates seamless DOM updates in response to events.
- Example: Zone handles events like setTimeout or HTTP requests automatically.

85. What would be a good use for NgZone service?

- Use NgZone to run code outside Angular's zone to avoid triggering change detection.
- Re-enter the zone for tasks requiring UI updates.
- Helps optimize performance in high-frequency async tasks.
- Example:

```
this.ngZone.runOutsideAngular(() => {
  setTimeout(() => this.ngZone.run(() => this.updateUI()), 1000);
});
```

86. How would you insert an embedded view from a prepared TemplateRef?

- Use ViewContainerRef to create and insert a view.
- Pass the TemplateRef to createEmbeddedView of ViewContainerRef.
- Useful for dynamic templates and reusable UI blocks.
- Example:

```
this.viewContainerRef.createEmbeddedView(this.templateRef);
```

87. What does the Just-in-Time (JIT) compiler do?

- JIT compiles Angular templates and code at runtime in the browser.
- Useful for development mode with faster rebuild times.
- Does not require precompiled templates, unlike AOT.
- Example: Angular CLI uses JIT by default for development:

```
ng serve
```

88. What are observable creation functions in RxJS?

- Functions like of, from, interval, and timer create observables.
- Enable the creation of different types of streams for various scenarios.
- Example:

```
const observable$ = of(1, 2, 3);
observable$.subscribe(val => console.log(val));
```

89. What is the Locality principle for Ivy?

- Locality means each component is compiled independently.
- Simplifies debugging and enables better tree-shaking.
- Reduces the coupling between components and modules.
- Example: Ivy doesn't need module context for compiling components.

90. How would you compare View Engine vs Ivy in Angular?

- Ivy has smaller bundle sizes due to tree-shaking; View Engine doesn't.
- Ivy uses incremental DOM for faster rendering, while View Engine relies on templates.
- Ivy supports locality for independent component compilation.
- Example: Upgrading to Ivy reduces bundle sizes and improves debugging.

91. When to use query parameters vs matrix parameters in URLs?

- **Query Parameters**: Used to pass data globally, typically for search, filters, or pagination.
- **Matrix Parameters**: Used for passing parameters specific to a route segment.
- Query parameters are part of the URL query string (?param=value), while matrix parameters use a semicolon (;param=value).
- Example:

```
// Query parameters
this.router.navigate(['/search'], { queryParams: { q: 'angular' } });
// Matrix parameters
this.router.navigate(['/product', { id: 1, color: 'red' }]);
```

92. Angular 8: What are some changes in the Location module?

- Support for multiple baseHref values during runtime.
- Added support for Location.getState() to retrieve history states.
- Improved handling of hash-based navigation.
- Example: Retrieve the current state:

```
const state = this.location.getState();
console.log(state);
```

93. Angular 9: Explain improvements in Tree-Shaking.

- Ivy eliminates unused code more effectively with enhanced tree-shaking.
- Components and directives are compiled independently, enabling better optimizations.
- Reduces bundle size by removing unused features and modules.
- Example: Unused Angular directives no longer inflate the bundle size.

94. How does Ivy affect the (Re)build time?

- Ivy improves incremental rebuild times due to its locality principle.
- Uses precompiled code, reducing the need for recompiling unchanged modules.
- Smaller bundle sizes lead to faster development and testing cycles.
- Example: Faster component-only rebuilds during development.

95. Just-in-Time (JIT) vs Ahead-of-Time (AOT) compilation: Explain the difference.

- **JIT**: Compilation happens in the browser during runtime; faster builds but slower runtime.
- **AOT**: Compilation occurs at build time, resulting in faster runtime and optimized bundles.
- AOT detects errors at build time, reducing runtime crashes.
- Example:

```
# JIT
ng serve
# AOT
ng build --prod --aot
```

96. Do you know how you can run AngularJS and Angular side by side?

- Use @angular/upgrade for hybrid apps to bridge AngularJS and Angular.
- Upgrade AngularJS components and services incrementally.
- Bootstrap the app using UpgradeModule.
- Example:

```
import { UpgradeModule } from '@angular/upgrade/static';
platformBrowserDynamic().bootstrapModule(AppModule).then(ref =>
ref.injector.get(UpgradeModule).bootstrap(document.body, ['myApp']));
```

97. Why does Angular use URL segments?

- URL segments allow Angular to parse and handle routes hierarchically.
- Provides better control over child routes and their parameters.
- Facilitates matrix parameters for route-specific data.
- Example: /product;id=123/details;tab=reviews uses segments for clarity.

98. What is the difference between BehaviorSubject vs Observable in Angular?

- **Observable**: Emits values only to subscribers when they subscribe.
- **BehaviorSubject**: Emits the last emitted value to new subscribers immediately.
- BehaviorSubject requires an initial value; Observable does not.
- Example:

```
const behaviorSubject = new BehaviorSubject('Initial Value');
behaviorSubject.subscribe(value => console.log(value)); // Outputs: 'Initial Value'
```

99. Name some differences between SystemJS vs Webpack.

- **SystemJS**: A dynamic module loader; less optimized for bundling.
- **Webpack**: A static bundler, handles multiple assets and optimizations.
- Webpack supports advanced features like lazy loading and tree-shaking.
- Example: Angular CLI uses Webpack under the hood for builds.

100. How would you extract Webpack config from an Angular CLI project?

- Angular CLI abstracts Webpack configuration by default.
- Use ng eject in earlier versions (deprecated after Angular 6).
- Alternative: Use custom builders to override Webpack configuration.
- Example: Modify configurations using angular-builders/custom-webpack.

101. Why is Incremental DOM Tree-Shakable?

- Incremental DOM directly manipulates the DOM, avoiding intermediate representations.
- Compiles each component into independent, self-contained code.
- Unused components are automatically excluded during tree-shaking.
- Example: Ivy leverages Incremental DOM for tree-shakable builds.

102. Why did the Google team go with Incremental DOM instead of Virtual DOM?

- Incremental DOM is more memory-efficient for large apps.
- Updates only the necessary DOM elements, reducing reconciliation overhead.
- It simplifies Angular's rendering pipeline with fewer abstractions.
- Example: Faster rendering for data-heavy apps using Angular Ivy.

103. Could you provide some particular examples of using Zone in Angular?

- Tracking asynchronous operations like setTimeout or HTTP requests.
- Ensuring UI updates automatically in response to async tasks.
- Useful in debugging with zone-related error stack traces.
- Example: Zone handles automatic change detection for promises.

104. What are Pipes in Angular? Give me an example.

- Pipes transform data for presentation in templates.
- Angular provides built-in pipes like uppercase, date, and currency.
- Custom pipes can handle domain-specific transformations.
- Example:

```
@Pipe({ name: 'square' })
export class SquarePipe {
  transform(value: number): number {
    return value * value;
  }
}
// Usage: {{ 5 | square }} outputs 25
```

105. What does this line do in Angular?

```
this.changeDetectorRef.detectChanges();
```

- Manually triggers change detection for the component.
- Useful in scenarios where Angular's automatic change detection doesn't suffice.
- Ensures updates to the DOM after async operations.
- Example: Force update a component after external state changes.

106. How can I select an element in a component template?

- Use Angular's @ViewChild or @ViewChildren decorators.
- ElementRef provides direct access to the DOM element.
- Prefer Renderer2 for DOM manipulations to maintain compatibility.
- Example:

```
@ViewChild('myElement') myElement: ElementRef;
ngAfterViewInit() {
  this.renderer.setStyle(this.myElement.nativeElement, 'color', 'red');
}
```

107. How would you control the size of an element on the resize of the window in a component?

- Use Angular's HostListener to listen to window resize events.
- Adjust the element size dynamically in the handler.
- Example:

```
@HostListener('window:resize', ['$event'])
onResize(event: any) {
  this.width = event.target.innerWidth;
}
```

108. How to bundle an Angular app for production?

- Use ng build --prod for an optimized production build.
- Minifies, compresses, and bundles the code.
- Enables AOT and tree-shaking by default.
- Example:

```
ng build --prod --aot
```

109. How to set headers for every request in Angular?

- Use HttpInterceptor to intercept outgoing requests.
- Add headers using HttpRequest.clone() method.
- Example:

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  const cloned = req.clone({ headers: req.headers.set('Authorization', 'Bearer token') });
  return next.handle(cloned);
}
```

1. What is the difference between an Azure Tenant and Azure Subscription?

- Azure Tenant represents an instance of Azure Active Directory, used to manage users and applications across an organization.
- Azure Subscription provides access to Azure services and resources with billing tied to it.
- A tenant can have multiple subscriptions, but a subscription can only belong to one tenant.
- In my project, I linked multiple Azure subscriptions under a single tenant for unified user management.

2. What is the difference between Azure API Apps, Logic Apps, Web Apps, and Azure Functions?

- API Apps are designed for hosting and managing APIs with built-in API management features.
- Logic Apps provide workflow automation and integration without writing code.
- Web Apps are used to host and manage web applications on Azure App Service.
- I used Azure Functions in my project to execute serverless background tasks triggered by events.

3. What is Azure CDN (Content Delivery Network) and why use it?

- Azure CDN distributes content globally to reduce latency and improve load times.
- It caches content at strategically located edge servers closer to users.
- It supports static content like images, videos, or JavaScript files.
- In my project, we used Azure CDN to cache large media files for a global video streaming application.

4. What is the difference between Azure AD Application Permissions and Delegated Permissions?

- Application Permissions are used by applications to access resources without user interaction.
- Delegated Permissions are used to access resources on behalf of a signed-in user.
- Application Permissions are ideal for backend services, while Delegated Permissions are for client apps.
- I used Delegated Permissions for an Angular app requiring user consent to access Microsoft Graph.

5. What are ARM Templates in Azure?

- ARM Templates are JSON files used to define and deploy Azure infrastructure.
- They enable declarative infrastructure management and support versioning.
- They support parameterization for reusable templates across environments.
- In my project, I used an ARM template to provision a Cosmos DB account and a web app in a single deployment.

6. When would you use Azure CLI vs PowerShell? Explain.

- Azure CLI is better for cross-platform scripting and is lightweight.
- PowerShell integrates well with Windows-based environments and supports automation tasks.
- Both can manage Azure resources, but the choice depends on the environment and language preference.
- I used Azure CLI in a CI/CD pipeline for deploying microservices to Azure Kubernetes Service.

7. What is the rule of thumb for choosing the ideal partitioning key in Cosmos DB?

- Choose a key with high cardinality to distribute data evenly across partitions.
- The key should minimize cross-partition queries for better performance.
- Avoid keys that lead to hot partitions or uneven distribution.
- In my project, we used the userId field as the partition key for a multi-tenant application.

8. What is WebJob in Azure?

- WebJobs are background tasks associated with Azure App Service.
- They support running continuously or on-demand with manual or scheduled triggers.
- They are ideal for recurring tasks like data processing or cleanup.
- I used WebJobs to process and archive log files from a web application nightly.

9. Is there a way to view deployed files in Azure?

- Yes, use Kudu (Advanced Tools) in Azure App Service to access the deployed files.
- Navigate to https://<appname>.scm.azurewebsites.net for the Kudu interface.
- It provides tools for file exploration, debugging, and deployment logs.
- In one project, I used Kudu to verify if a configuration file was correctly deployed to a staging environment.

10. When would you use Azure Storage Queues vs Azure Service Bus?

- Use Azure Storage Queues for simple, lightweight, and scalable messaging.
- Use Azure Service Bus for advanced features like dead-letter queues and transactional messages.
- Service Bus is better for enterprise-grade integration scenarios.
- I used Service Bus to implement reliable communication between microservices requiring guaranteed delivery.

11. What is RU (Request Unit) in Cosmos DB?

- RU (Request Unit) measures throughput in Cosmos DB for operations.
- It abstracts compute, memory, and IOPS costs into a single unit.
- Provision RUs based on workload requirements to ensure performance.
- In my project, I allocated 400 RUs to a collection handling moderate read/write operations.

12. What is the difference between Azure Queue Storage and Azure Service Bus with regards to dead-letter queues & poison messages?

- Azure Queue Storage lacks built-in dead-letter queues; you must implement custom logic.
- Azure Service Bus includes native dead-letter queues for message processing issues.
- Service Bus also supports automatic handling of poison messages.
- I used Service Bus dead-letter queues to diagnose failures in message processing for an e-commerce application.

13. Explain the difference between Event vs Message Services in the context of Azure Services.

- Events are lightweight notifications indicating state changes (e.g., Azure Event Grid).
- Messages contain raw data for processing, often between applications (e.g., Azure Service Bus).
- Use events for broadcast notifications and messages for reliable data delivery.
- In my project, Event Grid was used to notify microservices of database changes.

14. What's the difference between Azure SQL Database and Azure SQL Managed Instance?

- Azure SQL Database is a fully managed PaaS for single databases or elastic pools.
- Azure SQL Managed Instance provides near 100% compatibility with on-premises SQL Server.
- Use Managed Instance for migrating legacy applications; SQL Database is better for cloud-native apps.

- I used SQL Managed Instance to lift and shift an on-premises ERP system to Azure.

15. How to do a transaction with two collections on Azure CosmosDB?

- Cosmos DB supports multi-document transactions within a single logical partition.
- Use stored procedures for cross-collection operations within the same partition.
- Cross-partition transactions require custom handling or external frameworks.
- In my project, I created a stored procedure to update inventory and order records atomically.

```
function updateDocuments(order, inventory) {
  var context = getContext();
  var collection = context.getCollection();

  var isAccepted = collection.createDocument(collection.getSelfLink(),
order, function(err) {
      if (err) throw new Error("Failed to create order");
  });
  if (!isAccepted) throw new Error("Order creation was not accepted");

  isAccepted = collection.replaceDocument(inventory._self, inventory,
function(err) {
      if (err) throw new Error("Failed to update inventory");
  });
  if (!isAccepted) throw new Error("Inventory update was not accepted");
}
```

16. What is Azure Web PubSub Service? When shall you use it?

- Azure Web PubSub Service enables real-time web communication over WebSocket.
- It's suitable for scenarios requiring low-latency updates, like chat apps or live dashboards.
- It supports multiple client protocols and integrations with existing systems.
- I used Web PubSub to deliver live score updates for a sports application.

17. What Is Azure Resource Manager (ARM)?

- ARM is the deployment and management service for Azure resources.
- It uses declarative templates for creating, updating, or deleting resources.
- ARM ensures resource dependency management and consistent deployments.
- In my project, I used ARM to deploy a full-stack application, including a web app and database.

18. How can I test my ARM template before deploying it?

- Use the az deployment what-if command to preview changes without deploying.
- Validate the template structure using az deployment validate.
- Deploy in a test environment first to ensure correctness.
- I used az deployment what-if to identify potential misconfigurations before deploying a staging environment.

19. What is a Logical Partition in Cosmos DB?

- A logical partition groups data with the same partition key for efficient querying.
- It provides a scope for transactions and local indexing.
- Each logical partition can store up to 20GB of data in Cosmos DB.
- I used the orderId as a partition key in an e-commerce project to group all order-related items together.

20. When would you use Azure Event Grid vs Azure Service Bus and vice versa?

- Use Event Grid for event-driven architectures with high throughput and low latency.
- Service Bus is better for message-driven architectures with guaranteed delivery.
- Event Grid is ideal for broadcasting events, while Service Bus handles complex workflows.
- In one project, Event Grid was used to notify microservices of resource changes, while Service Bus handled order processing.

21. What's the difference between Logical and Physical partitions in Cosmos DB?

- Logical partitions group data based on partition key; they are user-defined.
- Physical partitions are managed by Cosmos DB and store logical partitions.
- Physical partitions can span multiple logical partitions for scalability.
- In a multi-tenant app, logical partitions represented tenants, while physical partitions were abstracted by Cosmos DB.

22. Explain the difference between At-most-once vs At-least-once message processing in Azure Service Bus.

- At-most-once ensures no duplicate message delivery but may drop messages.
- At-least-once ensures every message is processed but may result in duplicates.
- Use at-most-once for idempotent operations and at-least-once for critical tasks.
- I implemented at-least-once processing to ensure reliable order placement in a retail system.

23. What does "Cosmos DB automatically indexes the documents" mean? Explain.

- Cosmos DB indexes all data by default for efficient queries.
- This eliminates the need for manual index management.
- You can customize indexing policies to include or exclude specific fields.
- I disabled indexing for large blobs in a document to optimize query performance in a logging system.

24. Name some pros and cons of using GUID as Partition Key in Cosmos DB.

- Pros: Ensures high cardinality, avoids hot partitions, and distributes data evenly.
- Cons: Difficult to query by GUID and lacks semantic meaning.
- It's suitable for scenarios prioritizing write scalability over query efficiency.
- In one project, I used GUIDs for write-intensive logs to achieve even partitioning.

25. Using Azure Functions, can I reference and use NuGet packages in my C# function?

- Yes, NuGet packages can be referenced by adding them to the function.proj file.
- Use the #r directive for package references in the script-based approach.
- Ensure the package version aligns with the Azure Functions runtime.
- I referenced Azure.Storage.Blobs in an Azure Function to process uploaded files.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Azure.Storage.Blobs" Version="12.12.0" />
  </ItemGroup>
</Project>
```

26. What are Web Role, Worker Role, and VM Role in Azure?

- Web Role is for hosting web applications using IIS.
- Worker Role runs background tasks independent of IIS.

- VM Role provides OS-level control for custom software installations.
- I used Worker Roles for data processing tasks in an IoT solution.

27. What is the difference between an API App and a Web App?

- API Apps specialize in hosting APIs with enhanced API management.
- Web Apps focus on hosting and managing full-stack web applications.
- Both run on Azure App Service but have distinct optimizations.
- I deployed an API App to manage backend services for a mobile application.

28. What are On Demand WebJobs, Scheduled WebJobs, and Triggered WebJobs in Azure?

- On Demand WebJobs run manually via Azure portal or API.
- Scheduled WebJobs run at specified intervals using CRON expressions.
- Triggered WebJobs are invoked by external events like queue messages.
- In my project, Scheduled WebJobs generated daily reports for analytics.

29. What are the differences between Classic and Storage Accounts in Azure?

- Classic accounts use older deployment models with limited scalability.
- Storage accounts support the ARM model and advanced features.
- Modern storage accounts integrate with Azure RBAC and diagnostic logs.
- I migrated classic accounts to ARM-based accounts for better governance and security.

30. What to use: many small Azure Storage Blob containers vs one really large container with tons of blobs?

- Use many small containers for better organization and access control.
- Use a large container for simpler management when blobs have uniform access patterns.
- Consider performance implications for scenarios with high throughput.
- I opted for separate containers to segregate media files by department in a media management system.

31. How would you choose between Azure Blob Storage vs Azure File Service?

- Use Azure Blob Storage for unstructured data like media files, backups, or logs.
- Use Azure File Service for file shares that require SMB protocol and directory structures.
- Blob Storage offers more scalability, while File Service supports lift-and-shift scenarios.
- I used Azure File Service to provide shared storage for legacy applications during migration.

32. What is the difference between an Azure Web App and an Azure Web Role?

- Azure Web App is a PaaS offering to host web applications with built-in scaling and monitoring.
- Azure Web Role is part of the deprecated Cloud Services model used to host IIS-based apps.
- Web Apps are easier to manage and integrate with modern Azure services.
- I migrated a Web Role-based application to Azure Web App for improved manageability and features.

33. When to choose Worker Role vs Web Jobs on Azure?

- Worker Roles are standalone VMs for intensive background tasks (deprecated).
- WebJobs run within the App Service environment for lightweight background processing.
- WebJobs integrate better with modern services and are easier to deploy and scale.
- I used WebJobs for periodic data synchronization in a SaaS platform.

34. Explain the use of Claim Check Pattern in Azure Event Grid.

- The Claim Check Pattern offloads large payloads to storage while sending a lightweight reference in events.
- It improves performance and avoids size limits in messaging systems.
- Combine Event Grid with Blob Storage to implement this pattern.
- I used this pattern to send metadata in events while storing detailed logs in Azure Blob Storage.

35. Compare Azure WebJobs vs Azure Function. When to use one?

- WebJobs are ideal for long-running tasks tied to an App Service.
- Azure Functions are event-driven, scalable, and suitable for serverless architectures.
- Use WebJobs for simpler scenarios and Functions for flexibility and scalability.
- I used Azure Functions to handle real-time event processing in an IoT project.

36. When shall we use Azure Table Storage over Azure SQL?

- Use Table Storage for key-value or NoSQL data requiring high scalability.
- Azure SQL is better for relational data and complex queries.
- Table Storage is cost-effective for large, simple datasets.
- I used Table Storage to log IoT telemetry data for cost efficiency and speed.

37. What is the difference between Keys and Secrets in Azure Key Vault?

- Keys are used for encryption, signing, and cryptographic operations.
- Secrets store sensitive information like connection strings or passwords.
- Keys have cryptographic features, while secrets are for secure storage and retrieval.
- I stored database connection strings as secrets in Azure Key Vault for secure access.

38. How is Azure App Services different from Azure Functions?

- Azure App Services is a PaaS offering for web apps, APIs, and mobile backends.
- Azure Functions is a serverless compute service for running event-driven tasks.
- App Services offer broader hosting options, while Functions are lightweight and cost-efficient.
- I used App Services for hosting a multi-tenant SaaS platform and Functions for event processing.

39. What is the equivalent of a Windows Service on Azure?

- Azure Functions or WebJobs can act as equivalents for background tasks.
- Azure VMs can host traditional Windows Services for legacy applications.
- Azure Container Instances can also run custom services in isolated environments.
- I replaced a legacy Windows Service with an Azure Function for processing uploaded files.

40. What data schema shall you use for Event Grid events?

- Follow the CloudEvents schema for standardization and interoperability.
- Include minimal metadata for routing and processing efficiency.
- Store large payloads externally using the Claim Check Pattern.

- I designed custom events with CloudEvents schema to standardize inter-service communication.

41. What are main considerations when creating Azure Service Bus queues/topics?

- Define message size, retention policies, and maximum queue size.
- Use sessions for ordered message processing and dead-letter queues for fault handling.
- Configure access policies for secure message handling.
- I created Service Bus queues with custom TTL settings to optimize resource usage for temporary messages.

42. Explain the difference between Scheduled vs Deferred messages in Azure Service Bus.

- Scheduled messages are delayed for a specific time before becoming available for consumption.
- Deferred messages are postponed by the receiver and require a sequence number to be retrieved.
- Scheduled messages are set at sending time; deferred messages require manual retrieval logic.
- I used scheduled messages to delay notifications in a time-sensitive workflow.

43. Explain the use of Express Entities in Azure Service Bus.

- Express entities optimize performance by storing messages in memory temporarily.
- They are suitable for lightweight, high-throughput scenarios.
- Persistent storage is sacrificed for speed, so use them for non-critical workloads.
- I used express queues in a high-velocity telemetry pipeline where message loss was tolerable.

44. Why should you keep Azure Event Grid events as small as possible?

- Small events reduce latency and improve throughput.
- They prevent exceeding size limits and reduce processing overhead.
- Use external storage for large payloads to adhere to size constraints.
- I implemented compact events for notifying clients of file uploads, with URLs pointing to storage locations.

45. How would you get Azure Event Grid events in Angular/React client?

- Use Azure SignalR or Web PubSub to forward events to clients in real time.
- Poll a custom API or use WebSocket for direct communication.
- Implement subscription verification for secure Event Grid integration.
- I forwarded Event Grid events via Web PubSub to display live updates in an Angular dashboard.

46. What are pros and cons of Azure Web PubSub vs SignalR?

- Web PubSub offers native WebSocket support, high scalability, and integration flexibility.
- SignalR simplifies development but is tightly coupled with the .NET ecosystem.
- Web PubSub is more suitable for diverse clients and lightweight architectures.
- I used Web PubSub for a multi-platform chat application requiring WebSocket-based communication.

47. Explain the use of Bounded Staleness consistency model in Cosmos DB.

- Ensures reads lag behind writes by a defined staleness window or operation count.
- Provides a balance between strong consistency and high availability.
- Suitable for scenarios tolerating delayed reads with globally distributed systems.
- I used Bounded Staleness to sync a global inventory system while maintaining high write throughput.

48. How would you choose/design a multi-tenant architecture with Azure Cosmos DB?

- Use partition keys to isolate tenant data for scalability and cost-efficiency.
- Leverage shared throughput databases for cost sharing among tenants.
- Implement RBAC for secure tenant-level access.
- I designed a multi-tenant platform using tenantId as the partition key to segregate customer data.

49. Compare ARM Templates vs Azure CLI for Azure deployments.

- ARM Templates are declarative and reusable; ideal for complex, repeatable deployments.
- Azure CLI is imperative and better for ad-hoc or interactive resource management.
- ARM Templates ensure consistency; CLI offers flexibility for quick changes.
- I used ARM Templates to define a standardized infrastructure for a development environment.

50. What is the difference between Azure Active Directory and Azure Active Directory Domain Services?

- Azure AD is a cloud-based identity and access management service for apps.
- Azure AD Domain Services provides domain-join, LDAP, and NTLM/Kerberos authentication.
- Use Azure AD for modern apps and Domain Services for legacy apps requiring on-prem-style authentication.
- I used Azure AD for SSO in a modern web application and Domain Services for legacy ERP integration.

51. Name some advantages and disadvantages of Azure CDNs.

- Advantages: Improved performance, global availability, and reduced latency for static assets.
- Disadvantages: Costs increase with traffic, and changes to content may take time to propagate.
- CDNs are ideal for serving static assets like images, videos, or scripts.
- I implemented Azure CDN to optimize content delivery for a high-traffic e-commerce website.

52. Cosmos DB vs Azure Table Storage vs Azure SQL Database: what to choose?

- Cosmos DB: For globally distributed, low-latency applications with NoSQL flexibility.
- Azure Table Storage: For key-value pairs in cost-effective scenarios with simple querying.
- Azure SQL Database: For relational data requiring strong consistency and complex querying.
- I used Cosmos DB for a multi-region chat app, Table Storage for logs, and SQL for transactional data.

53. How would you choose between Azure Table Storage vs MongoDB?

- Azure Table Storage is simpler, cost-effective, and integrates natively with Azure services.
- MongoDB offers advanced querying, aggregation, and indexing capabilities.
- Use Table Storage for lightweight key-value scenarios and MongoDB for complex NoSQL workloads.
- I chose Table Storage for a logging system and MongoDB for a social media analytics platform.

54. Explain the difference between Block Blobs, Append Blobs, and Page Blobs in Azure.

- Block Blobs: Optimized for sequential uploads; ideal for text and media files.

- Append Blobs: Specialized for append-only operations; suitable for logging.
- Page Blobs: Optimized for random read/write operations; used for virtual disks.
- I used Append Blobs to collect real-time telemetry data from IoT devices.

55. When should I use Azure SQL vs Azure Table Storage?

- Use Azure SQL for relational data, transactional requirements, and complex querying.
- Use Table Storage for simple key-value storage with high scalability and low cost.
- Azure SQL ensures data integrity, while Table Storage excels in high-throughput scenarios.
- I stored product catalogs in SQL for querying and session data in Table Storage for scalability.

56. What is the difference between Enterprise Application and App Registration in Azure?

- Enterprise Application represents an instance of an app in a tenant for SSO.
- App Registration is a template to define app permissions and properties.
- Enterprise Apps are tenant-specific, while App Registrations can be multi-tenant.
- I created an App Registration for a multi-tenant API and used Enterprise Apps for client onboarding.

57. Explain Optimistic Concurrency Control (OCC) and how it is implemented in Cosmos DB.

- OCC ensures data consistency by using ETags to detect changes.
- Updates fail if the ETag doesn't match, indicating a conflict.
- Suitable for scenarios with low contention and high performance.
- I used OCC in a collaborative editing app to prevent overwrites of concurrent updates.

```
var itemResponse = await container.ReadItemAsync<MyItem>(id,
partitionKey);
var item = itemResponse.Resource;
item.SomeField = "Updated Value";
await container.ReplaceItemAsync(item, id, new
PartitionKey(item.PartitionKey), new ItemRequestOptions { IfMatchEtag =
itemResponse.ETag });
```

58. Why are Azure Resource Groups associated with a specific region?

- Resource Groups store metadata about resources and their locations.
- Associating with a region ensures optimal performance and latency.
- Resources within a group can span regions, but group metadata resides in one region.
- I deployed a global app with all resources in a group based in the East US region for performance reasons.

59. How to define an Environment Variable on Azure using Azure CLI?

- Use az webapp config appsettings set for Azure App Service environment variables.
- CLI allows defining key-value pairs for app-specific configurations.
- These settings override environment variables at runtime.
- I set DatabaseConnectionString using the following command for a production deployment.

```
az webapp config appsettings set --name MyWebApp --resource-group
MyResourceGroup --settings
DatabaseConnectionString="Server=tcp:mydb.database.windows.net;..."
```

60. When would you use Azure Event Grid vs Azure Service Bus?

- Event Grid is for reactive event-driven architectures with high throughput.
- Service Bus is for command-style messaging and reliable workflows.
- Event Grid suits publish-subscribe models, while Service Bus handles complex messaging patterns.
- I used Event Grid for resource change notifications and Service Bus for order processing in an e-commerce system.

61. What's the difference between Logical and Physical partitions in Cosmos DB?

- Logical partitions group data by a partition key; users define them.
- Physical partitions are Azure-managed, storing one or more logical partitions.
- Physical partitions scale automatically as data volume grows.
- I used userId as a logical partition to segregate user data, while physical partitions handled scalability transparently.

62. Explain the difference between Scheduled vs Deferred messages in Azure Service Bus.

- Scheduled messages delay visibility until a set time, useful for time-based actions.
- Deferred messages are manually postponed by the receiver and retrieved using a sequence number.
- Scheduled messages are ideal for reminders; deferred messages help manage workflows.
- I scheduled messages for payment retries and deferred error-handling messages in a payment gateway.

63. Explain the use of Express Entities in Azure Service Bus.

- Express entities store messages in memory for faster processing.
- They are suitable for scenarios where durability is not critical.
- Message retention is limited compared to standard entities.
- I used express queues for low-latency telemetry in a monitoring system.

64. Why should you keep Azure Event Grid events as small as possible?

- Small events reduce latency and minimize processing overhead.
- Large events may hit size limits and require external storage.
- Smaller payloads are easier to transmit and process in real time.
- I sent metadata in Event Grid events and stored detailed data in Blob Storage to optimize performance.

65. How would you get Azure Event Grid events in Angular/React client?

- Use Azure Web PubSub or SignalR to push events to the frontend in real time.
- Implement a WebSocket or polling strategy for receiving events.
- Secure the endpoint using Event Grid subscriptions and shared keys.
- I built a React app that consumed Event Grid notifications via a SignalR hub for live updates.

1. What are Property Accessors in C#?

- Property accessors allow controlled access to class properties.
- They use get and set keywords for reading and modifying property values.
- You can implement logic inside accessors for validation or other operations.
- *Example*:

```
        public class Person
        {
    private int age;
          public int Age
          {
            get { return age; }
            set { if (value > 0) age = value; }
          }
        }
```

// Applied in my project to validate user inputs for age.

2. What is an Object in C#?

- An object is an instance of a class that encapsulates data and methods.
- It is created using the new keyword.
- Objects allow interaction with data defined in classes.
- *Example*:
- Person person = new Person();
- person.Age = 30;
- Console.WriteLine(person.Age);
- *// Used objects to manage student data in my education portal project.*

3. What is the difference between continue and break statements in C#?

- continue skips the current iteration of a loop and moves to the next.
- break exits the loop immediately.
- Both improve control flow in looping structures.
- *Example*:
- for (int i = 0; i < 5; i++)
- {
-    if (i == 2) continue;
-    if (i == 4) break;
-    Console.WriteLine(i);
- }
- *// Utilized these in a task scheduler to manage incomplete tasks.*

4. What is C#?

- C# is a modern, object-oriented programming language developed by Microsoft.
- It runs on the .NET platform and supports cross-platform development.
- Features include type safety, garbage collection, and LINQ support.
- *Example*:
- Console.WriteLine("Hello, World!");
- *// Used C# to build APIs for microservices architecture in my cloud solutions project.*

5. What do you understand by Value types and Reference types in .NET? Provide some comparison.

- Value types store data directly, while reference types store references to data.
- Value types include int, float, and struct, whereas reference types include class and string.
- Value types are stored in the stack, reference types in the heap.
- *Example*:
- int a = 5;
- string b = "Hello";
- *// Differentiated storage requirements in a high-performance computation project.*

6. What are Generics in C#?

- Generics allow defining reusable classes or methods with type parameters.
- They increase code reusability and type safety.
- Examples include List<T> and Dictionary<TKey, TValue>.
- *Example*:
- public void PrintItems<T>(List<T> items)
- {
-    foreach (var item in items)
-       Console.WriteLine(item);
- }
- *// Used generics to handle collections in a data processing library.*

7. How is Exception Handling implemented in C#?

- Exception handling uses try, catch, and finally blocks.
- try encapsulates code that may throw exceptions.
- catch handles specific exceptions, and finally executes cleanup code.
- *Example*:
- try { int x = int.Parse("abc"); }
- catch (FormatException ex) { Console.WriteLine(ex.Message); }
- finally { Console.WriteLine("Done"); }
- *// Applied in my project to handle file I/O errors gracefully.*

8. Why use the finally block in C#?

- The finally block ensures the execution of cleanup code regardless of exceptions.
- It is optional but recommended for resource management.
- Commonly used for releasing resources like file streams or database connections.
- *Example*:
- StreamReader sr = null;
- try { sr = new StreamReader("file.txt"); }
- finally { sr?.Dispose(); }
- *// Implemented in a logging utility to handle resource management.*

9. What are partial classes in C#?

- Partial classes allow splitting a class definition into multiple files.
- All parts are combined into a single class during compilation.
- Useful for managing large classes and auto-generated code.
- *Example*:
- *// File1.cs*
- public partial class Demo { public void Method1() { } }
- *// File2.cs*
- public partial class Demo { public void Method2() { } }
- *// Utilized in an ASP.NET Core project to separate logic and UI code.*

10. Can this be used within a static method in C#?

- No, this refers to the instance of a class and is inaccessible in static methods.
- Static methods are bound to the class, not the instance.
- Use the class name directly in static methods instead.

- *Example*:
- public static void PrintClassName() { Console.WriteLine(typeof(MyClass).Name); }
- *// Applied in my project for utility methods.*

11. Can multiple catch blocks be executed?

- No, only the first matching catch block executes for a single exception.
- Further catch blocks are ignored.
- Ensure specific exceptions are caught before general ones.
- *Example*:
- try { int x = int.Parse("abc"); }
- catch (FormatException) { Console.WriteLine("Format issue."); }
- catch (Exception) { Console.WriteLine("General exception."); }
- *// Handled various exception types in a payment gateway project.*

12. What is Serialization in C#?

- Serialization converts objects into a format (like JSON or XML) for storage or transmission.
- Deserialization reconstructs objects from the serialized format.
- Commonly used for saving application state or transferring data.
- *Example*:
- string json = JsonSerializer.Serialize(obj);
- var obj = JsonSerializer.Deserialize<MyClass>(json);
- *// Used serialization in API communication for data exchange.*

13. What are the different types of classes in C#?

- Static classes: For utility methods and no instances.
- Abstract classes: Provide base functionality for derived classes.
- Sealed classes: Prevent inheritance.
- *Example*:
- public static class Utils { public static void Print() { } }
- *// Implemented static classes for reusable utility functions.*

14. What is Managed or Unmanaged Code in C#?

- Managed code runs under the control of the CLR, ensuring memory safety.
- Unmanaged code executes directly on the OS without CLR oversight.
- Managed code benefits from features like garbage collection.
- *Example*:
- *// Managed*
- string text = "Hello";
- *// Used unmanaged code through P/Invoke in a native library integration project.*

15. What are Reference Types in C#?

- Reference types store memory addresses, not the actual data.
- Examples include class, interface, and string.
- They allow sharing of the same data across multiple references.
- *Example*:
- string s1 = "Hello";
- string s2 = s1;
- *// Managed references in a caching mechanism for performance optimization.*

16. What is LINQ in C#?

- LINQ (Language-Integrated Query) is used to query data collections like arrays, lists, or databases.
- It provides a unified syntax for querying various data sources.
- Commonly used LINQ methods include Select, Where, OrderBy, and GroupBy.
- *Example*:
- var numbers = new List<int> { 1, 2, 3, 4 };
- var evenNumbers = numbers.Where(n => n % 2 == 0);
- *// Utilized LINQ to filter and process data in a reporting module.*

17. What is the difference between string and StringBuilder in C#?

- string is immutable; every modification creates a new instance.
- StringBuilder is mutable and optimized for frequent modifications.
- Use string for static or fewer changes; use StringBuilder for dynamic content.
- *Example*:
- StringBuilder sb = new StringBuilder("Hello");
- sb.Append(" World");
- Console.WriteLine(sb.ToString());
- *// Applied `StringBuilder` to dynamically generate large HTML strings.*

18. What is Boxing and Unboxing in C#?

- Boxing converts a value type to a reference type by wrapping it in an object.
- Unboxing extracts the value type from the boxed object.
- Both operations affect performance due to type conversion overhead.
- *Example*:
- int num = 10;
- object obj = num; *// Boxing*
- int newNum = (int)obj; *// Unboxing*
- *// Managed type conversions efficiently in a generic computation engine.*

19. What is the difference between a class and a structure in C#?

- A class is a reference type; a structure is a value type.
- Classes support inheritance; structures do not.
- Classes are stored on the heap, while structures are stored on the stack.
- *Example*:
- public struct Point { public int X, Y; }
- public class Shape { public int Width, Height; }
- *// Chose structures to represent lightweight coordinate data in a graphics application.*

20. What is the difference between a Struct and a Class in C#?

- Structs are value types, whereas classes are reference types.
- Structs do not support inheritance, while classes do.
- Structs are better for lightweight objects; classes for complex types.
- *Example*:
- struct Employee { public int Id; }
- class Department { public string Name; }
- *// Used structs for fixed, small data elements in a scientific computation app.*

21. What is an Abstract Class in C#?

- An abstract class provides a base for other classes but cannot be instantiated.
- It may contain abstract methods (without implementation) and concrete methods.
- Abstract classes enforce derived classes to implement certain functionality.
- *Example*:
- public abstract class Animal { public abstract void Speak(); }

- public class Dog : Animal { public override void Speak() => Console.WriteLine("Bark"); }
- *// Designed abstract classes for consistent behavior across animal types in a simulation.*

22. What is a namespace in C#?

- A namespace organizes code into a logical structure to prevent naming conflicts.
- It is declared using the namespace keyword.
- Namespaces can be nested and are commonly used in large projects.
- *Example*:
- namespace MyApp.Models { public class User { public string Name; } }
- *// Used namespaces to organize models, services, and utilities in a web application.*

23. What are Nullable types in C#?

- Nullable types allow value types to hold null.
- They are declared using ?, e.g., int?.
- Useful for representing the absence of a value, such as in databases.
- *Example*:
- int? age = null;
- Console.WriteLine(age.HasValue ? age.Value : "No Age");
- *// Managed nullable fields in a customer record system to represent optional data.*

24. In how many ways can you pass parameters to a method in C#?

- By value (default): A copy of the variable is passed.
- By reference (ref): The original variable can be modified.
- Out parameters (out): Used for returning multiple values.
- *Example*:
- void UpdateValue(ref int value) { value = 10; }
- int num = 5;
- UpdateValue(ref num);
- *// Applied ref parameters to optimize reusable utilities.*

25. What are dynamic type variables in C#?

- The dynamic type allows variables to hold any type, determined at runtime.
- It bypasses compile-time type checking.
- Use with caution as it may cause runtime errors.
- *Example*:
- dynamic data = 5;
- data = "Hello";
- Console.WriteLine(data);
- *// Utilized dynamic types to handle JSON data with unknown structures.*

26. What is an enum in C#?

- An enum is a value type that defines a set of named constants.
- It improves code readability and maintainability.
- Enums are strongly typed and cannot be implicitly converted to other types.
- *Example*:
- enum Status { Active, Inactive, Suspended }
- Status userStatus = Status.Active;
- *// Used enums to represent order statuses in an e-commerce platform.*

27. Is there a way to catch multiple exceptions at once without code duplication?

- Yes, use multiple catch blocks or a single block with a common base class like Exception.
- Pattern matching can also differentiate exceptions in a single block.
- Avoid excessive generalization to maintain clarity.
- *Example*:
- try { */* Code */* }
- catch (FormatException) { */* Handle format issues */* }
- catch (Exception ex) when (ex is IOException || ex is TimeoutException)
- { Console.WriteLine("File or timeout error"); }
- *// Centralized error handling in an email client.*

28. Explain assignment vs shallow copy vs deep copy for a Record in C#?

- Assignment: Both variables point to the same object in memory.
- Shallow copy: Copies the object but not its nested objects.
- Deep copy: Copies the object and all nested objects.
- *Example*:
- var record1 = new MyRecord(1, "Data");
- var record2 = record1 with { }; *// Shallow copy using `with`.*
- *// Applied these concepts to clone complex objects in a configuration manager.*

29. When to use Record, Class, or Struct in C#?

- Use Record for immutable data objects with value-based equality.
- Use Class for reference types requiring flexibility and inheritance.
- Use Struct for lightweight objects and small data representations.
- *Example*:
- record Customer(string Name, int Age);
- struct Point { public int X, Y; }
- *// Designed customer data as Records for immutability in a retail system.*

30. Why can't you specify the accessibility modifier for methods inside the Interface in C#?

- All interface methods are implicitly public and abstract.
- Accessibility modifiers are not allowed to enforce uniform access.
- Interfaces define a contract, not implementation.
- *Example*:
- public interface IAnimal { void Speak(); }
- *// Ensured consistent access rules in service interface design for microservices.*

31. What is Record in C#?

- A Record is a reference type designed for immutable data models with value-based equality.
- Introduced in C# 9.0, it simplifies the creation of DTOs (Data Transfer Objects).
- Supports "with" expressions to create modified copies of records.
- *Example*:
- public record Employee(string Name, int Age);
- var emp1 = new Employee("John", 30);
- var emp2 = emp1 with { Age = 31 };
- *// Used Records for designing immutability in financial reporting systems.*

32. What is an anonymous function in C#?

- An anonymous function is a function without a name, such as a lambda expression or a delegate.
- Used for inline code logic and event handling.
- Improves code readability in cases of short, reusable logic.
- *Example*:
- Func<int, int> square = x => x * x;
- Console.WriteLine(square(5));
- *// Leveraged anonymous functions to implement quick calculations in a dashboard app.*

33. What is the use of the IDisposable interface?

- It defines a Dispose method for releasing unmanaged resources.
- Commonly implemented in classes handling files, streams, or database connections.
- Ensures deterministic cleanup of resources, often used with using statements.
- *Example*:
- public class FileManager : IDisposable
- {
-    private FileStream _file;
-    public FileManager(string path) => _file = new FileStream(path, FileMode.Open);
-    public void Dispose() => _file?.Dispose();
- }
- *// Implemented IDisposable in resource-heavy file processing services.*

34. Explain the difference between Task and Thread in .NET.

- A Thread represents a single execution path in a process.
- A Task represents an asynchronous operation, often leveraging threads.
- Tasks are higher-level abstractions and integrate well with the async/await model.
- *Example*:
- var task = Task.Run(() => Console.WriteLine("Running in Task"));
- task.Wait();
- *// Used tasks to optimize background operations in a real-time analytics tool.*

35. What is a sealed class in C#?

- A sealed class cannot be inherited, ensuring its behavior remains unchanged.
- It improves security and prevents misuse in extensible systems.
- Marked with the sealed keyword before the class definition.
- *Example*:
- public sealed class Logger { public void Log(string message) => Console.WriteLine(message); }
- *// Used sealed classes for logging to prevent unintended modifications in production systems.*

36. What is the difference between overloading and overriding in C#?

- Overloading allows multiple methods in the same class with different parameter lists.
- Overriding modifies a base class method in a derived class with the override keyword.
- Overloading occurs at compile-time; overriding happens at runtime.
- *Example*:
- public class Base { public virtual void Display() => Console.WriteLine("Base"); }
- public class Derived : Base { public override void Display() => Console.WriteLine("Derived"); }
- *// Overriding was used in a polymorphic system for dynamic dispatch.*

37. What is a lambda expression in C#?

- A lambda expression is a concise way to represent an anonymous method.
- Syntax: (parameters) => expression or statement block.
- Commonly used in LINQ queries and functional programming.
- *Example*:
- var square = (int x) => x * x;
- Console.WriteLine(square(5));
- *// Applied lambda expressions to streamline query operations in a repository pattern.*

38. How is encapsulation implemented in C#?

- Encapsulation restricts direct access to class members by using access modifiers.
- Members are exposed via properties and methods.
- Improves code maintainability and security by hiding internal implementation.
- *Example*:
- public class Account
- {
-    private decimal balance;
-    public decimal Balance { get => balance; private set => balance = value; }
-    public void Deposit(decimal amount) => Balance += amount;
- }
- *// Encapsulation ensured secure updates to bank account data in a finance application.*

39. What is Reflection in C#?

- Reflection allows the inspection and manipulation of metadata at runtime.
- It enables accessing information about assemblies, types, and members.
- Commonly used for dynamic type discovery and late binding.
- *Example*:
- var type = typeof(String);
- Console.WriteLine($"Methods of {type}: {string.Join(", ", type.GetMethods().Select(m => m.Name))}");
- *// Used reflection to dynamically load plugins in a modular application.*

40. How can you prevent a class from being overridden in C#?

- Mark the class with the sealed keyword.
- Alternatively, mark specific methods as sealed in derived classes.
- Prevents unintended modifications to critical functionality.
- *Example*:
- public sealed class FinalClass { public void Show() => Console.WriteLine("Cannot override me."); }
- *// Designed sealed classes for finalizing implementations in core utilities.*

41. What is the use of the Null Coalescing Operator (??) in C#?

- It provides a default value when a nullable type or expression is null.
- Syntax: value ?? defaultValue.
- Simplifies null checks and fallback value assignments.
- *Example*:
- string name = null;
- Console.WriteLine(name ?? "Unknown");

- *// Employed null coalescing for safe fallback handling in customer data processing.*

42. What is a Destructor in C# and when should I create one?

- A destructor is used to release unmanaged resources when an object is garbage collected.
- Syntax: ~ClassName().
- Use only when handling unmanaged resources not covered by IDisposable.
- *Example*:
- public class FileHandler
- {
-    ~FileHandler() { Console.WriteLine("Destructor called."); }
- }
- *// Implemented destructors sparingly to finalize unmanaged resources in legacy components.*

43. What is the difference between Interface and Abstract Class in C#?

- Interfaces cannot contain implementation; abstract classes can have partial implementation.
- A class can implement multiple interfaces but inherit only one abstract class.
- Use interfaces for defining contracts and abstract classes for common behavior.
- *Example*:
- public interface IDrive { void Drive(); }
- public abstract class Vehicle { public abstract void StartEngine(); }
- *// Designed abstract base classes and interfaces for flexible hierarchy in a transport system.*

44. What is the difference between constant and readonly in C#?

- const values are compile-time constants, while readonly values are runtime constants.
- const must be assigned at declaration, readonly can be set in the constructor.
- const is static by default; readonly applies to instance or static fields.
- *Example*:
- public const double Pi = 3.14;
- public readonly DateTime CreatedOn = DateTime.Now;
- *// Used readonly fields to set initialization-time values in configuration classes.*

45. Explain the Anonymous type in C#.

- An anonymous type is a lightweight object without explicitly defining its type.
- Syntax: var obj = new { Property1 = Value1, Property2 = Value2 };.
- Commonly used in LINQ projections.
- *Example*:
- var person = new { Name = "John", Age = 30 };
- Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
- *// Utilized anonymous types for ad-hoc query results in a reporting system.*

46. Is there a difference between throw and throw ex in C#?

- throw preserves the original stack trace of the exception.
- throw ex resets the stack trace, losing the original exception's context.
- Always prefer throw unless there's a specific need to reset or log.
- *Example*:
- try
- {
-    int x = 0;
-    int y = 10 / x;
- }
- catch (Exception ex)
- {
-    Console.WriteLine("Logging exception");
-    throw; *// Keeps the original stack trace.*
- }

47. Explain Code Compilation in C#.

- C# code compilation involves transforming source code into Intermediate Language (IL).
- The compiler generates an assembly (.exe or .dll) containing IL.
- At runtime, the Common Language Runtime (CLR) converts IL to machine code using Just-In-Time (JIT) compilation.
- *Example*:
- Source Code → C# Compiler → IL (Intermediate Language) → JIT Compilation → Machine Code

48. What is the difference between Equality Operator (==) and Equals() method in C#?

- == compares object references for reference types and values for value types.
- Equals() can be overridden for custom comparison logic.
- Use Equals() for value-based equality in complex objects.
- *Example*:
- string s1 = "hello";
- string s2 = "hello";
- Console.WriteLine(s1 == s2); *// True*
- Console.WriteLine(s1.Equals(s2)); *// True*

49. What are the uses of using in C#?

- Ensures deterministic cleanup of resources like file streams or database connections.
- Automatically calls Dispose() at the end of the block.
- Reduces the risk of resource leaks in managed code.
- *Example*:
- using (var reader = new StreamReader("file.txt"))
- {
-    Console.WriteLine(reader.ReadToEnd());
- }

50. What is the difference between dynamic type variables and object type variables in C#?

- dynamic variables bypass compile-time type checking; object requires casting.
- dynamic variables allow operations determined at runtime.
- Use dynamic cautiously as it reduces type safety.
- *Example*:
- dynamic dyn = "hello";
- Console.WriteLine(dyn.Length); *// No compile-time error*

51. What is a Virtual Method in C#?

- A virtual method allows overriding in derived classes.
- Declared with the virtual keyword in the base class.
- Enhances polymorphism by enabling runtime method selection.
- *Example*:
- public class Base
- {
-    public virtual void Display() => Console.WriteLine("Base");

- }
- public class Derived : Base
- {
-     public override void Display() => Console.WriteLine("Derived");
- }

52. What is the difference between Virtual and Abstract method in C#?

- A virtual method has a default implementation; an abstract method does not.
- Abstract methods are declared in abstract classes.
- Virtual methods can be optionally overridden; abstract methods must be.
- *Example*:
- public abstract class Animal
- {
-     public abstract void Speak(); *// No implementation*
- }
- public class Dog : Animal
- {
-     public override void Speak() => Console.WriteLine("Woof!");
- }

53. What is the scope of an Internal member variable of a C# class?

- internal members are accessible within the same assembly.
- They are not visible to other assemblies unless explicitly specified via InternalsVisibleTo.
- Ideal for encapsulating details within a library.
- *Example*:
- internal class MyClass
- {
-     internal void Display() => Console.WriteLine("Internal");
- }

54. What is an Extension Method in C# and how to use them?

- Adds methods to existing types without modifying them.
- Declared in static classes with the this keyword for the first parameter.
- Widely used to enhance LINQ capabilities.
- *Example*:
- public static class StringExtensions
- {
-     public static int WordCount(this string str) => str.Split(' ').Length;
- }
- Console.WriteLine("Hello World".WordCount());

55. What is the difference between ref and out keywords in C#?

- ref requires the variable to be initialized before passing.
- out requires the variable to be assigned inside the method.
- Both allow passing variables by reference, enabling modification.
- *Example*:
- void Calculate(out int x) { x = 10; }
- Calculate(out int result);
- Console.WriteLine(result); *// Outputs 10*

56. Can you create a function in C# which can accept varying numbers of arguments?

- Use the params keyword to accept a variable number of arguments.
- All arguments must be of the same type.
- Simplifies method calls for collections of data.
- *Example*:
- public void PrintParams(params int[] numbers)
- {
-     foreach (var num in numbers)
-         Console.WriteLine(num);
- }
- PrintParams(1, 2, 3, 4);

57. What are pointer types in C#?

- Pointer types directly store the memory address of another variable.
- Allowed only in unsafe context and require explicit enabling.
- Used in performance-critical scenarios like interacting with hardware.
- *Example*:
- unsafe
- {
-     int x = 10;
-     int* ptr = &x;
-     Console.WriteLine((int)ptr);
- }

58. What is the difference between Dispose and Finalize methods in C#?

- Dispose is called explicitly for releasing resources; Finalize is invoked by the GC.
- Dispose is part of the IDisposable interface.
- Finalizers should be avoided unless dealing with unmanaged resources.
- *Example*:
- public void Dispose() => fileStream.Dispose();

59. What's the difference between StackOverflowException and OutOfMemoryException in C#?

- StackOverflowException occurs due to excessive recursion.
- OutOfMemoryException happens when the heap runs out of memory.
- Both are critical and usually unrecoverable.
- *Example*:
- Recursive function without exit condition → StackOverflowException
- Large memory allocations → OutOfMemoryException

60. What is an Indexer in C#?

- Indexers enable a class to be indexed like an array.
- Defined using the this keyword with parameters.
- Used to simplify accessing collections.
- *Example*:
- public class SampleCollection
- {
-     private string[] data = new string[10];
-     public string this[int index] { get => data[index]; set => data[index] = value; }
- }

61. What is the difference between Func<string, string> and delegate in C#?

- Func<string, string> is a predefined delegate for methods that take a string as input and return a string.
- delegate is a user-defined type for referencing methods with custom signatures.
- Func is concise and reusable for standard method signatures, while delegate offers flexibility.

- *Example*:
- Func<string, string> greet = name => $"Hello, {name}!";
- Console.WriteLine(greet("John")); *// Outputs: Hello, John!*
-
- delegate string CustomGreet(string name);
- CustomGreet greetDelegate = name => $"Hi, {name}!";
- Console.WriteLine(greetDelegate("John")); *// Outputs: Hi, John!*

62. Explain what is Short-Circuit Evaluation in C#.

- In && (AND) and || (OR) operations, evaluation stops as soon as the result is determined.
- Prevents unnecessary computation and potential runtime errors.
- Applies only to logical operators, not bitwise operators like & or |.
- *Example*:
- int x = 5;
- if (x > 0 && x / 0 == 1) *// Division by zero is never evaluated due to short-circuiting.*
-     Console.WriteLine("This won't run.");

63. Explain the difference between Select and Where in LINQ.

- Select is used to transform or project data into a new form.
- Where is used to filter data based on a condition.
- Both can be combined for filtering and transformation.
- *Example*:
- var numbers = new[] { 1, 2, 3, 4 };
- var evenSquares = numbers.Where(n => n % 2 == 0).Select(n => n * n);
- Console.WriteLine(string.Join(", ", evenSquares)); *// Outputs: 4, 16*

64. What is the best practice to achieve optimal performance using Lazy objects?

- Use Lazy<T> to defer initialization until the value is accessed.
- Configure thread-safety with the appropriate constructor.
- Avoid heavy computations in the Lazy factory method that negate its benefits.
- *Example*:
- Lazy<string> lazyValue = new Lazy<string>(() => ComputeValue());
- string result = lazyValue.Value; *// ComputeValue() is called only here.*

65. What is a static constructor in C#?

- A static constructor initializes static data members of a class.
- It is called automatically before the first instance is created or any static member is accessed.
- There is no access modifier, and it cannot take parameters.
- *Example*:
- public class Example
- {
-     static Example() => Console.WriteLine("Static constructor called.");
-     public static int Value = 10;
- }

66. Explain what is Ternary Search.

- A divide-and-conquer algorithm similar to binary search but splits the array into three parts.
- Used for unimodal functions or sorted arrays.
- Less efficient than binary search due to additional comparisons.
- *Example*:
- Given an array [1, 3, 5, 7], find the target by dividing the range into three sections.

67. Explain how asynchronous tasks with async/await work in .NET.

- async marks a method as asynchronous, allowing the use of await.
- await pauses execution until the awaited task completes.
- Improves responsiveness by freeing threads for other operations during wait times.
- *Example*:
- public async Task<string> GetDataAsync()
- {
-     await Task.Delay(1000);
-     return "Data retrieved";
- }

68. What happens when we Box or Unbox Nullable types in C#?

- Boxing converts a nullable type to object.
- Unboxing assigns a boxed value back to a nullable type.
- Null values remain null after boxing/unboxing.
- *Example*:
- int? num = 5;
- object boxed = num;
- int? unboxed = (int?)boxed;

69. Can you explain the difference between Interface, Abstract Class, Sealed Class, Static Class, and Partial Class in C#?

- **Interface**: Defines a contract; no implementation.
- **Abstract Class**: Can have abstract methods and implemented methods.
- **Sealed Class**: Cannot be inherited.
- **Static Class**: Contains only static members.
- **Partial Class**: Splits the definition into multiple files.
- *Example*:
- public abstract class Shape { public abstract void Draw(); } *// Abstract*
- public interface IDrawable { void Draw(); } *// Interface*
- public sealed class Circle : Shape { public override void Draw() => Console.WriteLine("Circle"); } *// Sealed*
- public static class Utils { public static void Print() => Console.WriteLine("Static"); } *// Static*

70. How to solve Circular Reference problems in C#?

- Use weak references to prevent strong dependency cycles.
- Leverage IDisposable and proper resource management.
- Avoid circular dependencies in object graphs.
- *Example*:
- WeakReference obj = new WeakReference(new MyClass());

71. Test if a number belongs to the Fibonacci Series.

- A number n is Fibonacci if $5n^2 + 4$ or $5n^2 - 4$ is a perfect square.
- Use a helper function to check if a number is a perfect square.
- Efficient for checking membership without generating the series.
- *Example*:
- bool IsFibonacci(int n) => IsPerfectSquare(5 * n * n + 4) || IsPerfectSquare(5 * n * n - 4);
- bool IsPerfectSquare(int x) => Math.Sqrt(x) % 1 == 0;

72. What is the output of the program below? Explain.

- Provide the code snippet to explain the behavior.
- Discuss concepts like scope, static behavior, or threading as relevant.
- Illustrate edge cases where applicable.

73. Can you do Iterative Pre-order Traversal of a Binary Tree without Recursion?

- Use a stack to simulate recursion for traversing the tree.
- Push and pop nodes to visit in pre-order (root, left, right).
- Avoid recursion to prevent stack overflow for large trees.
- *Example*:
- void PreOrder(Node root)
- {
-    var stack = new Stack<Node>();
-    stack.Push(root);
-    while (stack.Count > 0)
-    {
-       Node current = stack.Pop();
-       Console.WriteLine(current.Value);
-       if (current.Right != null) stack.Push(current.Right);
-       if (current.Left != null) stack.Push(current.Left);
-    }
- }

74. Can you return multiple values from a function in C#? Provide some examples.

- Use out parameters or return a tuple for multiple values.
- Simplifies data handling when multiple results are needed.
- *Example*:
- (int, int) GetDimensions() => (5, 10);
- var (width, height) = GetDimensions();
- Console.WriteLine($"Width: {width}, Height: {height}");

75. Given an array of ints, write a C# method to total all the values that are even numbers.

- Use LINQ for concise filtering and summation.
- Enhances readability and reduces boilerplate code.
- *Example*:
- int TotalEvenNumbers(int[] numbers) => numbers.Where(n => n % 2 == 0).Sum();

76. Refactor the code provided.

- Refactoring involves improving code readability, maintainability, and performance without changing its behavior.
- Apply principles like DRY (Don't Repeat Yourself), SOLID, and design patterns.
- Ensure to write test cases before and after refactoring to confirm correctness.
- *Example*: Before:
- if (x > 0) Console.WriteLine("Positive");
- else if (x == 0) Console.WriteLine("Zero");
- else Console.WriteLine("Negative");

After:
Console.WriteLine(x > 0 ? "Positive" : x == 0 ? "Zero" : "Negative");

77. Explain how the Sentinel Search works.

- A linear search algorithm that places a sentinel (special marker) at the end of the array.
- Reduces the number of boundary checks during iteration.
- Useful for unsorted arrays with frequent searches.
- *Example*:
- int SentinelSearch(int[] arr, int key)
- {
-    int last = arr[^1];
-    arr[^1] = key;
-    int i = 0;
-    while (arr[i] != key) i++;
-    arr[^1] = last;
-    return i < arr.Length - 1 || arr[^1] == key ? i : -1;
- }

78. Reverse the ordering of words in a string.

- Split the string into words, reverse the array, and join them back.
- Handles extra spaces appropriately.
- *Example*:
- string ReverseWords(string input)
- {
-    return string.Join(" ", input.Split(' ', StringSplitOptions.RemoveEmptyEntries).Reverse());
- }
- Console.WriteLine(ReverseWords("Hello World!")); *// Outputs: World! Hello*

79. How to check if two strings (words) are anagrams?

- Two strings are anagrams if their sorted characters match.
- Ignore case and spaces for accurate comparison.
- *Example*:
- bool AreAnagrams(string str1, string str2)
- {
-    return string.Concat(str1.ToLower().OrderBy(c => c)) == string.Concat(str2.ToLower().OrderBy(c => c));
- }
- Console.WriteLine(AreAnagrams("listen", "silent")); *// Outputs: True*

80. What is the output of the short program below? Explain.

- Present the program and describe the logic step-by-step.
- Discuss specific constructs like loops, recursion, or LINQ as applicable.
- Provide a practical example showing edge cases.

81. What is the Fibonacci Search technique?

- A searching algorithm that uses Fibonacci numbers to split the array.
- Reduces the search range based on Fibonacci proportions.
- Suitable for sorted and uniformly distributed data.
- *Example*:
- Use Fibonacci numbers to narrow down search intervals in an array.

82. Is relying on && short-circuiting safe in .NET?

- Yes, it prevents evaluation of subsequent conditions if the first condition fails.
- Avoids potential runtime errors like null reference exceptions.
- Only use short-circuiting where conditions are guaranteed to follow an order.
- *Example*:
- if (obj != null && obj.Property == value) *// Safe from null reference errors.*
-    Console.WriteLine("Valid.");

83. Find the Merge (Intersection) Point of Two Linked Lists.

- Use two pointers, each traversing one list.
- When a pointer reaches the end, move it to the start of the other list.
- The intersection point is where the pointers meet.
- *Example*:
- Node FindIntersection(Node headA, Node headB)
- {

- Node p1 = headA, p2 = headB;
- while (p1 != p2)
- {
- p1 = p1 == null ? headB : p1.Next;
- p2 = p2 == null ? headA : p2.Next;
- }
- return p1;
- }

84. Binet's formula: How to calculate Fibonacci numbers without recursion or iteration?

- Binet's formula uses the golden ratio to calculate Fibonacci numbers.
- Provides an exact result for small numbers but suffers rounding errors for large indices.
- *Example*:
- double Fibonacci(int n)
- {
- double phi = (1 + Math.Sqrt(5)) / 2;
- return Math.Round(Math.Pow(phi, n) / Math.Sqrt(5));
- }

85. What is the output of the program below? Explain your answer.

- Analyze the program and explain its logic clearly.
- Discuss potential pitfalls like variable scope or initialization.
- Use comments to clarify each step of the explanation.

86. Is the comparison of time and null in an if statement valid or not? Why or why not?

- Valid if time is a nullable type or reference type.
- Invalid for value types like DateTime unless it's nullable.
- *Example*:
- DateTime? time = null;
- if (time == null) Console.WriteLine("Time is null.");

87. Calculate the circumference of a circle.

- Use the formula C = 2 * π * r, where r is the radius.
- Leverage constants like Math.PI for accuracy.
- *Example*:
- double Circumference(double radius) => 2 * Math.PI * radius;
- Console.WriteLine(Circumference(5)); *// Outputs: 31.4159...*

88. What is the difference between as and is keywords in C#?

- is checks if an object is of a specified type and returns a boolean.
- as casts an object to the specified type or returns null if the cast fails.
- Use is for type checking and as for safe casting.
- *Example*:
- object obj = "Hello";
- if (obj is string str) Console.WriteLine(str); *// Safe type pattern matching.*
- string result = obj as string; *// Safe casting.*

89. How can you implement multi-threading in C#?

- Use Thread, Task, or Parallel classes for concurrent execution.
- Ensure proper synchronization to avoid race conditions.
- Use async/await for asynchronous programming.
- *Example*:
- Task.Run(() => Console.WriteLine("Running in a separate thread."));

90. What is the role of a thread pool in C#?

- Manages a pool of worker threads for efficient task execution.
- Reduces overhead by reusing threads instead of creating new ones.
- Optimized for short-lived and repetitive tasks.
- *Example*:
- ThreadPool.QueueUserWorkItem(_ => Console.WriteLine("Thread pool example."));

91. How do you handle memory leaks in C#?

- Avoid unmanaged resources or release them promptly using the Dispose method.
- Use the using statement to ensure proper disposal of resources.
- Regularly profile and monitor memory usage with tools like dotMemory.
- *Example*:
- using (var resource = new StreamReader("file.txt"))
- {
- Console.WriteLine(resource.ReadToEnd());
- } *// Automatically disposes the StreamReader.*

92. What is the role of garbage collection in C#?

- Automatically manages memory allocation and deallocation.
- Identifies and frees unused objects to prevent memory leaks.
- Operates in generations (0, 1, 2) for efficiency.
- *Example*:
- GC.Collect(); *// Explicitly triggers garbage collection (rarely needed).*

93. Can you explain the significance of the async keyword in C#?

- Marks methods as asynchronous to enable non-blocking execution.
- Used with await to handle asynchronous tasks.
- Improves application responsiveness, especially in I/O-bound operations.
- *Example*:
- async Task<string> FetchDataAsync()
- {
- await Task.Delay(1000);
- return "Data fetched";
- }

94. What is the difference between ref and out parameters in C#?

- ref requires the variable to be initialized before passing.
- out allows the variable to be uninitialized but must be assigned in the method.
- Both pass arguments by reference, allowing modifications.
- *Example*:
- void SetValues(ref int a, out int b)
- {
- a *= 2;
- b = 10;
- }
- int x = 5, y;
- SetValues(ref x, out y);

95. What are the advantages of using var in C#?

- Simplifies code by inferring the type at compile time.
- Reduces redundancy in declarations, improving readability.
- Useful for LINQ queries or anonymous types.
- *Example*:
- var numbers = new List<int> { 1, 2, 3 };

96. How does the yield keyword work in C#?

- Produces a sequence of values in an iterator without creating an entire collection.
- Suspends execution and resumes from the last yield statement.
- Efficient for large or infinite data streams.
- *Example*:
- IEnumerable<int> GenerateNumbers()
- {
-     for (int i = 0; i < 5; i++)
-         yield return i;
- }

97. What is the difference between a delegate and an event in C#?

- A delegate is a type that holds a reference to methods.
- An event is a wrapper over a delegate that restricts direct invocation.
- event provides better encapsulation and is used for publish/subscribe patterns.
- *Example*:
- public delegate void Notify();
- public event Notify OnNotify;

98. How do you implement the Singleton pattern in C#?

- Ensures a class has only one instance and provides a global access point.
- Use a private constructor and a static instance.
- Thread-safe implementation involves locking or Lazy<T>.
- *Example*:
- public sealed class Singleton
- {
-     private static readonly Lazy<Singleton> instance = new(() => new Singleton());
-     private Singleton() { }
-     public static Singleton Instance => instance.Value;
- }

99. What is a static class in C# and when should it be used?

- A static class cannot be instantiated and only contains static members.
- Ideal for utility functions, constants, or extension methods.
- Enhances performance by eliminating the need for object creation.
- *Example*:
- public static class MathUtils
- {
-     public static int Add(int a, int b) => a + b;
- }

100. What is the role of an Interface in C#?

- Defines a contract that implementing classes must follow.
- Supports multiple inheritance by implementing multiple interfaces.
- Facilitates loose coupling and testability.
- *Example*:
- public interface IVehicle
- {
-     void Start();
- }
- public class Car : IVehicle
- {
-     public void Start() => Console.WriteLine("Car started.");
- }

101. How do you define a constant in C#?

- Use the const keyword for compile-time constants.
- Use readonly for runtime constants.
- Constants improve code readability and prevent accidental modification.
- *Example*:
- const double Pi = 3.14159;

102. How do you use a lambda expression with LINQ in C#?

- Lambda expressions define inline functions for LINQ operations.
- Commonly used with methods like Where, Select, and OrderBy.
- Simplifies filtering and transforming data collections.
- *Example*:
- var evenNumbers = numbers.Where(n => n % 2 == 0);

103. What is a "null" reference exception in C#?

- Occurs when attempting to access members of a null object.
- Prevented using null checks or the null conditional operator (?.).
- Avoided with nullable reference types (?) and null coalescing.
- *Example*:
- string? name = null;
- Console.WriteLine(name?.Length);

104. How can you prevent a class from being instantiated in C#?

- Mark the class as static or use a private constructor.
- Prevents unintended usage while exposing functionality.
- Common for utility or helper classes.
- *Example*:
- public static class Utilities
- {
-     public static void DoWork() => Console.WriteLine("Working!");
- }

105. What is a thread-safe collection in C#?

- A collection designed to handle concurrent access without data corruption.
- Examples include ConcurrentDictionary and BlockingCollection.
- Ideal for multi-threaded applications.
- *Example*:
- var dict = new ConcurrentDictionary<int, string>();
- dict.TryAdd(1, "Value1");

There are **25 questions remaining** (106 to 130). Here's the completion:

---

106. What is the difference between try-catch and try-finally in C#?

- try-catch is used for handling exceptions, where the catch block processes errors.
- try-finally ensures cleanup or final steps regardless of exceptions.
- Use try-catch for error handling and try-finally for cleanup.
- *Example*:
- try
- {
-     int result = 10 / 0;
- }
- finally
- {
-     Console.WriteLine("Cleanup executed.");
- }

107. How do you implement dependency injection in C#?

- Inject dependencies into a class via constructor, property, or method.

- Promotes loose coupling and testability.
- Use frameworks like ASP.NET Core's built-in DI container.
- *Example*:
- public class Service { }
- public class Consumer
- {
-    private readonly Service _service;
-    public Consumer(Service service) => _service = service;
- }

108. What is a collection initializer in C#?

- Allows initializing collections with values at the time of declaration.
- Reduces boilerplate code.
- Works with any collection that implements ICollection<T>.
- *Example*:
- var numbers = new List<int> { 1, 2, 3, 4 };

109. How do you implement deep cloning in C#?

- Use serialization or manual member-wise copy for deep cloning.
- Ensures all nested objects are cloned.
- Use libraries like Newtonsoft.Json for simpler implementation.
- *Example*:
- var deepCopy = JsonConvert.DeserializeObject<MyClass>(JsonConvert.SerializeObject(original));

110. What are the different types of collections in C#?

- Non-generic: ArrayList, Hashtable, etc.
- Generic: List<T>, Dictionary<TKey, TValue>, etc.
- Concurrent: ConcurrentBag<T>, ConcurrentDictionary<TKey, TValue>.
- *Example*:
- var dict = new Dictionary<int, string> { { 1, "One" } };

111. What is the purpose of a constructor in C#?

- Initializes objects of a class.
- Can be parameterized or parameterless.
- Automatically invoked when an object is created.
- *Example*:
- public class Car
- {
-    public Car(string model) { Model = model; }
-    public string Model { get; }
- }

112. How do you make a class thread-safe in C#?

- Use locks, Monitor, or thread-safe collections.
- Minimize shared resources and critical sections.
- Use immutability for objects when possible.
- *Example*:
- private static readonly object lockObj = new();
- lock (lockObj) { */* Critical section */* }

113. What is an iterator in C#?

- Used to traverse a collection using yield statements.
- Implements IEnumerable or IEnumerator.
- Simplifies creating custom collections.
- *Example*:
- public IEnumerable<int> GetNumbers()
- {
-    for (int i = 0; i < 5; i++) yield return i;
- }

114. How does a using statement work in C#?

- Ensures resources are disposed of automatically.
- Commonly used with objects implementing IDisposable.
- Shortens and simplifies resource management.
- *Example*:
- using (var resource = new StreamReader("file.txt"))
- {
-    Console.WriteLine(resource.ReadToEnd());
- }

115. What is the purpose of the params keyword in C#?

- Allows methods to accept a variable number of arguments.
- Useful for simplifying parameter passing.
- Accepts zero or more arguments as an array.
- *Example*:
- void Print(params int[] numbers)
- {
-    foreach (var num in numbers) Console.WriteLine(num);
- }

116. How does C# handle exception filtering?

- Enables conditional filtering using the when keyword.
- Improves readability and separates exception handling logic.
- Reduces nested if conditions in catch blocks.
- *Example*:
- catch (Exception ex) when (ex.Message.Contains("Specific Error"))
- {
-    Console.WriteLine("Filtered exception.");
- }

117. How do you handle exceptions in asynchronous methods in C#?

- Use await to capture exceptions in try-catch.
- Handle exceptions in the Task returned by async methods.
- Optionally use Task.Exception for unobserved exceptions.
- *Example*:
- try
- {
-    await SomeAsyncMethod();
- }
- catch (Exception ex)
- {
-    Console.WriteLine(ex.Message);
- }

118. What are anonymous methods in C#?

- Methods without a name, defined inline using the delegate keyword.
- Useful for short, simple operations.
- Replaced by lambda expressions in most cases.
- *Example*:
- Action<int> print = delegate (int x) { Console.WriteLine(x); };
- print(10);

119. What is the difference between IEnumerable and IEnumerator in C#?

- IEnumerable provides an iterator for a collection.
- IEnumerator allows iteration with MoveNext() and Current.
- IEnumerable is used for collection exposure; IEnumerator for iteration logic.

- *Example*:
- foreach (var item in collection) { */* Uses IEnumerable */* }

120. What are the differences between a class and a struct in C#?

- Classes are reference types, while structs are value types.
- Classes support inheritance; structs do not.
- Structs are lightweight and ideal for small data types.
- *Example*:
- struct Point { public int X, Y; }

121. How do you implement error handling in asynchronous methods in C#?

- Wrap await calls in try-catch.
- Use Task.ContinueWith for additional handling.
- Ensure tasks are awaited to capture exceptions.
- *Example*:
- try
- {
-    await ProcessAsync();
- }
- catch (Exception ex)
- {
-    Console.WriteLine(ex.Message);
- }

122. What is the difference between Array and List in C#?

- Array is fixed-size, while List is dynamic.
- Array provides better performance for fixed-size collections.
- List supports many helper methods like Add and Remove.
- *Example*:
- List<int> numbers = new() { 1, 2, 3 };

123. What is the difference between a shallow copy and a deep copy of an object in C#?

- A shallow copy duplicates only the top-level structure.
- A deep copy duplicates all referenced objects recursively.
- Use serialization or cloning libraries for deep copies.
- *Example*:
- var deepCopy = JsonConvert.DeserializeObject<MyClass>(JsonConvert.SerializeObject(original));

124. What is the purpose of the params keyword in a method signature?

- Simplifies passing a variable number of parameters to a method.
- Eliminates the need for creating arrays explicitly.
- *Example*:
- void Print(params string[] names) { foreach (var name in names) Console.WriteLine(name); }

125. What is the difference between string and String in C#?

- Both refer to System.String; string is an alias in C#.
- String is used for accessing methods and properties explicitly.
- Functionally identical but stylistically different.
- *Example*:
- string name = "John";
- String upperName = name.ToUpper();

126. How do you use a constructor with parameters in C#?

- Define parameters in the constructor to initialize fields.
- Invoke using the new keyword with arguments.
- *Example*:
- public Person(string name, int age) { Name = name; Age = age; }

127. What is the importance of finally in exception handling in C#?

- Ensures code runs regardless of exceptions.
- Ideal for releasing resources or cleanup.
- *Example*:
- try { */* Work */* } finally
-

{ Console.WriteLine("Cleanup"); }

128. How do you create a read-only property in C#?
- Define a property with only a `get` accessor.
- Use `readonly` keyword for backing fields.
- *Example*:
```csharp
public int Age { get; } = 30;

**129. What is the difference between a static method and an instance method in C#?**

- Static methods belong to the class; instance methods to objects.
- Static methods do not require object instantiation.
- *Example*:
- public static int Add(int x, int y) => x + y;

**130. How do you implement a generic method in C#?**

- Define type parameters in the method signature.
- Enables type-safe code reuse.
- *Example*:
- public T GetMax<T>(T a, T b) where T : IComparable<T>
- {
-    return a.CompareTo(b) > 0 ? a : b;
- }

---
```

1. What are the main categories of design patterns?

- Creational patterns deal with object creation mechanisms, enhancing flexibility and reuse.
- Structural patterns focus on object composition, simplifying relationships between entities.
- Behavioral patterns address object collaboration and responsibility delegation.
- Examples include Singleton (Creational), Adapter (Structural), and Observer (Behavioral).
- I used Behavioral patterns in an event-driven project to manage communication between objects using the Observer pattern. Example:

```
public class ObserverExample
{
   public interface IObserver { void Update(string message); }
   public class Subject
   {
      private List<IObserver> observers = new List<IObserver>();
      public void AddObserver(IObserver observer) =>
observers.Add(observer);
      public void NotifyObservers(string message)
      {
         observers.ForEach(o => o.Update(message));
      }
   }
}
```

2. What is a design pattern and why should anyone use them?

- A design pattern is a proven, reusable solution to a common software design problem.
- They improve communication by providing shared terminology.
- Using them promotes code reusability, scalability, and maintainability.
- In my .NET Core API project, Singleton ensured a single instance of a logger, optimizing resource usage. Example:

```
public sealed class Logger
{
   private static readonly Logger instance = new Logger();
   private Logger() {}
   public static Logger Instance => instance;
   public void Log(string message) { Console.WriteLine(message); }
}
```

3. What is a pattern in software design?

- A pattern is a general reusable solution to a recurring problem.
- It's not a finished design but a template for solving problems.
- Patterns guide developers to write optimized and clean code.
- In an Angular project, I used patterns to structure reusable services and components effectively.

4. What is the Singleton design pattern?

- Ensures only one instance of a class exists throughout the application.
- Provides a global point of access to that instance.
- It is useful for managing shared resources like configuration or logging.
- Implemented Singleton for database connections in my project, reducing overhead. Example:

```
public sealed class DatabaseConnection
{
   private static readonly DatabaseConnection instance = new
DatabaseConnection();
   private DatabaseConnection() { }
   public static DatabaseConnection Instance => instance;
}
```

5. What is Dependency Injection?

- A technique to achieve Inversion of Control by providing dependencies externally.
- It decouples the creation of an object from its usage.
- Promotes testability and easier maintenance.
- Used Dependency Injection in .NET Core using built-in support to inject services into controllers. Example:

```
public class MyService : IMyService
{
   public void Execute() { /* Implementation */ }
}
```

6. What is the State design pattern?

- Allows an object to alter its behavior when its internal state changes.
- Encapsulates state-specific behavior in separate classes.
- Improves code organization and makes it easy to add new states.
- Implemented this pattern for managing payment statuses in an e-commerce application. Example:

```
public interface IState { void Handle(); }
public class ApprovedState : IState { public void Handle() {
Console.WriteLine("Payment approved."); } }
public class RejectedState : IState { public void Handle() {
Console.WriteLine("Payment rejected."); } }
```

7. What is the Null Object pattern?

- Provides a default implementation for a class to avoid null checks.
- Improves code readability and reduces null reference errors.
- Useful for cases where the absence of an object should be handled gracefully.
- Used this pattern to avoid null checks in optional service handling. Example:

```
public interface IService { void Execute(); }
public class NullService : IService { public void Execute() { } }
```

8. What is the Template Method pattern?

- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- Promotes code reuse and flexibility.
- Useful in situations with a common sequence of steps but variation in certain steps.
- Applied in a report generation system where formatting differed by report type. Example:

```
public abstract class ReportTemplate
{
   public void GenerateReport() { FetchData(); FormatReport();
SaveReport(); }
   protected abstract void FetchData();
   protected abstract void FormatReport();
   protected void SaveReport() { Console.WriteLine("Saving Report..."); }
}
```

9. What is the Iterator pattern?

- Provides a way to access elements of a collection sequentially without exposing its structure.
- Useful for traversing complex data structures.
- Promotes encapsulation and simplifies traversal.
- Used this pattern for custom collections in a financial app. Example:

```
public interface IIterator { bool HasNext(); object Next(); }
public class ArrayIterator : IIterator
{
    private int[] _array;
    private int position = 0;
    public ArrayIterator(int[] array) { _array = array; }
    public bool HasNext() => position < _array.Length;
    public object Next() => _array[position++];
}
```

10. What is the Strategy pattern?

- Defines a family of algorithms and makes them interchangeable.
- Promotes flexibility and reduces conditional logic.
- Ideal for implementing various behaviors dynamically.
- Used for implementing sorting algorithms in a library system. Example:

```
public interface ISortingStrategy { void Sort(int[] data); }
public class QuickSort : ISortingStrategy { public void Sort(int[] data) { /*
QuickSort implementation */ } }
```

11. What is the Proxy pattern?

- Provides a surrogate or placeholder for another object to control access.
- Useful for lazy initialization, logging, or security purposes.
- Reduces resource usage by controlling object creation.
- Used a Proxy for caching API responses in a project. Example:

```
public class ServiceProxy : IService
{
    private RealService _realService;
    public void Execute()
    {
        if (_realService == null) _realService = new RealService();
        _realService.Execute();
    }
}
```

12. What are some benefits of the Repository pattern?

- Centralizes data access logic, improving code maintainability.
- Abstracts database operations, making code database-agnostic.
- Simplifies testing by enabling mock repositories.
- Used in a Unit of Work setup for efficient transaction management. Example:

```
public interface IRepository<T> { T GetById(int id); void Add(T entity); }
public class Repository<T> : IRepository<T> { /* Implementation */ }
```

13. What is the Filter pattern?

- Filters a set of objects using different criteria and chaining them together.
- Promotes flexibility and reusability.
- Useful in applications requiring dynamic filtering.
- Implemented in a product filtering module of an e-commerce site. Example:

```
public interface IFilter<T> { IEnumerable<T> Filter(IEnumerable<T> items);
}
```

14. What is the Builder pattern?

- Separates object construction from its representation.
- Ideal for creating complex objects with multiple configurations.
- Simplifies object creation and promotes immutability.
- Used for building configuration files dynamically. Example:

```
public class ProductBuilder { public ProductBuilder SetName(string name)
{ /* Implementation */ return this; } }
```

15. What are the types of design patterns?

- Creational patterns focus on object creation.
- Structural patterns emphasize object relationships and compositions.
- Behavioral patterns govern object collaboration and interactions.
- Patterns like Singleton, Adapter, and Strategy cover these categories comprehensively.
- Implemented Strategy for configurable export formats in a document processing system. Example:

```
public interface IExportStrategy { void Export(string data); }
public class PDFExport : IExportStrategy { public void Export(string data) {
/* Export to PDF */ } }
```

16. What is Inversion of Control?

- A design principle where the control of object creation and flow is transferred from the program to a framework or container.
- Helps decouple the code by abstracting dependencies.
- Implemented using Dependency Injection, Service Locators, or Events.
- In .NET Core, IoC is achieved through the built-in Dependency Injection container. Example:

```
services.AddTransient<IMyService, MyService>();
```

17. Why would you want to use a Repository pattern with an ORM?

- Abstracts database queries, allowing switching ORMs without changing business logic.
- Simplifies testing by mocking repositories instead of actual database interactions.
- Centralizes and organizes data access logic.
- Used with Entity Framework Core to simplify queries and CRUD operations. Example:

```
public class ProductRepository : IRepository<Product>
{
    private readonly DbContext _context;
    public ProductRepository(DbContext context) { _context = context; }
    public Product GetById(int id) => _context.Products.Find(id);
}
```

18. Can we create a clone of a Singleton object?

- Technically, it is possible but violates the Singleton principle.
- Using reflection or serialization can bypass Singleton constraints.
- This is generally discouraged as it defeats the purpose of Singleton.
- Prevented cloning in a project by implementing ICloneable and throwing an exception. Example:

```
public class Singleton : ICloneable
{
    private static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
    public object Clone() => throw new NotSupportedException("Cloning
not allowed.");
}
```

19. What is the Factory pattern?

- A creational pattern that provides an interface for creating objects without specifying their exact class.
- Promotes loose coupling and enhances code flexibility.
- Used for creating instances based on runtime conditions.
- Implemented for a notification system to generate email or SMS services dynamically. Example:

```
public interface INotification { void Send(string message); }
public class NotificationFactory
{
    public INotification CreateNotification(string type) => type switch
    {
        "Email" => new EmailNotification(),
        "SMS" => new SmsNotification(),
        _ => throw new ArgumentException("Invalid type")
```

```
  };
}
```

20. What is Unit of Work?

- A design pattern that groups related operations into a single transaction.
- Ensures that all operations either succeed or fail together.
- Works with Repository to manage object changes efficiently.
- Used in my project to wrap multiple database operations in a transaction. Example:

```
public interface IUnitOfWork { void Commit(); }
public class UnitOfWork : IUnitOfWork
{
  private readonly DbContext _context;
  public UnitOfWork(DbContext context) { _context = context; }
  public void Commit() { _context.SaveChanges(); }
}
```

21. What is the Claim Check pattern in Azure Event Grid?

- A messaging pattern to offload payloads from messages and replace them with references (claims).
- Reduces message size, improving performance and scalability.
- Claims are used to retrieve the full payload from storage when needed.
- Used this to handle large messages efficiently in a distributed system. Example:

```
{
  "eventType": "PayloadReference",
  "data": {
    "url":
"https://storage.blob.core.windows.net/container/message.json"
  }
}
```

22. What is the Chain of Responsibility pattern?

- Passes a request along a chain of handlers until one processes it.
- Decouples sender and receiver, promoting flexibility.
- Useful for workflows or dynamic request processing.
- Implemented for dynamic request validation in an API project. Example:

```
public abstract class Handler
{
  protected Handler Next;
  public void SetNext(Handler next) => Next = next;
  public abstract void HandleRequest(int request);
}
```

23. What is the Memento pattern?

- Captures and restores an object’s state without violating encapsulation.
- Useful for undo functionality.
- Separates the concerns of saving state and managing state restoration.
- Used in a text editor application for undo/redo features. Example:

```
public class Memento { public string State { get; } }
public class Originator
{
  private string state;
  public void SetState(string state) { this.state = state; }
  public Memento SaveState() => new Memento(state);
}
```

24. What is the Command pattern?

- Encapsulates a request as an object, enabling parameterization and queuing.
- Useful for undoable operations and logging changes.
- Decouples sender and receiver of requests.
- Used to implement an undo system in a task management app. Example:

```
public interface ICommand { void Execute(); void Undo(); }
public class CreateTaskCommand : ICommand
{
  public void Execute() { Console.WriteLine("Task Created"); }
  public void Undo() { Console.WriteLine("Task Creation Undone"); }
}
```

25. What is Event Sourcing?

- Captures all changes to an application state as a sequence of events.
- Ensures immutability and traceability of data changes.
- Used for audit logs and temporal queries.
- Applied in an inventory system to track stock changes. Example:

```
public class Event { public string EventType; public DateTime Timestamp; }
public class EventStore { private List<Event> events = new(); public void
AddEvent(Event e) => events.Add(e); }
```

26. What are the benefits of CQRS?

- Separates command (write) and query (read) responsibilities for better scalability.
- Optimizes performance by tailoring read and write models.
- Improves maintainability by decoupling concerns.
- Implemented for high-traffic e-commerce sites to scale queries independently.

27. What are the drawbacks of the Active Record pattern?

- Tight coupling between database and domain logic.
- Poor testability due to direct database dependency.
- Difficult to manage complex business logic in Active Records.
- Faced challenges with Active Record in a legacy project, moving to Repository solved them.

28. What is the Command and Query Responsibility Segregation (CQRS) pattern?

- Splits write operations (commands) from read operations (queries).
- Enables scaling reads and writes independently.
- Improves performance and simplifies query optimization.
- Used CQRS in an analytics app to handle large-scale reporting.

29. What are the advantages of using Dependency Injection?

- Promotes loose coupling and improves maintainability.
- Makes unit testing easier by enabling mocking.
- Enhances flexibility and reusability of components.
- Injected services in my .NET API project for streamlined testing and modularity.

30. What are some reasons to use the Repository pattern?

- Centralizes data access logic and promotes code reuse.
- Decouples business logic from database operations.
- Simplifies testing by mocking repositories.
- Used in conjunction with Unit of Work for transaction management.

31. What is an Aggregate Root in the context of the Repository pattern?

- It is the primary entity that controls the lifecycle of a group of related objects.

- Ensures that all changes to related objects go through the root entity.
- Maintains consistency and prevents direct access to related objects.
- Used Aggregate Roots in an e-commerce project for managing orders and their line items. Example:

```
public class Order
{
   public int Id { get; set; }
   public List<OrderItem> Items { get; } = new();
   public void AddItem(OrderItem item) => Items.Add(item);
}
```

32. In OOP, what is the difference between the Repository pattern and a Service Layer?

- The Repository pattern handles data access and persistence logic.
- The Service Layer contains business logic and coordinates operations across repositories.
- Repositories focus on CRUD operations, while services orchestrate workflows.
- Used Service Layer in conjunction with Repository to encapsulate business rules. Example:

```
public class OrderService
{
   private readonly IRepository<Order> _repository;
   public OrderService(IRepository<Order> repository) { _repository =
repository; }
   public void PlaceOrder(Order order) { /* Business logic */
_repository.Add(order); }
}
```

33. Is Unit of Work equal to Transaction or is it more than that?

- Unit of Work coordinates changes across multiple repositories in a single transaction.
- Transactions are low-level, while Unit of Work is higher-level and manages multiple operations.
- Unit of Work also tracks changes to entities to minimize database interactions.
- Used Unit of Work in a multi-repository scenario to commit changes atomically. Example:

```
public class UnitOfWork : IUnitOfWork
{
   private readonly DbContext _context;
   public UnitOfWork(DbContext context) { _context = context; }
   public void Commit() { _context.SaveChanges(); }
}
```

34. When should I use the Active Record vs. Repository pattern?

- Use Active Record for simpler applications where domain logic is minimal.
- Use Repository for complex systems with intricate business rules and large teams.
- Repository promotes separation of concerns, making it more testable.
- Transitioned from Active Record to Repository in a project to better manage complexity.

35. What is the Interpreter pattern?

- Defines a grammar for a language and an interpreter to interpret sentences in the language.
- Useful for parsing and evaluating expressions.
- Promotes extensibility by allowing new grammar rules.
- Used this pattern to evaluate mathematical expressions in a custom scripting language. Example:

```
public interface IExpression { int Interpret(); }
public class Number : IExpression
{
   private readonly int _number;
   public Number(int number) { _number = number; }
   public int Interpret() => _number;
}
```

36. What is the Abstract Factory pattern?

- Provides an interface for creating families of related objects without specifying their concrete classes.
- Useful for creating platform-specific implementations.
- Promotes consistency across products in the same family.
- Used for generating UI components for different operating systems. Example:

```
public interface IUIFactory { IButton CreateButton(); }
public class WindowsUIFactory : IUIFactory { public IButton CreateButton()
=> new WindowsButton(); }
```

37. What is the Adapter pattern?

- Converts the interface of a class into another interface the client expects.
- Promotes compatibility between incompatible interfaces.
- Used for integrating third-party libraries in legacy systems.
- Applied this pattern to use a new logging framework in an existing application. Example:

```
public interface ILogger { void Log(string message); }
public class LogAdapter : ILogger
{
   private readonly ThirdPartyLogger _logger;
   public LogAdapter(ThirdPartyLogger logger) { _logger = logger; }
   public void Log(string message) => _logger.WriteLog(message);
}
```

38. What is the Bridge pattern?

- Decouples abstraction from implementation so that they can vary independently.
- Useful for designing cross-platform systems.
- Promotes flexibility and scalability in design.
- Used for implementing shape rendering in a graphics application. Example:

```
public interface IRenderer { void Render(string shape); }
public class Circle
{
   private readonly IRenderer _renderer;
   public Circle(IRenderer renderer) { _renderer = renderer; }
   public void Draw() => _renderer.Render("Circle");
}
```

39. What does "program to interfaces, not implementations" mean?

- Encourages using abstractions (interfaces) instead of concrete classes.
- Promotes flexibility, making code more extensible and testable.
- Reduces coupling between modules and facilitates dependency injection.
- Followed this principle to inject dependencies in a modular application. Example:

```
public interface INotification { void Send(string message); }
```

40. What is the Decorator pattern?

- Dynamically adds new behaviors to objects without altering their structure.
- Promotes code reuse and adherence to the open/closed principle.

- Used this pattern to add logging and validation to services. Example:

```
public interface INotifier { void Notify(string message); }
public class EmailNotifier : INotifier { public void Notify(string message) {
/* Email sending */ } }
public class LoggingNotifier : INotifier
{
   private readonly INotifier _notifier;
   public LoggingNotifier(INotifier notifier) { _notifier = notifier; }
   public void Notify(string message)
   {
      Console.WriteLine("Logging: " + message);
      _notifier.Notify(message);
   }
}
```

---

41. What is the Prototype pattern?

- Creates new objects by copying an existing object.
- Useful for creating objects with similar configurations.
- Promotes efficiency when object creation is expensive.
- Used this pattern for cloning configurations in a template-based system. Example:

```
public class Prototype : ICloneable
{
   public string Name { get; set; }
   public object Clone() => MemberwiseClone();
}
```

---

42. What is the Facade pattern?

- Provides a simplified interface to a complex subsystem.
- Reduces the dependency of client code on subsystem classes.
- Used this pattern to simplify access to third-party APIs in a project. Example:

```
public class EmailService
{
   public void SendEmail(string to, string subject, string body) { /*
Implementation */ }
}
public class NotificationFacade
{
   private readonly EmailService _emailService = new();
   public void Notify(string message) =>
_emailService.SendEmail("user@example.com", "Notification", message);
}
```

---

43. What is the difference between Proxy and Decorator patterns?

- Proxy controls access to the object, while Decorator adds new behavior.
- Proxy focuses on resource management; Decorator enhances functionality.
- Both follow similar structures but serve different purposes.
- Used Proxy for caching and Decorator for dynamic feature extension in different projects.

---

44. What are the differences between a Static class and a Singleton class?

- A Static class cannot be instantiated; a Singleton ensures a single instance.
- Singleton allows controlled initialization; Static does not manage state.
- Singleton enables lazy loading and dependency injection.
- Used Singleton for shared configurations and Static for utility methods.

---

45. When should I use the Composite design pattern?

- Use when you need to treat individual objects and compositions uniformly.
- Ideal for hierarchical structures like file systems or menus.
- Simplifies client code by providing a unified interface.
- Implemented for rendering nested UI elements in a web application. Example:

```
public interface IComponent { void Render(); }
public class Composite : IComponent
{
   private List<IComponent> _children = new();
   public void Add(IComponent component) =>
_children.Add(component);
   public void Render() { foreach (var child in _children) child.Render(); }
}
```

46. What is the Observer pattern?

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.
- Useful for implementing publish-subscribe mechanisms.
- Promotes loose coupling between subjects and observers.
- Used this pattern to notify multiple components of state changes in an Angular project. Example:

```
public class Subject
{
   private readonly List<IObserver> observers = new();
   public void Attach(IObserver observer) => observers.Add(observer);
   public void Notify() { foreach (var observer in observers)
observer.Update(); }
}
public interface IObserver { void Update(); }
```

---

47. What is the Mediator pattern?

- Encapsulates interactions between multiple objects, promoting loose coupling.
- Centralizes communication to simplify object dependencies.
- Useful for managing complex workflows with multiple interacting components.
- Used this pattern in a chat application to coordinate user interactions. Example:

```
public class ChatRoom
{
   public void ShowMessage(string user, string message) =>
Console.WriteLine($"{user}: {message}");
}
```

---

48. How is the Bridge pattern different from the Adapter pattern?

- Bridge separates abstraction and implementation, allowing them to vary independently.
- Adapter makes an existing class compatible with a new interface.
- Bridge is used proactively for extensibility, while Adapter solves compatibility issues retroactively.
- Applied Bridge for modular development and Adapter for legacy integration.

---

49. What is the Service Locator pattern?

- Provides a central registry for locating services in an application.
- Decouples client code from dependency initialization.
- Can make code harder to trace compared to Dependency Injection.
- Used this pattern in legacy codebases where DI was not feasible. Example:

```
public class ServiceLocator
{
   private static readonly Dictionary<Type, object> Services = new();
```

```
    public static void Register<T>(T service) => Services[typeof(T)] = service;
    public static T Resolve<T>() => (T)Services[typeof(T)];
}
```

50. What is the Flyweight pattern?

- Reduces memory usage by sharing common parts of objects that are expensive to instantiate.
- Useful when dealing with large numbers of objects with shared state.
- Separates intrinsic (shared) and extrinsic (unique) data.
- Used this pattern for rendering thousands of text characters in a graphics application. Example:

```
public class Flyweight
{
    private readonly string _intrinsicState;
    public Flyweight(string intrinsicState) { _intrinsicState = intrinsicState; }
    public void Operation(string extrinsicState) =>
Console.WriteLine($"{_intrinsicState} - {extrinsicState}");
}
```

51. What is the difference between Strategy and State design patterns?

- Strategy allows choosing an algorithm at runtime, while State changes behavior based on object state.
- Strategy focuses on behavior delegation; State focuses on transitions between states.
- Both promote flexibility but serve different purposes.
- Used Strategy for dynamic sorting and State for a finite state machine in different projects.

52. What are some disadvantages of Dependency Injection?

- Can make code harder to read and debug due to indirection.
- Requires additional setup and configuration, increasing complexity.
- Overuse can lead to an explosion of interfaces and abstractions.
- Faced challenges with overly complex DI in a large-scale .NET application.

53. What is the relationship between Repository and Unit of Work?

- Repository manages CRUD operations for entities.
- Unit of Work coordinates transactions across multiple repositories.
- Together, they ensure consistency and reduce database interaction.
- Implemented this combination to handle operations on multiple aggregates atomically.

54. Why would I use the Chain of Responsibility over a Decorator?

- Chain of Responsibility processes requests in a sequence, allowing multiple handlers.
- Decorator adds behavior dynamically to a single object.
- Use Chain for request processing workflows; use Decorator for extensible functionality.
- Applied Chain in a middleware pipeline and Decorator for feature toggles in projects.

55. Why shouldn't I use the Repository pattern with Entity Framework?

- EF already provides a DbContext, which acts as a repository.
- Adding another Repository layer can lead to redundant abstractions.
- Complex repositories may hide EF features like LINQ and tracking.
- Used EF directly in simple projects but added Repository for complex systems with business logic.

56. When would you use the Builder pattern? Why not just use the Factory pattern?

- Use Builder for constructing complex objects step-by-step.
- Factory is suitable for creating objects without complex configuration.
- Builder separates the construction process from the representation.
- Used Builder for configuring multi-step object creation in a reporting tool. Example:

```
public class ReportBuilder
{
    private Report _report = new();
    public ReportBuilder AddTitle(string title) { _report.Title = title; return
this; }
    public Report Build() => _report;
}
```

57. What is the difference between Composition and Inheritance?

- Composition involves "has-a" relationships, while Inheritance represents "is-a" relationships.
- Composition is more flexible and avoids tight coupling.
- Inheritance can lead to brittle hierarchies if overused.
- Used Composition for dynamic behavior and Inheritance for shared attributes in a UI library.

58. How should I group my Repositories when using the Repository pattern?

- Group repositories by aggregate roots or domain areas for better modularity.
- Avoid creating a repository for every entity; focus on aggregates.
- Centralize common logic in a base repository when possible.
- Organized repositories by domain in an e-commerce system: OrderRepository, ProductRepository, etc.

59. Is the Repository pattern the same as the Active Record pattern?

- Repository separates domain logic from database access, while Active Record combines them.
- Repository promotes testability and decoupling.
- Active Record is simpler for smaller projects but harder to scale for complex systems.
- Migrated from Active Record to Repository in a project to simplify testing.

60. What would you choose: a Repository pattern or "smart" business objects?

- Repository pattern for clear separation of data access and domain logic.
- Smart business objects for simpler, self-contained applications.
- Repository scales better for large systems, while smart objects are quick for prototypes.
- Chose Repository for an enterprise app requiring complex query handling.

61. Could you explain some benefits of the Repository pattern?

- Simplifies data access logic by providing a clean abstraction layer.
- Promotes testability by allowing mock implementations.
- Centralizes database access, reducing duplication across codebases.
- Implemented this pattern to streamline data access in a multi-layered web application.

62. Could you explain the difference between Facade and Mediator patterns?

- Facade simplifies access to a complex subsystem by providing a unified interface.
- Mediator manages communication between multiple objects without them knowing about each other.
- Facade is about reducing system complexity; Mediator focuses on object interaction.
- Used Facade for API calls and Mediator for event-driven workflows in projects.

63. What is the difference between Template and Strategy patterns?

- Template defines the skeleton of an algorithm, deferring steps to subclasses.
- Strategy defines a family of algorithms and allows them to be interchangeable.
- Template enforces structure; Strategy promotes flexibility.
- Used Template for defining report generation steps and Strategy for dynamic payment methods.

64. What is the Deadly Diamond of Death in design patterns?

- Refers to ambiguity in multiple inheritance when a class inherits from two classes with a common ancestor.
- Common in languages that support multiple inheritance (e.g., C++).
- Avoided in languages like C# or Java by using interfaces or composition.
- Resolved this issue in a C++ project by ensuring proper virtual inheritance.

65. Could you explain the differences between Facade, Proxy, Adapter, and Decorator design patterns?

- **Facade**: Simplifies access to a subsystem by providing a single interface.
- **Proxy**: Controls access to an object (e.g., caching, logging).
- **Adapter**: Converts an interface into another compatible interface.
- **Decorator**: Dynamically adds behavior to an object without altering its structure.
- Applied these patterns in different scenarios: Facade for API, Proxy for caching, Adapter for legacy systems, and Decorator for runtime extensions.

66. Can we use CQRS without Event Sourcing?

- Yes, CQRS can be implemented without Event Sourcing.
- Event Sourcing is optional and records changes as events; CQRS separates reads and writes.
- Often used together but not inherently dependent on each other.
- Implemented CQRS with traditional database reads in a reporting tool without Event Sourcing.

67. What's the difference between Dependency Injection and Service Locator patterns?

- Dependency Injection provides dependencies explicitly, while Service Locator allows components to fetch them.
- DI promotes explicit dependencies and is more transparent.
- Service Locator can lead to hidden dependencies and harder-to-trace code.
- Switched from Service Locator to DI in a .NET Core project for better testability.

68. How should I add an object into a collection maintained by an Aggregate Root?

- Use a method on the Aggregate Root to encapsulate the operation.
- This ensures business rules are enforced during object addition.
- Direct manipulation of collections violates the Aggregate's consistency boundary.
- Example from an e-commerce project:

```
public class Order
{
   private readonly List<OrderItem> _items = new();
   public void AddItem(OrderItem item) => _items.Add(item); // Enforce rules here
}
```

69. What is the Data Mapper pattern?

- Maps data between in-memory objects and a database while keeping them independent.
- Promotes decoupling between domain logic and database schemas.
- Used by tools like Entity Framework and Hibernate.
- Example:

```
public class DataMapper
{
   public Order MapToOrder(DataRow row) => new Order { Id = (int)row["Id"], Name = (string)row["Name"] };
}
```

70. What is the Repository and Specification pattern combination?

- Specification defines criteria for querying data, and Repository implements data access.
- This combination simplifies complex queries and keeps them reusable.
- Used to filter orders dynamically in a .NET application. Example:

```
public class OrderSpecification
{
   public Expression<Func<Order, bool>> Criteria { get; } = order => order.IsPaid;
}
```

71. What is a Domain Event in DDD?

- Represents something that happened in the domain that is significant to business.
- Often used to trigger side effects or notify other parts of the system.
- Encapsulates changes to the state in an immutable event object.
- Example:

```
public class OrderPlacedEvent
{
   public int OrderId { get; }
   public OrderPlacedEvent(int orderId) { OrderId = orderId; }
}
```

72. What is the Repository pattern in CQRS?

- Provides separate repositories for queries and commands.
- Query repositories focus on read operations, and command repositories focus on writes.
- Promotes clarity and scalability in CQRS implementations.
- Example structure: OrderQueryRepository, OrderCommandRepository.

73. What is an Anti-Corruption Layer (ACL)?

- Translates between different models or systems to maintain domain integrity.
- Prevents external systems from polluting the internal domain model.
- Used to integrate a legacy billing system with a modern payment service.

- Example:

```
public class BillingAcl
{
   public ModernBillingRequest
ConvertToModernRequest(LegacyBillingData legacyData) { /* Mapping
logic */ }
}
```

---

74. What is the purpose of a Value Object in DDD?

- Represents an immutable type defined by its attributes, not identity.
- Promotes consistency and encapsulates logic related to attributes.
- Used for reusable concepts like money, address, or date range.
- Example:

```
public class Money
{
   public decimal Amount { get; }
   public string Currency { get; }
   public Money(decimal amount, string currency) { Amount = amount;
Currency = currency; }
}
```

---

75. What is a bounded context in Domain-Driven Design?

- Represents a logical boundary within which a domain model is consistent.
- Defines clear separations of concerns between subdomains.
- Reduces complexity by isolating models and logic.
- Used bounded contexts to structure a microservices-based application.

1. What is .NET Standard?

- .NET Standard is a specification that defines a set of APIs that all .NET implementations must provide.
- It allows developers to share code across different .NET platforms like .NET Framework, .NET Core, and Xamarin.
- It simplifies library development by eliminating compatibility issues across platforms.
- I used .NET Standard in my project to create a shared library for a cross-platform application using .NET Core and Xamarin.

```
// Shared library in .NET Standard
public class Utility
{
   public string GetMessage() => "Hello from .NET Standard!";
}
```

---

2. What is the .NET Framework?

- .NET Framework is a Windows-only framework for building and running desktop, web, and enterprise applications.
- It includes libraries like ASP.NET for web apps and WPF/WinForms for desktop applications.
- It supports features like garbage collection, type safety, and interoperability with COM components.
- I worked on an enterprise application using .NET Framework to manage employee data with WPF for the UI.

```
// Example of a WPF Application
<Button Content="Click Me" Click="Button_Click"/>
```

---

3. What is .NET Core?

- .NET Core is a cross-platform, open-source framework for building modern applications.
- It supports multiple OSs, including Windows, Linux, and macOS, and is optimized for cloud and microservices.
- .NET Core provides improved performance and a modular approach compared to the .NET Framework.
- I used .NET Core in a project to build a scalable REST API for managing e-commerce transactions.

```
// Minimal API in .NET Core
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello, .NET Core!");
app.Run();
```

---

4. What is the difference between string and String in C#?

- Both string and String refer to the same type in C#; they are aliases for System.String.
- string is a C# keyword, while String is a .NET class type defined in the System namespace.
- Using string is more common in C# coding conventions for declarations, while String is used for accessing static methods.
- I used both in a project to manipulate file paths and content efficiently.

```
string lower = "hello";
string upper = String.ToUpper(lower);
Console.WriteLine(upper); // Output: HELLO
```

---

5. What is Generic Hosting in .NET Core?

- Generic Hosting is a framework for building applications with dependency injection, configuration, and logging.
- It provides a unified way to host applications, including console apps, web apps, and background services.
- The HostBuilder API is used to configure and build the host, which acts as the runtime environment.
- I used Generic Hosting to host a background service for processing real-time data in a .NET Core application.

```
var host = Host.CreateDefaultBuilder(args)
   .ConfigureServices(services =>
services.AddHostedService<MyBackgroundService>())
   .Build();

await host.RunAsync();
```

---

6. What do you understand by Value types and Reference types in .NET? Provide some comparison.

- Value types store data directly, while reference types store references to their data in memory.
- Value types are allocated on the stack, whereas reference types are allocated on the heap.
- Modifying a value type does not affect the original instance; modifying a reference type does.
- I applied these concepts to optimize memory usage in a performance-critical project.

```
// Example
int valueType = 10; // Value type
string referenceType = "Hello"; // Reference type
```

---

7. What is IoC (DI) Container?

- Inversion of Control (IoC) Container manages the lifecycle and dependencies of objects automatically.
- It decouples components by using dependency injection (DI) to provide required services.
- Popular IoC Containers in .NET include built-in Microsoft.Extensions.DependencyInjection.
- I implemented an IoC Container in my project to simplify service injection in an ASP.NET Core application.

```
services.AddScoped<IService, ServiceImplementation>();
```

---

8. What is MSIL?

- Microsoft Intermediate Language (MSIL) is the CPU-independent code generated by the .NET compiler.
- It gets converted to native code by the Just-In-Time (JIT) compiler at runtime.
- MSIL ensures code portability across different architectures supported by .NET.
- I analyzed MSIL using tools like ILDASM for debugging low-level issues in a project.

```
// Sample MSIL (simplified)
IL_0001: ldstr "Hello, World!"
IL_0006: call void [mscorlib]System.Console::WriteLine(string)
```

---

9. What is .NET Standard and why do we need to consider it?

- .NET Standard is a unifying library specification for all .NET implementations.
- It enables code sharing across different platforms, improving development consistency.
- It reduces duplication of effort when creating libraries for cross-platform applications.
- I used .NET Standard for creating a reusable library to handle logging in a multi-platform project.

10. Name some CLR services.

- Memory management, including garbage collection.
- Code access security and role-based security.
- Just-In-Time (JIT) compilation and execution.
- I leveraged garbage collection and security services in a secure financial application.

11. What is a .NET application domain?

- Application Domain (AppDomain) isolates applications from one another in the same process.
- It ensures that faults in one application do not affect others.
- AppDomains were commonly used in the .NET Framework but are replaced by AssemblyLoadContext in .NET Core.
- I utilized AppDomains for dynamically loading plugins in a legacy .NET Framework application.

12. What is CTS?

- Common Type System (CTS) defines how types are declared, used, and managed in .NET.
- It ensures interoperability between languages supported by the .NET runtime.
- CTS supports two categories: Value types and Reference types.
- I worked with CTS to ensure type compatibility between C# and VB.NET components in a project.

13. What is CLR?

- Common Language Runtime (CLR) is the execution engine of .NET, managing code execution.
- It handles memory, thread management, and garbage collection.
- CLR provides cross-language integration and exception handling.
- I utilized CLR features like garbage collection for memory-efficient processing in a data-heavy project.

14. What is an unmanaged resource in .NET?

- Unmanaged resources are resources not managed by the CLR, such as file handles or database connections.
- These resources must be explicitly released to prevent memory leaks.
- The IDisposable interface is used to clean up unmanaged resources.
- I implemented Dispose in a class managing database connections to ensure resource cleanup.

```
public void Dispose()
{
   connection.Close();
}
```

15. What is the difference between decimal, float, and double in .NET?

- decimal is used for high-precision calculations like financial applications.
- float is a single-precision floating-point type with 7 significant digits.
- double is a double-precision floating-point type with 15-16 significant digits.
- I used decimal for calculations in a payroll system to maintain accuracy.

```
decimal salary = 12345.67m;
float rate = 0.5f;
double pi = 3.14159;
```

16. What is Boxing and Unboxing?

- Boxing is the process of converting a value type to an object type.
- Unboxing is converting an object type back to a value type.
- Boxing and unboxing are costly operations as they involve heap allocation and type conversion.
- I optimized performance in a project by minimizing boxing operations when working with collections.

```
int number = 42; // Value type
object boxed = number; // Boxing
int unboxed = (int)boxed; // Unboxing
```

17. What are some characteristics of .NET Core?

- .NET Core is cross-platform, running on Windows, Linux, and macOS.
- It is modular, with a lightweight runtime and framework libraries.
- It provides high performance and supports microservices and containers.
- I used .NET Core in a containerized microservices architecture to build scalable APIs.

```
# Dockerfile for .NET Core application
FROM mcr.microsoft.com/dotnet/aspnet:6.0
COPY ./publish /app
WORKDIR /app
ENTRYPOINT ["dotnet", "MyApp.dll"]
```

18. What is the difference between .NET Core and Mono?

- .NET Core is a modern, cross-platform framework for building applications.
- Mono is a .NET implementation focused on mobile (Xamarin) and gaming platforms (Unity).
- Mono provides a broader API surface for older .NET Framework compatibility.
- I used Mono in a Xamarin project to create a mobile app for iOS and Android.

19. What's the difference between SDK and Runtime in .NET Core?

- SDK (Software Development Kit) includes tools, libraries, and the runtime for development.
- Runtime is only for executing .NET applications, without development tools.
- Developers need the SDK, while end-users need only the runtime to run applications.
- I ensured the runtime was installed on production servers for running a .NET Core web app.

20. What officially replaces WCF in .NET Core?

- WCF is replaced by gRPC, which is a modern RPC framework.
- gRPC supports cross-platform, high-performance communication with features like streaming.
- Unlike WCF, gRPC uses Protocol Buffers (protobuf) for serialization.
- I migrated a legacy WCF service to gRPC for improved performance and cross-platform compatibility.

```
// gRPC Service definition
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}
```

21. What are some benefits of using the Options Pattern in ASP.NET Core?

- Centralizes configuration settings in strongly-typed classes.
- Simplifies validation and management of application settings.
- Reduces dependency on direct configuration retrieval in the codebase.
- I used the Options Pattern to manage API keys and other settings in a secure project.

```
services.Configure<MySettings>(Configuration.GetSection("MySettings"));
```

22. How can you create your own scope for a Scoped object in .NET?

- Use IServiceScopeFactory to create a new scope explicitly.
- Scoped objects are resolved within the lifetime of the created scope.
- It ensures proper lifecycle management in background tasks or non-HTTP contexts.
- I used IServiceScopeFactory to handle database operations in a background worker service.

```
using (var scope = scopeFactory.CreateScope())
{
  var dbContext =
scope.ServiceProvider.GetRequiredService<MyDbContext>();
  dbContext.SaveChanges();
}
```

23. Explain the IoC (DI) Container service lifetimes.

- Transient: New instance created each time it is requested.
- Scoped: Instance is created per scope (e.g., per HTTP request in web apps).
- Singleton: Single instance throughout the application lifecycle.
- I used Scoped services in an API project to manage database contexts per request.

```
services.AddScoped<IMyService, MyService>();
```

24. What is the correct pattern to implement long-running background work in ASP.NET Core?

- Use IHostedService or its derived class BackgroundService.
- Ensure proper lifecycle management with StartAsync and StopAsync methods.
- Use dependency injection to access required services in the background worker.
- I implemented a BackgroundService to process a queue of tasks in a real-time system.

```
public class MyBackgroundService : BackgroundService
{
  protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
  {
    while (!stoppingToken.IsCancellationRequested)
    {
      // Background task logic
      await Task.Delay(1000);
    }
  }
}
```

25. What about MVC in .NET Core?

- MVC (Model-View-Controller) is a design pattern for building web applications.
- ASP.NET Core MVC supports routing, dependency injection, and middleware integration.
- Razor views are used for creating dynamic HTML templates.
- I developed an e-commerce website using ASP.NET Core MVC for scalable and maintainable code.

26. Explain the use of the BackgroundService class in ASP.NET Core.

- BackgroundService is an abstract base class for implementing long-running background tasks.
- It simplifies lifecycle management with built-in methods like ExecuteAsync.
- Dependency injection can be used to access services in the background task.
- I used BackgroundService to implement a task scheduler for periodic email notifications.

27. What is the difference between .NET Standard and PCL (Portable Class Libraries)?

- .NET Standard defines a consistent API set across all .NET implementations.
- PCL allows targeting multiple platforms but has limited API availability compared to .NET Standard.
- .NET Standard is easier to maintain and supports more modern frameworks.
- I replaced a PCL library with .NET Standard to simplify cross-platform library sharing.

28. What is JIT Compiler?

- The Just-In-Time (JIT) compiler converts MSIL into native machine code at runtime.
- It optimizes code execution by compiling only the parts that are needed.
- JIT enables cross-platform execution by adapting to the target machine.
- I analyzed JIT performance using profiling tools to optimize a high-load application.

29. What does Common Language Specification (CLS) mean?

- CLS defines a subset of common features that all .NET languages must support.
- It ensures interoperability between different .NET languages like C# and VB.NET.
- CLS-compliant code can be reused across any .NET language.
- I wrote CLS-compliant libraries to ensure compatibility with a VB.NET legacy system.

```
// CLS-compliant example
public class MyClass
{
  public int Add(int a, int b) => a + b;
}
```

30. What's the difference between .NET Core, .NET Framework, and Xamarin?

- .NET Core is cross-platform and optimized for modern applications.
- .NET Framework is Windows-only, primarily for legacy applications.
- Xamarin is for cross-platform mobile and desktop app development.

- I used Xamarin to develop a single codebase app for Android and iOS.

31. Explain the difference between Managed and Unmanaged code in .NET.

- Managed code is executed by the CLR, which handles memory management and security.
- Unmanaged code is executed directly by the OS and requires manual memory management.
- Managed code benefits from garbage collection, while unmanaged code relies on IDisposable for cleanup.
- I worked with managed and unmanaged code in a project that required calling native libraries using P/Invoke.

```
[DllImport("user32.dll")]
public static extern int MessageBox(IntPtr hWnd, string text, string
caption, uint type);
```

32. What is FCL?

- Framework Class Library (FCL) is a collection of reusable classes, interfaces, and value types.
- It provides APIs for common programming tasks like file I/O, data access, and networking.
- FCL is part of the .NET Base Class Library (BCL) and is included in all .NET implementations.
- I used FCL to handle file management and database operations in an enterprise app.

```
using System.IO;
File.WriteAllText("example.txt", "Hello, FCL!");
```

33. What is Kestrel?

- Kestrel is a lightweight, cross-platform web server used in ASP.NET Core.
- It is built on libuv for asynchronous I/O and offers high performance.
- Kestrel can be used standalone or behind a reverse proxy like Nginx or IIS.
- I deployed an ASP.NET Core application with Kestrel as the server for high throughput.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.Run();
```

34. What is the difference between Class Library (.NET Standard) and Class Library (.NET Core)?

- Class Library (.NET Standard) is cross-platform and supports multiple .NET implementations.
- Class Library (.NET Core) is specific to .NET Core and newer .NET versions.
- .NET Standard is ideal for shared libraries, while .NET Core libraries leverage specific features of .NET Core.
- I used a .NET Standard library for a logging module shared across .NET Core and Xamarin projects.

35. What is Explicit Compilation?

- Explicit Compilation compiles source code to a binary format, such as MSIL, before runtime.
- It contrasts with implicit compilation, where code is compiled on the fly (e.g., Razor views).
- Explicit Compilation improves performance by reducing runtime overhead.
- I used explicit compilation to precompile views in an ASP.NET Core MVC app for faster response times.

```
dotnet build
```

36. Is there a way to catch multiple exceptions at once and without code duplication?

- Use a single catch block with multiple exception filters.
- The when keyword allows custom logic for filtering exceptions.
- This approach reduces code duplication and keeps the logic concise.
- I handled different exceptions with specific actions in a payment processing system.

```
try
{
   // Code that may throw exceptions
}
catch (InvalidOperationException ex) when
(ex.Message.Contains("specific"))
{
   Console.WriteLine("Handle InvalidOperationException");
}
catch (Exception ex)
{
   Console.WriteLine("Handle General Exception");
}
```

37. What is CoreCLR?

- CoreCLR is the runtime for .NET Core, providing execution, garbage collection, and type safety.
- It is modular and lightweight, designed for high-performance applications.
- CoreCLR supports Just-In-Time (JIT) compilation and multi-platform execution.
- I optimized application performance by profiling CoreCLR behavior in a microservices project.

38. What is the use of the IDisposable interface?

- IDisposable defines a method for releasing unmanaged resources explicitly.
- It is implemented in classes managing file handles, database connections, or other unmanaged resources.
- The using statement ensures proper disposal of resources.
- I used IDisposable to manage database connections in a data-heavy processing application.

```
using (var connection = new SqlConnection("connection string"))
{
   connection.Open();
   // Database operations
}
```

39. What is BCL?

- Base Class Library (BCL) is a core subset of the .NET Framework Class Library (FCL).
- It provides essential classes like System, System.IO, and System.Collections.
- BCL is the foundation for all .NET applications, ensuring consistent functionality.
- I used BCL classes like List<T> and DateTime for data handling and scheduling tasks.

40. What are the benefits of Explicit Compilation (AOT)?

- Ahead-Of-Time (AOT) compilation reduces startup time by precompiling code to native binaries.
- It eliminates JIT overhead, improving performance and predictability.
- AOT can produce smaller executables by removing unused code.
- I leveraged AOT to optimize a .NET Core application for deployment on low-resource devices.

```
dotnet publish -c Release -r linux-x64 --self-contained
```

---

41. Explain the difference between Task and Thread in .NET.

- Task represents an asynchronous operation, while Thread is a low-level OS resource.
- Task is managed by the Task Parallel Library (TPL), which optimizes thread usage.
- Tasks support cancellation and continuation, whereas threads are more static.
- I used tasks in a data processing project to handle multiple parallel operations efficiently.

```
Task.Run(() => Console.WriteLine("Running a Task"));
```

---

42. When should we use .NET Core and .NET Standard Class Library project types?

- Use .NET Standard for libraries shared across multiple .NET platforms.
- Use .NET Core for libraries leveraging .NET Core-specific features.
- For cross-platform compatibility, prefer .NET Standard.
- I used .NET Standard for a logging library and .NET Core for application-specific extensions.

---

43. Explain two types of deployment for .NET Core applications.

- Framework-Dependent Deployment (FDD): Requires the runtime to be installed on the target machine.
- Self-Contained Deployment (SCD): Bundles the runtime with the application, requiring no installation.
- FDD reduces size, while SCD ensures compatibility.
- I used SCD to deploy a web app on Linux without worrying about runtime installation.

---

44. Explain what is included in .NET Core.

- CoreCLR for runtime execution and garbage collection.
- CoreFX as the foundational library for .NET Core applications.
- CLI tools for development and publishing applications.
- I used these features in a project to develop and deploy a high-performance API.

---

45. What is the difference between .NET Core and .NET Framework?

- .NET Core is cross-platform, while .NET Framework is Windows-only.
- .NET Core is open-source and modular, whereas .NET Framework is monolithic.
- .NET Core supports modern workloads like microservices and containers.
- I migrated a legacy .NET Framework app to .NET Core to enable cross-platform support.

We have completed answers for 45 out of 68 questions. The remaining 23 questions are listed below with answers provided in the same format.

---

46. What's the difference between gRPC and WCF?

- gRPC is a modern RPC framework, while WCF is a legacy Windows-specific framework.
- gRPC uses HTTP/2 and Protocol Buffers for performance, while WCF supports various protocols.
- gRPC is cross-platform; WCF is tied to the .NET Framework and Windows.
- I replaced WCF with gRPC for a cross-platform microservices project using .NET Core.

```
// gRPC Service implementation
public class GreeterService : Greeter.GreeterBase
{
  public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
  {
    return Task.FromResult(new HelloReply { Message = "Hello " +
request.Name });
  }
}
```

---

47. Why does .NET Standard library exist?

- .NET Standard provides a consistent API across all .NET platforms.
- It simplifies sharing code between .NET Framework, .NET Core, and Xamarin.
- Developers can target multiple platforms with one library.
- I used .NET Standard to create a shared library for an analytics module reused in mobile and web projects.

---

48. Why shouldn't I use the Repository Pattern with Entity Framework?

- EF Core already implements a Unit of Work and Repository pattern.
- Adding another layer often leads to unnecessary abstraction and complexity.
- It can result in code duplication and maintenance challenges.
- I avoided using the Repository Pattern with EF Core and relied on DbContext directly for simplicity.

```
var products = await _dbContext.Products.ToListAsync();
```

---

49. When using DI in Controller, should I call Dispose on any injected service?

- No, the DI container manages the lifecycle of injected services.
- Disposing manually can lead to runtime errors.
- Scoped and transient services are disposed automatically at the end of the scope.
- I relied on DI for managing a DbContext's lifecycle in an ASP.NET Core web application.

---

50. Explain how Asynchronous tasks (async/await) work in .NET.

- async marks a method as asynchronous, enabling non-blocking calls.
- await suspends the method until the awaited task completes.
- It improves scalability by releasing threads during I/O operations.
- I implemented async/await in a web API to handle database operations efficiently.

```
public async Task<IActionResult> GetDataAsync()
{
  var data = await _dbContext.Data.ToListAsync();
  return Ok(data);
```

}

51. What's the difference between JIT and Roslyn?

- JIT compiles MSIL to native code at runtime; Roslyn compiles C# to MSIL at design time.
- JIT improves runtime performance through optimizations.
- Roslyn is a compiler-as-a-service for building dynamic .NET applications.
- I used Roslyn to implement real-time C# code evaluation in a project.

```
var syntaxTree = CSharpSyntaxTree.ParseText("int x = 10;");
```

52. What is the difference between IHost, IHostBuilder, and IHostedService?

- IHost represents the application's lifecycle, including starting and stopping.
- IHostBuilder is used to configure and build an IHost instance.
- IHostedService is for long-running background tasks.
- I used IHostBuilder and IHostedService to implement a background worker for an email service.

53. When to use Transient, Scoped, and Singleton DI service lifetimes?

- Transient: Use for lightweight, stateless services needed for each request.
- Scoped: Use for services tied to a single request or scope.
- Singleton: Use for services that should persist throughout the application's lifetime.
- I used Singleton for configuration services in a microservices project.

54. What is the difference between Hosted Services vs Windows Services?

- Hosted Services run in ASP.NET Core apps, while Windows Services run as standalone services.
- Hosted Services integrate with the application's lifecycle, Windows Services do not.
- Windows Services are OS-dependent; Hosted Services are cross-platform.
- I migrated a Windows Service to a Hosted Service for cross-platform compatibility.

55. Explain different types of Inheritance.

- Single Inheritance: A class inherits from one base class.
- Multilevel Inheritance: A class inherits from a derived class.
- Multiple Inheritance: Not directly supported in C#, but achieved via interfaces.
- I used multilevel inheritance in a project to manage different layers of functionality.

```
class Base { }
class Derived : Base { }
```

56. What is the difference between CIL and MSIL (IL)?

- Common Intermediate Language (CIL) and Microsoft Intermediate Language (MSIL) are the same.
- They represent platform-independent code executed by the CLR.
- The term CIL is used in ECMA standards, and MSIL is Microsoft's implementation.
- I debugged MSIL to analyze performance issues in a complex application.

57. What are the benefits of using JIT?

- JIT optimizes code for the specific runtime environment.
- It enables platform independence by compiling IL to machine code.
- Features like inlining and branch optimization improve performance.
- I leveraged JIT for high-performance scenarios in a CPU-intensive API.

58. Explain the Implicit Compilation process.

- Implicit Compilation occurs automatically when code changes are detected.
- Common in dynamic content rendering like Razor pages in ASP.NET Core.
- Simplifies development by removing manual build steps.
- I relied on implicit compilation for rapid prototyping of Razor views.

59. Why does .NET use a JIT compiler instead of just compiling the code once on the target machine?

- JIT allows platform independence by deferring compilation to runtime.
- It optimizes code based on runtime conditions.
- Reduces the application size compared to Ahead-Of-Time (AOT) compilation.
- I used JIT for flexibility while deploying applications on diverse server environments.

60. What is the difference between AppDomain, Assembly, Process, and Thread?

- AppDomain: Logical container for running .NET code; isolated environment.
- Assembly: Physical deployment unit containing MSIL.
- Process: OS-level container for executing an application.
- Thread: Smallest unit of execution within a process.
- I managed threads for parallel data processing in a multithreaded app.

61. Does .NET support Multiple Inheritance?

- No, .NET does not support multiple inheritance with classes.
- It supports multiple inheritance using interfaces.
- This avoids complexity and ambiguity in method resolution.
- I implemented multiple inheritance using interfaces to define modular components.

```
interface IShape { void Draw(); }
interface IColor { void Paint(); }
class Square : IShape, IColor { }
```

62. What is the difference between .NET Framework/Core and .NET Standard Class Library project types?

- .NET Framework/Core libraries are tied to specific platforms.
- .NET Standard libraries ensure compatibility across multiple .NET implementations.
- .NET Core libraries can leverage platform-specific features; .NET Standard cannot.
- I chose .NET Standard for libraries shared across web and mobile apps.

63. How to choose the target version of .NET Standard library?

- Select the version based on the minimum framework that supports your target platforms.
- Higher versions include more APIs but have limited platform support.
- Consider .NET Standard 2.0 for broad compatibility.
- I targeted .NET Standard 2.0 to support both .NET Framework 4.6.1 and .NET Core 2.0.

---

64. Could you name the difference between .NET Core, Portable, Standard, Compact, UWP, and PCL?

- .NET Core: Cross-platform, modern, modular.
- Portable Class Libraries (PCL): Limited API, legacy compatibility.
- .NET Standard: Unified API for all .NET platforms.
- Compact: For devices with limited resources, deprecated.
- Universal Windows Platform (UWP): For Windows 10 apps.
- I migrated a PCL library to .NET Standard for better compatibility and maintainability.

---

65. Explain when to use Finalized vs Dispose.

- Finalize: Used for unmanaged resource cleanup, called by the garbage collector.
- Dispose: Explicit resource cleanup, called manually or via using.
- Implement both in cases where unmanaged resources must be handled.
- I implemented both in a class managing database connections and file streams.

public void Dispose() { */* Cleanup */* }
~MyClass() { */* Finalizer Logic */* }

---

66. Explain some deployment considerations for Hosted Services.

- Ensure proper configuration for lifecycle management.
- Use scoped services to handle DI.
- Plan for resource cleanup and graceful shutdown.
- I deployed a Hosted Service to manage periodic data sync tasks in a distributed system.

---

67. How many types of JIT Compilations do you know?

- Pre-JIT: Compiles entire code during application deployment.
- Econo-J

IT: Compiles methods as they are called with minimal optimization.

- Normal-JIT: Compiles methods on demand with full optimization.
- I used Normal-JIT to balance startup performance and runtime efficiency.

---

68. What are some differences between X86, AnyCPU, and X64 compilations?

- X86: Compiled for 32-bit platforms.
- X64: Compiled for 64-bit platforms.
- AnyCPU: Runs on both but defaults to 64-bit if available.
- I compiled a project with AnyCPU to ensure compatibility across diverse environments.

Q1: What are the benefits of using EF?

- Entity Framework simplifies database access and management by abstracting database queries and updates into LINQ-based operations.
- It eliminates the need for most of the boilerplate code associated with ADO.NET or raw SQL queries, saving development time.
- EF integrates seamlessly with different database systems, making switching or scaling easier.
- Example: In a project, I used EF to handle CRUD operations without writing SQL, allowing rapid changes to the data model through migrations.

```
using (var context = new MyDbContext())
{
   var product = new Product { Name = "Laptop", Price = 1000 };
   context.Products.Add(product);
   context.SaveChanges();
}
```

Q2: What is Entity Framework?

- Entity Framework (EF) is an ORM (Object-Relational Mapping) tool for .NET developers.
- It enables developers to work with databases using .NET objects instead of SQL queries.
- EF supports multiple approaches, including Code First, Database First, and Model First, to model data.
- Example: I used EF Code First to create a database dynamically from C# classes in an e-commerce project.

```
public class Product
{
   public int ProductId { get; set; }
   public string Name { get; set; }
   public decimal Price { get; set; }
}
```

Q3: What is Conceptual Model?

- The Conceptual Model in EF represents the high-level view of the data structure as entities and relationships.
- It is part of the EDMX file and defines entities, their properties, and associations.
- It abstracts away the database-specific details and focuses on the domain model.
- Example: In one project, I customized the Conceptual Model to include computed properties like FullName for User entities.

Q4: What is Mapping?

- Mapping in EF refers to the process of linking the conceptual model to the storage model.
- It connects classes and properties in the domain model to database tables and columns.
- The mapping can be configured through Data Annotations, Fluent API, or the EDMX file.
- Example: I used Fluent API to map a composite key in the OrderDetails table in an inventory system.

```
modelBuilder.Entity<OrderDetail>()
   .HasKey(od => new { od.OrderId, od.ProductId });
```

Q5: What is pluralize and singularize in the Entity Framework?

- Pluralize means converting entity names to their plural forms when generating table names (e.g., Product becomes Products).
- Singularize converts table names back to singular forms when creating entities.
- These features help maintain consistency between domain classes and database tables.
- Example: In my project, I enabled pluralization to ensure consistent table naming conventions across the database schema.

Q6: What is the purpose of a DbContext class?

- The DbContext class is the primary class in EF for interacting with the database.
- It manages database connections and tracks changes to entities for saving data.
- DbContext provides APIs for querying and saving data using LINQ.
- Example: I used DbContext to fetch and update customer orders in a retail management system.

```
using (var context = new RetailDbContext())
{
   var orders = context.Orders.Where(o => o.Status == "Pending").ToList();
}
```

Q7: What is migration in Entity Framework?

- Migrations in EF are a feature to incrementally update the database schema while preserving existing data.
- They are used in the Code First approach to apply schema changes via code.
- Migrations generate C# files containing database commands for schema changes.
- Example: I used migrations to add a new column to the Products table during a feature upgrade.

```
Add-Migration AddDescriptionToProducts
Update-Database
```

Q8: Mention in what all scenarios Entity Framework can be applicable?

- CRUD operations in data-driven applications.
- Applications requiring a high level of abstraction over database queries.
- Multi-database support where switching providers is necessary.
- Example: I implemented EF in a logistics application to support multiple databases (SQL Server and PostgreSQL) seamlessly.

Q9: What are scalar and navigation properties in Entity Framework?

- Scalar properties map directly to database columns, representing primitive data types.
- Navigation properties link entities and enable navigation between related data.
- Scalar properties represent fields like Name or Price, while navigation properties represent relationships like Category or Orders.
- Example: I used navigation properties in a blogging platform to load related posts and comments.

Q10: Mention what is Code First Approach and Model First Approach in Entity Framework?

- Code First Approach defines the model in code, and the database schema is generated from these classes.
- Model First Approach uses a visual designer to define the model, generating the database schema and code from it.
- Code First is more flexible for developers comfortable with coding, while Model First is better for visually-oriented schema design.
- Example: I used Code First for rapid prototyping and Model First for a fixed schema in a corporate database.

Q11: What is Code First approach in Entity Framework?

- The Code First approach uses C# classes to define the domain model.

- EF generates the database schema based on these classes and their configurations.
- It supports migrations for schema changes without manual intervention.
- Example: I used Code First to build a dynamic catalog system for an online store.

Q12: What is Storage Model?

- The Storage Model in EF represents the database structure, including tables, columns, keys, and relationships.
- It is part of the EDMX file and reflects the actual database schema.
- The Storage Model is mapped to the Conceptual Model through mappings.
- Example: I adjusted the Storage Model to include custom indexing for frequently queried columns.

Q13: How can we handle concurrency in Entity Framework?

- Use a concurrency token column to track changes and detect conflicts.
- Implement optimistic concurrency by checking row versions during updates.
- Handle DbUpdateConcurrencyException in code to manage conflicts.
- Example: I used concurrency tokens to handle simultaneous updates in a multi-user accounting application.

Q14: Explain Lazy Loading, Eager Loading, and Explicit Loading?

- Lazy Loading loads related data when accessed for the first time.
- Eager Loading loads related data along with the primary entity query.
- Explicit Loading loads related data explicitly via code after the primary query.
- Example: I used Eager Loading to reduce query count in a report generation module.

Q15: Could you explain the difference between Optimistic vs Pessimistic locking?

- Optimistic locking assumes no conflicts and checks for changes at update time.
- Pessimistic locking prevents conflicts by locking data during access.
- Optimistic is better for read-heavy scenarios; Pessimistic is used in write-heavy or critical data scenarios.
- Example: I implemented Optimistic Locking to handle edits in a collaborative document editing tool.

```
try
{
   context.SaveChanges();
}
catch (DbUpdateConcurrencyException ex)
{
   // Handle conflict
}
```

Q16: What are POCO classes in Entity Framework?

- POCO (Plain Old CLR Objects) classes are simple C# classes without any EF-specific base classes or attributes.
- They represent the domain model and are used to maintain the separation of concerns.
- POCO classes are lightweight, making them testable and easier to maintain.
- Example: In a blogging platform, I used POCO classes for entities like Post and Comment to maintain a clean domain model.

```
public class Post
{
   public int PostId { get; set; }
   public string Title { get; set; }
   public string Content { get; set; }
   public ICollection<Comment> Comments { get; set; }
}
```

Q17: What is Optimistic Locking?

- Optimistic Locking allows multiple users to access a resource but detects conflicts when saving changes.
- It uses a concurrency token, like a version number or timestamp, to check for changes.
- This approach is suitable for applications with low conflict probability.
- Example: I used Optimistic Locking in an inventory system to prevent stock updates from overwriting each other.

Q18: What are complex types in Entity Framework?

- Complex types are non-scalar properties of an entity that map to multiple columns in a table.
- They cannot have keys and are always embedded within an entity.
- Complex types are used to group related fields for better organization.
- Example: I used a complex type Address for entities like Customer and Supplier to avoid redundancy.

```
public class Address
{
   public string Street { get; set; }
   public string City { get; set; }
   public string PostalCode { get; set; }
}
```

Q19: What are the different approaches supported in the Entity Framework to create Entity Model?

- Code First: Define the model in code and generate the database schema.
- Database First: Start with an existing database and generate the model.
- Model First: Design the model visually and generate the database schema and code.
- Example: I used Database First for a legacy system integration project to quickly scaffold the database schema.

Q20: What is EF Data Access Architecture?

- EF Data Access Architecture involves the layers of the application that interact with the database through EF.
- The architecture includes the domain model, DbContext, LINQ queries, and database provider.
- It abstracts data access logic, promoting separation of concerns.
- Example: In a multi-tenant application, I used a layered architecture with EF to manage tenant-specific data.

Q21: Can you explain Lazy Loading in a detailed manner?

- Lazy Loading defers the loading of related data until it is accessed for the first time.
- It uses proxy objects to intercept property calls and load data dynamically.
- Lazy Loading can lead to performance issues if not managed properly (e.g., N+1 queries).
- Example: In a forum application, Lazy Loading was used for comments to load them only when viewed by the user.

Q22: What are the advantages and disadvantages of Database First Approach?

- Advantages: Suitable for existing databases, provides a clear starting point, and minimizes initial setup effort.
- Disadvantages: Less flexibility for model customizations and requires database changes for schema updates.
- Database First is ideal for integrating with legacy databases.
- Example: I used Database First for an HR system to work with an established database schema.

Q23: What are the advantages of Model First Approach?

- Visual design of models allows for easier collaboration with non-technical stakeholders.
- Automatically generates the database schema and code from the model.
- Provides a centralized view of the data structure.
- Example: I used Model First to design and deploy a database for a project management tool.

Q24: What is Eager Loading?

- Eager Loading loads related entities as part of the initial query.
- It uses the Include method to specify the relationships to load.
- Eager Loading reduces query count but may fetch unnecessary data.
- Example: In an order tracking system, I used Eager Loading to fetch orders and their associated products in one query.

```
var orders = context.Orders.Include(o => o.Products).ToList();
```

Q25: What is the role of Entity Client Data Provider?

- It serves as a bridge between the Entity Framework and the underlying database provider.
- Converts LINQ queries to database-specific SQL queries.
- Facilitates communication between the Conceptual Model and the Storage Model.
- Example: I used Entity Client Data Provider to support cross-database operations in a hybrid environment.

Q26: What are the components of Entity Framework Architecture?

- Conceptual Model: Represents the high-level view of the data.
- Storage Model: Represents the database schema.
- Mapping: Links the conceptual and storage models.
- Example: I worked on an EDMX file in an analytics system to configure these components for complex queries.

Q27: Explain how you can load related entities in EF?

- Use Lazy Loading for on-demand loading.
- Use Eager Loading with the Include method to load related data in the initial query.
- Use Explicit Loading to load related data explicitly in code.
- Example: I used Explicit Loading to fetch related customer data in a billing system only when needed.

Q28: What is the importance of EDMX file in Entity Framework?

- EDMX (Entity Data Model XML) file contains the Conceptual Model, Storage Model, and Mapping.
- It serves as a blueprint for the database and model relationships.
- EDMX files are essential for Model First and Database First approaches.
- Example: I modified an EDMX file to add navigation properties in a legacy reporting system.

Q29: What are the advantages/disadvantages of Code First Approach?

- Advantages: Highly flexible, allows use of migrations, and no dependency on the database schema.
- Disadvantages: Initial setup requires more effort and knowledge of EF configurations.
- Ideal for greenfield projects or rapid prototyping.
- Example: I used Code First for a healthcare system to dynamically evolve the database schema during development.

Q30: When would you use EF6 vs EF Core?

- Use EF6 for mature projects requiring full feature support and compatibility with .NET Framework.
- Use EF Core for lightweight, high-performance, and cross-platform applications.
- EF Core is better for new projects with modern requirements.
- Example: I chose EF Core for a cross-platform mobile app to leverage its performance and flexibility.

Q31: Which type of loading is good in which scenario?

- Lazy Loading: Best for small, infrequent, or on-demand data access scenarios.
- Eager Loading: Ideal for scenarios where related data is always needed to avoid additional queries.
- Explicit Loading: Useful when data requirements vary and are loaded selectively.
- Example: In an online store, Eager Loading was used for frequently accessed product categories and their details.

Q32: Can you explain CSDL, SSDL, and MSL sections in an EDMX file?

- **CSDL (Conceptual Schema Definition Language):** Defines the conceptual model, including entities and relationships.
- **SSDL (Store Schema Definition Language):** Represents the database schema, including tables and columns.
- **MSL (Mapping Specification Language):** Maps the conceptual model to the storage model.
- Example: I customized MSL to map a composite key to a domain model in a logistics application.

Q33: What are T4 templates?

- T4 (Text Template Transformation Toolkit) templates are code generation tools used in EF to generate classes based on the model.
- They generate entity classes, DbContext, and other supporting code.
- T4 templates can be customized to fit specific project requirements.
- Example: I modified a T4 template to include custom logging in entity classes for an auditing system.

Q34: Is DbContext thread-safe?

- DbContext is not thread-safe and should not be shared across threads.
- Each thread or operation should have its own instance of DbContext.
- Using DbContext in a multi-threaded environment may lead to unexpected behaviors.
- Example: I ensured a separate DbContext instance for each API request in a RESTful service.

Q35: How can you enhance the performance of Entity Framework?

- Use AsNoTracking() for read-only queries to avoid change tracking overhead.
- Optimize queries with LINQ and avoid loading unnecessary data.
- Batch updates and inserts to minimize database round-trips.
- Example: I used AsNoTracking() in a dashboard application to improve performance for large dataset queries.

```
var products = context.Products.AsNoTracking().ToList();
```

Q36: What is the difference between ObjectContext and DbContext?

- DbContext is a simpler and lightweight API introduced in EF 4.1 for easier use.

- ObjectContext is the older, more complex API with additional features like ObjectStateManager.
- DbContext supports modern patterns like dependency injection and LINQ directly.
- Example: I migrated a legacy project from ObjectContext to DbContext for better readability and performance.

Q37: What is faster - ADO.NET or ADO.NET Entity Framework?

- ADO.NET is faster due to its lower-level operations and minimal abstraction overhead.
- EF provides productivity and maintainability benefits at the cost of some performance.
- ADO.NET is ideal for performance-critical applications, while EF suits business applications.
- Example: I used ADO.NET for batch data processing but EF for standard CRUD operations in a CMS.

Q38: Name some differences between Express vs Recoverable messages.

- Express messages are stored in memory and faster but not durable.
- Recoverable messages are stored on disk, ensuring durability and reliability.
- Express messages are suitable for high-performance, non-critical applications.
- Example: I used recoverable messages in a financial transaction system to ensure data integrity.

Q39: What types of system-generated messages do you know?

- System-generated error messages for exceptions or invalid operations.
- Log messages for tracking application or system behavior.
- Audit trail messages to record user or system actions.
- Example: I used system-generated log messages to monitor API calls in a microservices architecture.

Q40: Why shouldn't I use the Repository Pattern with Entity Framework?

- EF already acts as a repository by providing DbSet for managing entities.
- Adding another repository layer can lead to redundancy and unnecessary complexity.
- It may hinder EF's advanced features like LINQ and change tracking.
- Example: I avoided a custom repository pattern in an EF project to simplify data access and leverage EF's capabilities.

Q41: What is the relationship between Repository and Unit of Work?

- The Repository handles CRUD operations for a specific entity.
- The Unit of Work manages transactions and tracks changes across multiple repositories.
- Together, they provide a cohesive way to manage data access and ensure consistency.
- Example: I used both patterns in a modular application to decouple business logic from data access.

Q42: What are the disadvantages of using static DbContext?

- It may lead to memory leaks due to retained connections and untracked entities.
- Not thread-safe, causing issues in multi-threaded applications.
- Difficult to test and manage lifecycle in large applications.
- Example: I replaced a static DbContext with dependency injection for a scalable web application.

Q43: What is the difference between POCO, Code First, and simple EF approach?

- POCO: Focuses on plain CLR objects without EF dependencies.
- Code First: Generates a database from domain classes and migrations.
- Simple EF: Typically involves Database First or EDMX-driven development.
- Example: I used POCO with Code First to create a clean and flexible data layer in a project.

Q44: Could you explain Pessimistic locking?

- Pessimistic locking locks a resource when it is accessed to prevent concurrent modifications.
- It ensures data integrity in high-contention environments but may cause performance issues.
- Pessimistic locking is achieved using transactions or specific SQL locking hints.
- Example: I used FOR UPDATE in a banking application to prevent overdraft issues during concurrent withdrawals.

Q45: What's the difference between LINQ to SQL and Entity Framework?

- LINQ to SQL only supports SQL Server, while EF supports multiple databases.
- EF offers features like Code First, migrations, and navigation properties, which LINQ to SQL lacks.
- LINQ to SQL is simpler but less flexible compared to EF.
- Example: I migrated a LINQ to SQL project to EF for PostgreSQL compatibility and improved scalability.

Q46: What is the difference between Code First, Model First, and Database First?

- Code First starts with C# classes and generates the database.
- Model First uses a visual designer to create the model and database.
- Database First scaffolds the model and code from an existing database.
- Example: I used Database First for integrating an existing ERP system with minimal disruption.

Q47: How can we do pessimistic locking in Entity Framework?

- Use explicit SQL queries or stored procedures with locking hints like WITH (ROWLOCK, UPDLOCK).
- Execute raw SQL commands using context.Database.ExecuteSqlCommand.
- Ensure transactions are managed to avoid deadlocks.
- Example: I implemented pessimistic locking in EF to secure inventory updates during a flash sale.

Q48: What is the difference between Automatic Migration vs Code-based Migration?

- Automatic Migrations apply schema changes without requiring migration scripts.
- Code-based Migrations involve writing migration scripts for fine-grained control.
- Automatic Migrations are faster but less precise than Code-based Migrations.
- Example: I used Code-based Migrations in a production app to version-control schema changes.

Q49: What difference does .AsNoTracking() make?

- Disables change tracking, improving performance for read-only operations.
- Reduces memory overhead for queries involving large datasets.
- Ideal for scenarios where the entities won't be updated.
- Example: I used .AsNoTracking() for read-only dashboards to reduce query execution time.

Q50: What are the advantages and disadvantages of creating a Global Entities Context for the application?

- Advantages: Centralized access to data, simplifying configuration.
- Disadvantages: High memory usage, potential data inconsistency, and thread-safety issues.
- Not suitable for large or multi-threaded applications.
- Example: I replaced a global context with a scoped DbContext in a microservices-based project.

Q51: When would you use SaveChanges(false) + AcceptAllChanges()?

- Use SaveChanges(false) to save changes without affecting the state of tracked entities.
- Call AcceptAllChanges() explicitly to mark entities as unchanged after manual operations.
- Useful in scenarios where custom transaction handling is required.
- Example: I used this approach in a batch processing system to commit changes in chunks.

Q52: What is client wins and store wins mode in Entity Framework concurrency?

- **Client Wins:** Overwrites database values with client changes in case of a conflict.
- **Store Wins:** Discards client changes and retains database values.
- These modes are used to resolve concurrency conflicts.
- Example: I applied Store Wins in a ticket booking system to prioritize server-side accuracy.

Q53: What's the difference between .SaveChanges() and .AcceptAllChanges()?

- .SaveChanges() persists changes to the database and marks entities as unchanged.
- .AcceptAllChanges() only updates the state of entities without saving to the database.
- Typically, .SaveChanges() calls .AcceptAllChanges() internally.
- Example: I used .AcceptAllChanges() in a custom transaction rollback handler.

Q54: Can I use Entity Framework 6 in .NET Core?

- EF6 can be used in .NET Core, but EF

Core is preferred for better performance and features.

- EF6 is suitable when migrating legacy apps to .NET Core without changing data access logic.
- Example: I used EF6 in a .NET Core app to leverage an existing EDMX model during migration.

Q55: How do you handle multiple DbContexts in a single application?

- Use dependency injection to configure multiple DbContexts with different lifetimes.
- Ensure each DbContext handles distinct parts of the domain or database.
- Example: I managed multiple DbContexts in a multi-tenant app, each pointing to a tenant-specific database.

Q56: What is a shadow property in Entity Framework Core?

- Shadow properties are not defined in the entity class but exist in the EF Core model.
- They are primarily used for audit fields like CreatedDate or UpdatedDate.
- Access shadow properties using the EF Core ChangeTracker or raw queries.
- Example: I used shadow properties to track entity modification timestamps without cluttering domain models.

```
modelBuilder.Entity<Product>().Property<DateTime>("LastUpdated");
```

Q57: What are the features of Entity Framework Core 6?

- Supports many-to-many relationships without explicit join tables.
- Improved performance for LINQ queries and data seeding.
- Introduced compiled models for faster startup times.
- Example: I utilized EF Core 6's compiled models in a high-throughput API to reduce latency.

```
var options = new DbContextOptionsBuilder<MyDbContext>()
   .UseModel(compiledModel)
   .Options;
```

HTML Interview Questions

1. What is the difference between id and class attributes in HTML?

- id is unique and used to identify a single element, while class can be shared by multiple elements.
- id is used for specific tasks like targeting elements in JavaScript or CSS, whereas class is for grouping elements.
- You can use the # selector in CSS for id and the . selector for class.
- Example: In a project, I used id for a login form and class for styling multiple buttons.

```
<div id="login-form">Login Form</div>
<button class="btn">Submit</button>
```

2. Explain the difference between <section> and <div>.

- <section> is semantic, indicating a thematic grouping of content.
- <div> is non-semantic and used for general-purpose grouping.
- <section> provides context for accessibility and SEO, while <div> requires extra attributes for the same.
- Example: I used <section> to divide the blog into articles and <div> for layout structuring.

```
<section>
  <h2>Blog Post</h2>
  <p>Content of the blog.</p>
</section>
<div class="container">Wrapper for layout</div>
```

3. What is the difference between inline and block-level elements?

- Inline elements occupy only as much width as necessary and don't start on a new line.
- Block-level elements take up the full width of their container and start on a new line.
- Common inline elements include <span> and <a>, while <div> and <p> are block elements.
- Example: I styled a navigation menu using <a> (inline) and organized sections using <div> (block).

```
<a href="#">Home</a>
<div class="block-element">Content</div>
```

4. How does the <meta> tag work in HTML?

- It provides metadata about the HTML document, such as character set and viewport settings.
- Metadata doesn't appear on the page but affects how it's rendered and indexed by search engines.
- Examples include <meta charset="UTF-8"> and <meta name="viewport" content="width=device-width, initial-scale=1.0">.
- Example: I used <meta> to ensure responsiveness in my e-commerce project.

```
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

5. What is the purpose of the <link> tag in HTML?

- It connects the HTML document to external resources like stylesheets or icons.
- Commonly used for including CSS files with rel="stylesheet".
- It can also be used for preloading assets with attributes like rel="preload".
- Example: In my project, I used <link> to include a CSS file for consistent theming.

```
<link rel="stylesheet" href="styles.css">
```

6. What are semantic HTML tags? Provide examples.

- Semantic tags clearly define the meaning and structure of the content.
- Examples include <header>, <article>, <footer>, and <aside>.
- They improve accessibility and SEO by providing context to the content.
- Example: I used semantic tags to enhance the readability of my portfolio website.

```
<header>
  <h1>Welcome to My Portfolio</h1>
</header>
<article>
  <p>About me section...</p>
</article>
```

7. How does the <canvas> element work in HTML?

- <canvas> is used for rendering graphics, like drawings or animations, using JavaScript.
- It requires a width and height attribute for defining the canvas area.
- Graphics are created using a 2D or 3D context with JavaScript.
- Example: I implemented an interactive graph on a dashboard using <canvas>.

```
<canvas id="myCanvas" width="200" height="100"></canvas>
<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
  ctx.fillStyle = "blue";
  ctx.fillRect(10, 10, 100, 50);
</script>
```

8. What is the difference between async and defer attributes in a <script> tag?

- async loads the script asynchronously and executes it as soon as it's available.
- defer loads the script asynchronously but waits until the HTML parsing is complete before execution.
- Use async for independent scripts and defer for scripts that depend on the DOM.
- Example: I used defer in my project to load a script that manipulates DOM elements.

```
<script src="script.js" defer></script>
```

9. What are empty elements in HTML?

- Elements that do not have closing tags and cannot contain content.
- Examples include <br>, <img>, <meta>, and <input>.
- They are self-closing and used for standalone purposes like inserting a line break or an image.
- Example: I used <img> for displaying product images in a catalog.

```
<img src="product.jpg" alt="Product Image">
```

10. Explain the purpose of the <template> tag.

- The <template> tag holds HTML content that is not rendered on page load.
- It is useful for reusing blocks of content dynamically using JavaScript.
- Content inside a <template> is inert until cloned and appended to the DOM.
- Example: I used <template> to dynamically generate cards in a shopping app.

```
<template id="card-template">
  <div class="card">
    <h3>Product Name</h3>
  </div>
</template>
<script>
  const template = document.getElementById("card-
template").content.cloneNode(true);
  document.body.appendChild(template);
</script>
```

11. What is the purpose of the data-* attributes in HTML?

- Custom attributes to store extra data directly on HTML elements.
- Useful for transferring data to JavaScript without affecting layout or performance.

- Data is accessed via dataset in JavaScript.
- Example: I used data-* to store user preferences in an interactive app.

```
<div id="user" data-role="admin">User Info</div>
<script>
  const role = document.getElementById("user").dataset.role;
  console.log(role); // Output: admin
</script>
```

12. What is the difference between <b> and <strong> tags?

- <b> is used for stylistic purposes to bold text, without semantic meaning.
- <strong> conveys importance and has semantic meaning in addition to styling.
- Search engines and screen readers prioritize <strong> over <b>.
- Example: I used <strong> in a terms and conditions page to emphasize critical points.

```
<p><strong>Important:</strong> Read the guidelines carefully.</p>
```

13. How does the contenteditable attribute work in HTML?

- It makes an element editable directly in the browser.
- Accepts values true (editable) or false (non-editable).
- Useful for creating rich-text editors or inline editing features.
- Example: I implemented contenteditable in a project for live profile editing.

```
<div contenteditable="true">Edit this text</div>
```

14. Explain the <details> and <summary> tags in HTML.

- <details> creates a collapsible container for content.
- <summary> defines the visible heading that the user clicks to expand/collapse the details.
- It's a semantic and interactive alternative to toggle visibility.
- Example: I used <details> to display FAQs on a support page.

```
<details>
  <summary>What is your refund policy?</summary>
  <p>We offer a 30-day refund policy.</p>
</details>
```

15. What is the purpose of the autofocus attribute in HTML?

- Automatically focuses an input element when the page loads.
- It's applied to elements like <input>, <textarea>, or <button>.
- Improves usability by directing the user to the primary interaction field.
- Example: I used autofocus on the search bar of an e-commerce homepage.

```
<input type="text" autofocus placeholder="Search products">
```

HTML Interview Questions (Continued)

16. What is the purpose of the <picture> element in HTML?

- The <picture> element is used for providing multiple image sources for responsive designs.
- It allows specifying images for different screen sizes or formats using <source> tags.
- Helps optimize performance by loading the most appropriate image for the device.
- Example: I used <picture> to display high-resolution images on desktop and smaller ones on mobile.

```
<picture>
  <source srcset="image-large.jpg" media="(min-width: 800px)">
  <source srcset="image-small.jpg" media="(max-width: 799px)">
  <img src="image-default.jpg" alt="Responsive Image">
</picture>
```

17. How do you include external JavaScript in an HTML document?

- Use the <script> tag with the src attribute pointing to the script file.
- Place the <script> tag in the <head> or before the closing <body> tag.
- Add async or defer attributes to control script execution timing.
- Example: In my project, I included a JavaScript file to handle form validation.

```
<script src="form-validation.js" defer></script>
```

18. What are web storage APIs, and how are they used?

- Web storage APIs, localStorage and sessionStorage, store data in the browser.
- localStorage persists data across sessions, while sessionStorage clears after the tab is closed.
- Data is stored as key-value pairs and accessed using JavaScript.
- Example: I used localStorage to save user preferences in a dashboard application.

```
localStorage.setItem("theme", "dark");
console.log(localStorage.getItem("theme")); // Output: dark
```

19. What is the purpose of the <noscript> tag?

- Displays alternative content if JavaScript is disabled or unsupported in the browser.
- Useful for providing fallback options to enhance user experience.
- Typically contains instructions or links to upgrade the browser or enable JavaScript.
- Example: I used <noscript> to show a message for a JavaScript-based interactive map.

```
<noscript>
  <p>Please enable JavaScript to use the interactive map.</p>
</noscript>
```

20. What is the purpose of the aria-* attributes in HTML?

- ARIA (Accessible Rich Internet Applications) attributes improve web accessibility.
- Used to define roles, states, and properties for screen readers.
- Examples include aria-label, aria-hidden, and aria-live.
- Example: I implemented aria-label to improve navigation accessibility in a web app.

```
<button aria-label="Submit form">Submit</button>
```

21. How does the <fieldset> element work in HTML?

- Groups related elements in a form for better organization and accessibility.
- Typically used with the <legend> element to provide a caption.
- Enhances form semantics and user experience.
- Example: I used <fieldset> to group address fields in a registration form.

```
<fieldset>
  <legend>Address</legend>
  <input type="text" placeholder="Street">
  <input type="text" placeholder="City">
</fieldset>
```

22. What is the difference between relative, absolute, and fixed URLs?

- Relative URLs are based on the location of the current document.
- Absolute URLs specify the complete path, including protocol and domain.
- Fixed URLs typically reference local files using a specific root.
- Example: I used relative URLs to link internal pages and absolute URLs for external resources.

```
<a href="/about.html">Relative URL</a>
<a href="https://example.com">Absolute URL</a>
```

23. What are the limitations of the <iframe> element?

- Can cause performance issues due to additional resource loading.
- Some websites block embedding in <iframe> using the X-Frame-Options header.
- Less secure, as it may expose the site to clickjacking attacks.
- Example: I used <iframe> to embed Google Maps but ensured proper security measures.

```
<iframe src="https://maps.google.com" width="600"
height="400"></iframe>
```

24. What is the purpose of the download attribute in <a> tag?

- Enables file downloads when the link is clicked.
- Specifies a filename for the downloaded file.
- Improves user experience by skipping the file preview step.
- Example: I used the download attribute to allow users to download invoices.

```
<a href="invoice.pdf" download="Invoice.pdf">Download Invoice</a>
```

25. How does the required attribute in form elements work?

- Makes the input field mandatory for form submission.
- It is supported by <input>, <textarea>, and <select> elements.
- Provides basic client-side validation without JavaScript.
- Example: I used required to ensure users fill in critical fields in a contact form.

```
<input type="text" required placeholder="Enter your name">
```

26. What are the differences between the <ol> and <ul> elements?

- <ol> creates an ordered list with numbered or lettered items.
- <ul> creates an unordered list with bullet points.
- Both use <li> for list items, but their purposes differ.
- Example: I used <ol> for steps in a process and <ul> for navigation links.

```
<ol>
 <li>Step 1</li>
 <li>Step 2</li>
</ol>
<ul>
 <li>Home</li>
 <li>About</li>
</ul>
```

27. How does the placeholder attribute work in form fields?

- Displays a short hint inside the input field before the user enters data.
- Used in <input> and <textarea> elements.
- It disappears once the user starts typing.
- Example: I used placeholder in a login form to guide users.

```
<input type="text" placeholder="Enter your username">
```

28. What is the use of the <time> element?

- Represents a specific point in time or a duration.
- Useful for events, publishing dates, or time-based data.
- Can include the datetime attribute for machine-readable formatting.
- Example: I used <time> to display event dates in a calendar app.

```
<time datetime="2024-12-01">December 1, 2024</time>
```

29. What are the autocomplete attribute values in form fields?

- on enables and off disables browser autofill suggestions.
- Helps improve the user experience by remembering input values.
- Supported by <input>, <textarea>, and <form> elements.
- Example: I enabled autocomplete for email fields in a signup form.

```
<input type="email" autocomplete="on" placeholder="Enter your email">
```

30. How does the <progress> element work in HTML?

- Displays a progress bar for tasks like uploads or installations.
- Requires value and max attributes to define progress and completion.
- Provides a semantic way to represent progress.
- Example: I used <progress> to show file upload status in a project.

```
<progress value="50" max="100">50%</progress>
```

CSS Interview Questions

31. What is the difference between inline, inline-block, and block elements in CSS?

- inline: Elements do not start on a new line and only take up as much width as necessary.
- inline-block: Similar to inline but allows setting width and height.
- block: Elements start on a new line and take up the full width available.
- Example: I used inline-block for navigation links and block for main content sections.

```
.nav-item {
 display: inline-block;
 margin: 5px;
}
```

32. How does the z-index property work in CSS?

- Defines the stack order of elements; higher values are closer to the front.
- Works only on elements with a position value other than static.
- Negative values are allowed to push elements further back.
- Example: I used z-index to overlay a modal on top of page content.

```
.modal {
 position: fixed;
 z-index: 1000;
}
```

33. What are pseudo-classes in CSS?

- Pseudo-classes define the special state of an element, such as :hover or :nth-child().
- They do not modify the content directly but change its appearance based on the state.
- Useful for user interactions and conditional styling.
- Example: I used :hover to create button hover effects in a landing page.

```
button:hover {
 background-color: blue;
 color: white;
}
```

34. What is the difference between relative, absolute, fixed, and sticky positioning in CSS?

- relative: Positions the element relative to its normal position.
- absolute: Positions the element relative to its nearest positioned ancestor.
- fixed: Positions the element relative to the viewport, staying in place on scroll.
- Example: I used sticky for a header that stays at the top while scrolling down a page.

```
header {
 position: sticky;
 top: 0;
 background: white;
}
```

35. How does the flexbox layout work in CSS?

- Provides a flexible layout for distributing space among items in a container.
- Uses properties like justify-content, align-items, and flex-wrap.
- Simplifies alignment and spacing for both horizontal and vertical layouts.
- Example: I used flexbox to center align content in a card layout.

```
.container {
 display: flex;
 justify-content: center;
 align-items: center;
 height: 100vh;
}
```

36. What is the purpose of the grid-template-areas property in CSS Grid?

- Defines named grid areas to simplify the placement of elements in the grid.

- Areas are specified using strings that map out the grid layout.
- Works with the grid-area property to position elements.
- Example: I used grid-template-areas to organize a header, sidebar, and main content in a dashboard.

```
.container {
  display: grid;
  grid-template-areas:
    "header header"
    "sidebar main";
}
```

37. What is the difference between em, rem, and px in CSS?

- px: Absolute unit, fixed size independent of parent or root.
- em: Relative to the font size of the parent element.
- rem: Relative to the root element's font size.
- Example: I used rem for scalable typography in a responsive design.

```
h1 {
  font-size: 2rem;
}
```

38. How does the clip-path property work in CSS?

- Clips an element to a specific shape, such as a circle or polygon.
- Accepts values like circle(), ellipse(), and polygon().
- Used for creative design effects and masking content.
- Example: I used clip-path to create custom shapes for images in a gallery.

```
.image {
  clip-path: circle(50%);
}
```

39. What are the different types of CSS selectors?

- Universal selector (*): Selects all elements.
- Type selector (element): Targets specific tags like div or p.
- Class selector (.class): Selects elements with a specific class.
- Example: I used a combination of type and class selectors to style a blog post layout.

```
p.intro {
  font-size: 1.2em;
  color: gray;
}
```

40. What is the difference between absolute and relative units in CSS?

- Absolute units (px, in, cm): Fixed size, unaffected by parent elements.
- Relative units (em, %, vw, vh): Dependent on parent or viewport size.
- Absolute units provide consistency, while relative units enable responsiveness.
- Example: I used % and vw for responsive layout designs.

```
.container {
  width: 80%;
  height: 50vh;
}
```

41. What are media queries, and how are they used?

- Media queries apply styles based on device characteristics like screen size.
- Use @media rules to define styles for different breakpoints.
- Commonly used for responsive designs.
- Example: I used media queries to adjust grid layouts for tablets and mobile devices.

```
@media (max-width: 768px) {
  .container {
    flex-direction: column;
  }
}
```

42. What is the purpose of the transition property in CSS?

- Adds smooth animations between state changes, like hover effects.
- Can specify properties like duration, delay, and easing functions.
- Simplifies creating animations without JavaScript.
- Example: I used transition for button hover effects in an interactive UI.

```
button {
  transition: background-color 0.3s ease;
}
button:hover {
  background-color: blue;
}
```

43. How does the object-fit property work in CSS?

- Defines how an image or video fits within a container.
- Common values: cover, contain, fill, and none.
- Prevents distortion by maintaining the aspect ratio of media.
- Example: I used object-fit: cover for profile pictures in a user list.

```
img {
  width: 100px;
  height: 100px;
  object-fit: cover;
}
```

44. What are keyframe animations in CSS?

- Define animations by breaking them into key states using @keyframes.
- Specify intermediate steps for properties like position or color.
- Animations are applied with the animation property.
- Example: I created a loading spinner using keyframe animations.

```
@keyframes spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
.loader {
  animation: spin 2s infinite linear;
}
```

45. What is the difference between visibility: hidden and display: none?

- visibility: hidden: Hides the element but still occupies space in the layout.
- display: none: Completely removes the element from the layout.
- visibility: hidden is useful for toggling visibility without layout changes.
- Example: I used display: none to collapse a sidebar menu on mobile.

```
.sidebar {
  display: none;
}
```

CSS Interview Questions (Continued)

46. How does the overflow property work in CSS?

- Controls what happens to content that overflows the container's bounds.
- Common values: visible, hidden, scroll, and auto.
- Ensures layout stability by handling overflow content gracefully.
- Example: I used overflow: scroll to enable scrolling for a long table in a dashboard.

```
.table-container {
  height: 300px;
  overflow: scroll;
}
```

47. What is the purpose of the box-shadow property in CSS?

- Adds shadow effects around elements.

- Defines offsets, blur radius, spread radius, and shadow color.
- Can create depth and elevate elements visually.
- Example: I applied box-shadow to buttons for a material design-inspired UI.

```
.button {
  box-shadow: 2px 2px 5px rgba(0, 0, 0, 0.3);
}
```

48. What is the difference between min-width, max-width, and width?

- width: Sets the exact width of an element.
- min-width: Defines the minimum width an element can have.
- max-width: Sets the maximum width, allowing flexibility.
- Example: I used max-width to constrain content in a responsive design.

```
.container {
  width: 100%;
  max-width: 1200px;
}
```

49. How does the opacity property work in CSS?

- Controls the transparency level of an element, with values from 0 (fully transparent) to 1 (fully opaque).
- Affects both the element and its content.
- Often combined with transitions for fade effects.
- Example: I used opacity to create hover effects for image captions.

```
.image-caption {
  opacity: 0;
  transition: opacity 0.5s;
}
.image:hover .image-caption {
  opacity: 1;
}
```

50. What are CSS variables, and how are they used?

- Custom properties defined using the -- prefix and accessed with var().
- Provide reusability and easier maintenance of styles.
- Can be scoped to specific elements or globally in :root.
- Example: I used CSS variables for consistent theming in a web application.

```
:root {
  --primary-color: #3498db;
}
.button {
  background-color: var(--primary-color);
}
```

51. How does the float property work in CSS?

- Positions elements to the left or right of their container.
- Commonly used for creating text wrapping around images.
- Requires clear to avoid overlapping issues with succeeding elements.
- Example: I used float to align product images beside descriptions in an e-commerce site.

```
.image {
  float: left;
  margin-right: 10px;
}
```

52. What is the difference between inline and inline-block display values?

- inline: Allows elements to flow inline with text but cannot set width/height.
- inline-block: Combines inline flow with the ability to set width/height.
- Suitable for creating flexible layouts with precise spacing.
- Example: I used inline-block for responsive grid items in a gallery.

```
.gallery-item {
  display: inline-block;
  width: 30%;
}
```

53. How does the cursor property work in CSS?

- Changes the mouse pointer style when hovering over an element.
- Common values: pointer, default, text, and not-allowed.
- Enhances UX by visually indicating interactions.
- Example: I used cursor: pointer for interactive buttons in a form.

```
.button {
  cursor: pointer;
}
```

54. What is the difference between absolute and fixed positioning in CSS?

- absolute: Positions the element relative to its nearest positioned ancestor.
- fixed: Positions the element relative to the viewport, unaffected by scrolling.
- Both require top, right, bottom, or left properties for placement.
- Example: I used fixed for a sticky navigation bar on a scrolling page.

```
.navbar {
  position: fixed;
  top: 0;
  width: 100%;
}
```

55. How do you use the nth-child pseudo-class in CSS?

- Targets elements based on their position within a parent.
- Accepts values like odd, even, or specific numbers (e.g., 2n+1).
- Useful for styling alternating rows in tables or lists.
- Example: I used nth-child to apply alternate background colors to table rows.

```
tr:nth-child(even) {
  background-color: #f2f2f2;
}
```

56. What is the purpose of the position: sticky property?

- Keeps an element fixed within a container until a specific scroll position is reached.
- Combines relative and fixed positioning.
- Requires top, right, bottom, or left to define the sticky boundary.
- Example: I used position: sticky for table headers in a long-scroll table.

```
th {
  position: sticky;
  top: 0;
  background: white;
}
```

57. How does the white-space property work in CSS?

- Controls how text is handled within an element.
- Common values: normal (default), nowrap, pre, and break-spaces.
- Helps manage line breaks and spacing.
- Example: I used white-space: nowrap to prevent wrapping in navigation menus.

```
.nav-item {
  white-space: nowrap;
}
```

58. What is the difference between transform and transition in CSS?

- transform: Applies transformations like rotation, scaling, or translation to elements.
- transition: Defines the timing for changes between states.
- Often used together for smooth animated effects.

- Example: I used transform and transition to create hover animations on icons.

```
.icon {
  transform: scale(1);
  transition: transform 0.3s;
}
.icon:hover {
  transform: scale(1.2);
}
```

59. How does the background-size property work in CSS?

- Defines how the background image is sized within its container.
- Common values: auto, cover, contain, and specific dimensions.
- Ensures proper scaling without distortion.
- Example: I used background-size: cover for full-page hero images.

```
.hero {
  background-image: url('hero.jpg');
  background-size: cover;
}
```

60. What is the difference between border and outline in CSS?

- border: Affects the layout and can change the size of the element box.
- outline: Does not affect layout; it's drawn outside the border box.
- Outlines are often used for focus indication and accessibility.
- Example: I used outline to highlight form fields during validation errors.

```
input:focus {
  outline: 2px solid blue;
}
```

JavaScript Interview Questions

61. What is the difference between var, let, and const in JavaScript?

- var: Function-scoped, can be redeclared, and hoisted but not block-scoped.
- let: Block-scoped, cannot be redeclared in the same scope, and not hoisted.
- const: Block-scoped, immutable reference, but the value can still be mutable if it's an object.
- Example: I used const for API keys and let for loop counters in a data-processing app.

```
const API_KEY = "abc123";
let counter = 0;
```

62. What are JavaScript closures?

- A closure is a function that retains access to its lexical scope even when executed outside it.
- Useful for data encapsulation and private variables.
- Created automatically when a function is defined inside another function.
- Example: I used closures to create reusable utility functions in a project.

```
function createCounter() {
  let count = 0;
  return function () {
    count++;
    return count;
  };
}
const counter = createCounter();
console.log(counter()); // Output: 1
```

63. What is the difference between == and === in JavaScript?

- == performs type coercion, converting operands to the same type before comparison.
- === checks for strict equality without type conversion.
- Using === avoids unexpected results due to type coercion.
- Example: I used === to validate user inputs in a form validation script.

```
console.log(1 == "1"); // true
console.log(1 === "1"); // false
```

64. How does the this keyword work in JavaScript?

- Refers to the context in which the function is executed.
- In global scope, it refers to window in browsers or global in Node.js.
- In methods, it refers to the object the method belongs to.
- Example: I used this in a class to refer to instance properties and methods.

```
class User {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, ${this.name}`);
  }
}
const user = new User("Alice");
user.greet(); // Output: Hello, Alice
```

65. What is event delegation in JavaScript?

- A technique where a parent element handles events for its child elements.
- Reduces memory usage by attaching fewer event listeners.
- Uses the target property to identify the clicked element.
- Example: I used event delegation to manage a dynamic list of tasks in a to-do app.

```
document.querySelector(".task-list").addEventListener("click", (e) => {
  if (e.target.tagName === "LI") {
    e.target.classList.toggle("completed");
  }
});
```

66. What is the difference between synchronous and asynchronous JavaScript?

- Synchronous: Tasks are executed one after another, blocking the main thread.
- Asynchronous: Tasks are executed without waiting for other tasks to complete, using callbacks or promises.
- Async methods prevent UI freezing during long-running tasks.
- Example: I used asynchronous APIs to fetch data without blocking page rendering.

```
fetch("https://api.example.com/data")
  .then((response) => response.json())
  .then((data) => console.log(data));
```

67. How do call(), apply(), and bind() work in JavaScript?

- call(): Invokes a function with a specific this value and arguments as a list.
- apply(): Similar to call() but accepts arguments as an array.
- bind(): Returns a new function with this bound to the specified context.
- Example: I used bind() to pass a method as a callback with the correct this context.

```
const obj = { name: "Alice" };
function greet() {
  console.log(`Hello, ${this.name}`);
}
const boundGreet = greet.bind(obj);
boundGreet(); // Output: Hello, Alice
```

68. What is the purpose of Promises in JavaScript?

- Promises handle asynchronous operations and avoid callback hell.
- Has three states: pending, resolved, and rejected.

- Methods like then(), catch(), and finally() handle promise outcomes.
- Example: I used Promises for sequential API calls in a weather app.

```
fetch("https://api.example.com/weather")
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

69. What is the difference between let and var in loops?

- let is block-scoped, so each iteration gets a new instance.
- var is function-scoped, so the value persists across iterations.
- Using let prevents issues in closures inside loops.
- Example: I used let in a loop to ensure proper scoping for event listeners.

```
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000); // Outputs 0, 1, 2
}
```

70. What is the purpose of the async and await keywords in JavaScript?

- Simplify writing asynchronous code, making it look synchronous.
- async marks a function to return a Promise.
- await pauses execution until the Promise resolves or rejects.
- Example: I used async/await to handle API responses in a cleaner way.

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
fetchData();
```

71. What are JavaScript modules, and how are they used?

- Modules allow code to be split into reusable and maintainable pieces.
- Export functionality using export and import it in other files using import.
- Supported natively in modern browsers and Node.js.
- Example: I used modules to separate utility functions in a large-scale app.

```
// math.js
export function add(a, b) {
  return a + b;
}
// app.js
import { add } from "./math.js";
console.log(add(2, 3)); // Output: 5
```

72. How does JavaScript handle data types?

- JavaScript has dynamic typing, so variables can hold any data type.
- Includes six primitive types (e.g., string, number) and objects.
- Type conversion occurs automatically or can be enforced.
- Example: I ensured type validation for user inputs in a form.

```
const age = "30";
console.log(Number(age) + 5); // Output: 35
```

73. What is the difference between null and undefined in JavaScript?

- null is explicitly assigned to indicate the absence of value.
- undefined means a variable is declared but not assigned a value.
- Both are falsy values but represent different states.
- Example: I used null to reset an object property after deletion.

```
let obj = { name: "Alice" };
obj.name = null;
console.log(obj.name); // Output: null
```

74. What are arrow functions, and how do they differ from regular functions?

- Arrow functions have a concise syntax and do not have their own this.
- Cannot be used as constructors or with new.
- Useful for callbacks and anonymous functions.
- Example: I used arrow functions for event listeners in a React project.

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // Output: 5
```

75. What is the purpose of the reduce() method in JavaScript?

- Executes a reducer function on an array to accumulate a single output.
- Takes two arguments: an accumulator and the current value.
- Commonly used for aggregating data.
- Example: I used reduce() to calculate the total price in a shopping cart.

```
const prices = [10, 20, 30];
const total = prices.reduce((acc, price) => acc + price, 0);
console.log(total); // Output: 60
```

JavaScript Interview Questions (Continued)

76. What is the difference between map(), filter(), and forEach() in JavaScript?

- map(): Returns a new array by applying a function to each element of the original array.
- filter(): Returns a new array with elements that satisfy a given condition.
- forEach(): Executes a function for each element but does not return a new array.
- Example: I used map() to transform data, filter() to extract matching records, and forEach() to log each item during debugging.

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2); // [2, 4, 6, 8]
const evens = numbers.filter(num => num % 2 === 0); // [2, 4]
numbers.forEach(num => console.log(num)); // Logs 1, 2, 3, 4
```

77. What are JavaScript generators, and how are they used?

- Generators are functions that can be paused and resumed, defined using function*.
- Use yield to pause execution and produce values.
- Useful for iterating over sequences or managing asynchronous workflows.
- Example: I used generators to handle paginated API responses lazily.

```
function* numbers() {
  yield 1;
  yield 2;
  yield 3;
}
const gen = numbers();
console.log(gen.next().value); // Output: 1
console.log(gen.next().value); // Output: 2
```

78. How does the setTimeout() function work in JavaScript?

- Executes a callback function after a specified delay in milliseconds.
- Does not block other code execution.
- Returns a timeout ID that can be used to clear the timeout.
- Example: I used setTimeout() to show a delayed welcome message on page load.

```
setTimeout(() => {
  console.log("Welcome!");
}, 2000); // Logs "Welcome!" after 2 seconds
```

79. What is the difference between setTimeout() and setInterval()?

- setTimeout(): Executes a function once after a specified delay.

- setInterval(): Executes a function repeatedly at specified intervals.
- Both return IDs for managing or clearing timers.
- Example: I used setInterval() to update a digital clock every second.

```
setInterval(() => {
  const now = new Date();
  console.log(now.toLocaleTimeString());
}, 1000);
```

80. How does the typeof operator work in JavaScript?

- Returns the type of a given operand as a string.
- Works for primitive types and objects but has some quirks (e.g., typeof null returns "object").
- Useful for type checking in dynamic environments.
- Example: I used typeof to validate API response data types.

```
console.log(typeof 42); // Output: "number"
console.log(typeof "Hello"); // Output: "string"
console.log(typeof null); // Output: "object"
```

81. What is the difference between Object.keys(), Object.values(), and Object.entries()?

- Object.keys(): Returns an array of an object's property names.
- Object.values(): Returns an array of an object's property values.
- Object.entries(): Returns an array of key-value pairs.
- Example: I used Object.entries() to iterate over an object in a dashboard app.

```
const user = { name: "Alice", age: 25 };
console.log(Object.keys(user)); // ["name", "age"]
console.log(Object.values(user)); // ["Alice", 25]
console.log(Object.entries(user)); // [["name", "Alice"], ["age", 25]]
```

82. What is the event loop in JavaScript?

- The mechanism that handles asynchronous operations in JavaScript.
- Uses a queue to process events and execute callbacks.
- Ensures non-blocking behavior in single-threaded environments.
- Example: I debugged an issue where setTimeout() callbacks were delayed due to a blocking loop.

```
setTimeout(() => console.log("First"), 0);
console.log("Second"); // Logs "Second" first, then "First"
```

83. How do you deep copy an object in JavaScript?

- Use JSON.parse(JSON.stringify(object)) for simple objects without methods.
- For complex objects, use libraries like Lodash or custom recursive functions.
- Deep copies create entirely new references, unlike shallow copies.
- Example: I used deep copying to duplicate nested configuration objects safely.

```
const obj = { a: 1, b: { c: 2 } };
const deepCopy = JSON.parse(JSON.stringify(obj));
console.log(deepCopy.b === obj.b); // Output: false
```

84. What are JavaScript WeakMaps and WeakSets?

- WeakMaps: Collections of key-value pairs where keys are weakly referenced objects.
- WeakSets: Collections of unique weakly referenced objects.
- Prevent memory leaks as objects are garbage-collected when no longer referenced.
- Example: I used WeakMaps to store metadata for DOM elements dynamically.

```
const weakMap = new WeakMap();
const obj = {};
weakMap.set(obj, "value");
console.log(weakMap.get(obj)); // Output: "value"
```

85. How do try, catch, and finally blocks work in JavaScript?

- try: Contains code that may throw an error.
- catch: Handles the error and prevents application crashes.
- finally: Executes code regardless of whether an error occurred.
- Example: I used a try-catch block to handle API request errors gracefully.

```
try {
  const data = JSON.parse('{"key": "value"}');
  console.log(data);
} catch (error) {
  console.error("Invalid JSON", error);
} finally {
  console.log("Parsing attempt completed.");
}
```

86. What is a JavaScript Proxy?

- An object that wraps another object and intercepts operations like property access and assignment.
- Provides custom behavior for fundamental operations using traps.
- Useful for validation, logging, or mocking.
- Example: I used a Proxy to validate object property updates in a settings manager.

```
const settings = {
  theme: "dark",
};
const proxy = new Proxy(settings, {
  set(target, prop, value) {
    if (prop === "theme" && !["dark", "light"].includes(value)) {
      throw new Error("Invalid theme");
    }
    target[prop] = value;
    return true;
  },
});
proxy.theme = "light"; // Works
// proxy.theme = "blue"; // Throws an error
```

87. What are template literals in JavaScript?

- Allow string interpolation using backticks () and placeholders ${expression}`.
- Support multi-line strings without escape characters.
- Simplify concatenation and dynamic string creation.
- Example: I used template literals to dynamically create HTML content in a component.

```
const name = "Alice";
const greeting = `Hello, ${name}!`;
console.log(greeting); // Output: "Hello, Alice!"
```

88. How does destructuring work in JavaScript?

- Extracts values from arrays or properties from objects into variables.
- Supports default values for missing properties.
- Simplifies code for accessing nested data.
- Example: I used destructuring to extract specific fields from API responses.

```
const user = { name: "Alice", age: 25 };
const { name, age } = user;
console.log(name, age); // Output: "Alice", 25
```

89. What are JavaScript promises chaining and error handling?

- Chaining allows sequential execution of multiple then() calls.
- Errors can be caught and handled with catch() in any step.
- Ensures clean and manageable asynchronous code.
- Example: I used promise chaining to fetch user data, then their posts.

```
fetch("/user")
  .then((response) => response.json())
  .then((user) => fetch(`/posts?userId=${user.id}`))
  .then((response) => response.json())
  .then((posts) => console.log(posts))
  .catch((error) => console.error(error));
```

90. How do you handle exceptions in asynchronous code?

- Use try-catch blocks inside async functions.
- Attach .catch() to promises to handle errors.
- Prevents unhandled rejections that can crash the application.
- Example: I used error handling in async/await functions to manage API failures.

```
async function fetchData() {
  try {
    const response = await fetch("/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}
fetchData();
```

1. Explain what is LINQ? Why is it required?

- LINQ (Language-Integrated Query) is a querying syntax in .NET that allows querying collections like arrays, lists, and databases.
- It provides a unified approach to querying different data sources using a single syntax.
- LINQ improves code readability and reduces the need for complex loops and conditional logic.
- Example: var evenNumbers = numbers.Where(n => n % 2 == 0).ToList(); This retrieves even numbers from a list in a clean, readable way.

2. What are the types of LINQ?

- LINQ to Objects: Queries in-memory collections such as arrays or lists.
- LINQ to SQL: Queries SQL Server databases and translates LINQ queries to SQL.
- LINQ to XML: Queries and manipulates XML data.
- Example: XDocument xmlDoc = XDocument.Load("data.xml"); var items = from item in xmlDoc.Descendants("Item") select item;

3. What are Extension Methods in LINQ?

- Extension methods allow adding new methods to existing types without modifying them.
- LINQ uses extension methods like Where, Select, and OrderBy to work with collections.
- They are defined as static methods but invoked like instance methods.
- Example: public static int Square(this int number) { return number * number; } int result = 5.Square();

4. What is an Anonymous function in LINQ?

- Anonymous functions are functions without a name, often used for short-lived operations.
- In LINQ, they are typically defined using lambda expressions.
- They help create concise and inline methods for querying data.
- Example: var oddNumbers = numbers.Where(n => n % 2 != 0).ToList(); uses a lambda as an anonymous function.

5. Explain the purpose of LINQ providers in LINQ?

- LINQ providers translate LINQ queries into the appropriate format for the data source.
- They allow LINQ to work with different sources like SQL, XML, or in-memory objects.
- Providers implement the IQueryable<T> interface to enable query execution.
- Example: LINQ to SQL translates LINQ queries into SQL commands for execution on a database.

6. List out the three main components of LINQ.

- Standard Query Operators: Methods like Where, Select, and GroupBy to query collections.
- LINQ Providers: Translate LINQ queries to the target data source's format.
- Query Syntax and Method Syntax: Two ways to write LINQ queries (SQL-like and method chaining).
- Example: Query syntax: from n in numbers where n > 10 select n; Method syntax: numbers.Where(n => n > 10);

7. Explain how LINQ is more useful than Stored Procedures.

- LINQ provides compile-time syntax checking, reducing runtime errors.
- It integrates seamlessly with C#, improving code maintainability.
- LINQ supports dynamic queries, unlike stored procedures that need pre-definition.
- Example: Dynamic query with LINQ: var results = data.Where(d => d.Name.Contains(searchText)).ToList();

8. Explain why SELECT clause comes after FROM clause in LINQ.

- The FROM clause defines the data source, which is essential before filtering or projecting results.
- This order mirrors how method syntax operates (.Select() follows .Where()).
- It enhances code readability and follows logical data access patterns.
- Example: var results = from n in numbers where n > 0 select n;

9. In LINQ, how will you find the index of an element using Where() with Lambda Expressions?

- LINQ's Select method can project indexes along with elements.
- You can filter elements based on a condition and use indexing.
- This is useful for retrieving the positions of items in a collection.
- Example: var indexedItems = numbers.Select((num, index) => new { num, index }).Where(x => x.num == target).Select(x => x.index);

10. Mention what is the role of DataContext classes in LINQ.

- DataContext acts as a bridge between LINQ and the database, tracking changes and submitting updates.
- It provides a strongly-typed interface to interact with database tables.
- It simplifies CRUD operations by mapping tables to objects.
- Example: DataContext db = new DataContext(); var customers = db.GetTable<Customer>().ToList();

11. What are Anonymous Types in LINQ?

- Anonymous types allow creating objects without defining a class.
- They are useful for projecting data into custom shapes in LINQ.
- They provide a quick way to hold data from multiple fields.
- Example: var result = from p in products select new { p.Name, p.Price };

12. Explain what is LINQ to Objects?

- LINQ to Objects allows querying in-memory collections like arrays and lists.
- It does not require a database or external data source.
- It enables filtering, sorting, and transforming collections with ease.
- Example: var oddNumbers = numbers.Where(n => n % 2 != 0).ToList();

13. What is LINQ in C#?

- LINQ is a feature in C# that allows querying data from different sources using a consistent syntax.
- It simplifies complex data manipulation tasks.
- It provides a readable and maintainable alternative to traditional loops and conditions.
- Example: var result = from item in collection where item.Age > 18 select item.Name;

14. When deciding between using Entity Framework and LINQ to SQL as an ORM, what's the difference?

- Entity Framework supports multiple databases, while LINQ to SQL is limited to SQL Server.
- EF provides more advanced features like lazy loading and better mapping capabilities.
- EF is more suitable for complex applications, while LINQ to SQL is simpler.
- Example: DbContext context = new DbContext(); var users = context.Users.ToList();

15. Explain what are compiled queries in LINQ?

- Compiled queries are pre-compiled and stored for reuse, improving performance.
- They are especially beneficial for queries executed multiple times.
- They reduce the overhead of query parsing and translation.
- Example: var compiledQuery = CompiledQuery.Compile((DataContext db) => db.Customers.Where(c => c.City == "London"));

16. What is Expression Trees and how are they used in LINQ?

- Expression trees represent code as a data structure, allowing dynamic modification and execution.
- LINQ uses them to translate queries into SQL or other formats for execution.
- They enable advanced scenarios like building dynamic queries at runtime.
- Example: Expression<Func<int, bool>> expr = num => num > 5; dynamically builds a condition for filtering numbers.

17. Explain what are Lambda Expressions in LINQ?

- Lambda expressions are anonymous functions used to create delegates or expression tree types.
- They simplify writing inline functions in LINQ queries.
- Syntax: parameters => expression.
- Example: var evenNumbers = numbers.Where(n => n % 2 == 0).ToList(); uses a lambda to filter even numbers.

18. Could you compare Entity Framework vs LINQ to SQL vs ADO.NET vs stored procedures?

- **Entity Framework**: ORM supporting multiple databases, rich features, and lazy loading.
- **LINQ to SQL**: Lightweight ORM, SQL Server-specific, easier setup.
- **ADO.NET**: Manual data access using SQL queries or stored procedures, high control.
- **Stored Procedures**: Precompiled SQL, best for complex batch operations.
- Example: EF query: var data = context.Users.Where(u => u.IsActive).ToList();

19. What is the difference between First() and Take(1) in LINQ?

- First() returns the first matching element and throws an exception if none found.
- Take(1) returns a collection with one element or an empty collection if none found.
- Use First() when exactly one element is expected, and Take(1) for safe enumeration.
- Example: var result = numbers.Take(1).ToList(); retrieves the first element safely.

20. Explain the difference between Skip() and SkipWhile() extension methods in LINQ.

- Skip() skips a specified number of elements.
- SkipWhile() skips elements based on a condition until it fails.
- SkipWhile() is useful when skipping based on dynamic criteria.
- Example: var result = numbers.SkipWhile(n => n < 10).ToList(); skips numbers less than 10.

21. Define what is let clause in LINQ?

- The let clause allows defining temporary variables in a query.
- It improves readability by storing intermediate results.
- It is useful when the same calculation is needed multiple times.
- Example: var query = from n in numbers let square = n * n where square > 10 select square;

22. Explain how standard query operators are useful in LINQ.

- Standard query operators are predefined methods like Where, Select, and GroupBy.
- They provide a consistent way to filter, transform, and aggregate data.
- They work across different data sources, enhancing code portability.
- Example: var result = numbers.Where(n => n % 2 == 0).OrderBy(n => n).ToList();

23. When to use First() and when to use FirstOrDefault() in LINQ?

- Use First() when you are sure the collection contains elements.
- Use FirstOrDefault() to avoid exceptions if no elements are found, returning a default value.
- It is safer to use FirstOrDefault() in scenarios with unknown data.
- Example: var firstEven = numbers.FirstOrDefault(n => n % 2 == 0); safely retrieves the first even number.

24. Could you explain the difference between deferred execution and lazy evaluation in C#?

- **Deferred execution**: Queries are not executed until enumerated (e.g., using ToList()).
- **Lazy evaluation**: Objects are not created or populated until needed.
- Both improve performance by avoiding unnecessary computations.
- Example: var query = numbers.Where(n => n > 10); only executes when iterated.

25. What is an equivalent to the let keyword in chained LINQ extension method calls?

- The equivalent is using the Select method to project intermediate results.
- It allows temporary variables within a method chain.
- This improves clarity in complex queries.
- Example: var result = numbers.Select(n => new { n, Square = n * n }).Where(x => x.Square > 10);

26. Explain the difference between Select() and Where() in LINQ.

- Select() projects each element into a new form.
- Where() filters elements based on a condition.
- Use Select() for transformation and Where() for filtering.
- Example: var squares = numbers.Select(n => n * n).ToList();

27. Name some advantages of LINQ over Stored Procedures.

- LINQ provides compile-time checks and type safety.
- It reduces development time with readable, maintainable code.
- LINQ is integrated with C# and allows dynamic queries.
- Example: LINQ query: var results = dbContext.Users.Where(u => u.IsActive).ToList();

28. When should I use a Compiled Query in LINQ?

- Use compiled queries when running the same query multiple times with different parameters.
- They reduce the cost of query parsing and translation.
- Ideal for performance-critical applications.
- Example: var compiled = CompiledQuery.Compile((db, id) => db.Users.Where(u => u.Id == id));

29. What are the benefits of Deferred Execution in LINQ?

- Deferred execution improves performance by executing queries only when needed.
- It allows chaining multiple operations efficiently.
- It supports dynamic query modification before execution.
- Example: var query = numbers.Where(n => n > 10); does not execute until iterated.

30. Name some disadvantages of LINQ over Stored Procedures.

- LINQ queries may have performance overhead compared to optimized stored procedures.
- Complex queries might be harder to translate into LINQ.
- It depends on the ORM's translation capabilities.
- Example: In complex reporting, stored procedures often outperform LINQ queries. **31. What is the difference between Select() and SelectMany() in LINQ?**
- Select() projects each element into a new form, maintaining the structure of the collection.
- SelectMany() flattens nested collections into a single collection.
- Use Select() for simple projections and SelectMany() for handling nested collections.
- Example: var flatList = customers.SelectMany(c => c.Orders).ToList(); combines all orders from multiple customers into one list.

32. Why use AsEnumerable() rather than casting to IEnumerable in LINQ?

- AsEnumerable() forces LINQ to use in-memory processing instead of query translation.
- It is useful to switch from database evaluation to in-memory evaluation.
- It avoids exceptions from unsupported operations in LINQ to SQL or LINQ to Entities.
- Example: var localQuery = dbContext.Users.AsEnumerable().OrderBy(u => u.Name); performs ordering in memory.

33. What is the difference between returning IQueryable vs. IEnumerable in LINQ?

- IQueryable<T> allows query composition and deferred execution on the data source.
- IEnumerable<T> represents in-memory collections and supports immediate execution.
- Use IQueryable<T> for database queries and IEnumerable<T> for in-memory collections.
- Example: public IQueryable<User> GetActiveUsers() => db.Users.Where(u => u.IsActive); allows further query composition.

34. Can you provide a concise distinction between anonymous methods and lambda expressions in LINQ?

- Anonymous methods and lambda expressions both define inline functions.
- Lambda expressions are more concise and support expression trees.
- Anonymous methods use the delegate keyword and are more verbose.
- Example: Func<int, int> square = x => x * x; is a lambda, while delegate(int x) { return x * x; } is an anonymous method.

35. Filter out the first 3 even numbers from a list using LINQ.

- Use Where() to filter even numbers and Take(3) to select the first three.
- The combination allows efficient querying.
- It demonstrates deferred execution until enumeration.
- Example: var firstThreeEvens = numbers.Where(n => n % 2 == 0).Take(3).ToList();

36. Get the indexes of top items where item value = true using LINQ.

- Use Select with indexing to project indexes and values.
- Filter with Where to get only true items.
- Extract indexes using another Select.
- Example: var indexes = items.Select((value, index) => new { value, index }).Where(x => x.value).Select(x => x.index).ToList();

37. Using LINQ to remove elements from a List.

- LINQ queries are immutable, so filtering creates a new list.
- Use Where to exclude elements and reassign the result.
- For actual list modification, combine LINQ with RemoveAll.
- Example: numbers = numbers.Where(n => n % 2 != 0).ToList(); removes even numbers from the list.

38. Explain the importance of the GroupBy operator in LINQ.

- GroupBy groups elements based on a key, returning a collection of groups.
- It is useful for aggregation and categorization.
- Groups can be further processed with Select or SelectMany.
- Example: var groups = numbers.GroupBy(n => n % 2).Select(g => new { Key = g.Key, Count = g.Count() });

39. What is the difference between ToList() and ToArray() in LINQ?

- ToList() converts a sequence to a List<T>, supporting dynamic resizing.
- ToArray() converts a sequence to an array with fixed size.
- Use ToList() when frequent additions/removals are needed.
- Example: var list = numbers.Where(n => n > 10).ToList(); var array = numbers.Where(n => n > 10).ToArray();

40. Explain the difference between Aggregate() and Reduce() in LINQ.

- Aggregate() applies a function cumulatively to the sequence, resulting in a single value.
- It is useful for custom aggregations beyond simple sums or counts.
- LINQ does not have a direct Reduce() method but Aggregate() serves a similar purpose.
- Example: var product = numbers.Aggregate((acc, n) => acc * n); calculates the product of all numbers.

41. What is the purpose of the Join operator in LINQ?

- Join combines elements from two collections based on a key.
- It supports inner joins, yielding matching elements.
- It simplifies complex join logic compared to SQL.
- Example: var result = customers.Join(orders, c => c.Id, o => o.CustomerId, (c, o) => new { c.Name, o.OrderId });

42. When to use Single() vs. SingleOrDefault() in LINQ?

- Single() expects exactly one matching element and throws an exception otherwise.
- SingleOrDefault() returns the default value if no elements match, avoiding exceptions.
- Use Single() for unique constraints and SingleOrDefault() for optional results.
- Example: var user = users.SingleOrDefault(u => u.Id == userId);

43. Explain the difference between Distinct() and Union() in LINQ.

- Distinct() removes duplicate elements from a single collection.
- Union() combines two collections and removes duplicates.
- Use Distinct() for deduplication and Union() for set operations.
- Example: var uniqueNumbers = numbers.Distinct().ToList(); var combined = list1.Union(list2).ToList();

44. How to perform a left join using LINQ?

- Use GroupJoin or a combination of DefaultIfEmpty and SelectMany.
- It ensures all elements from the left collection are included.
- It handles nulls for non-matching right elements.

- Example: var result = from c in customers join o in orders on c.Id equals o.CustomerId into gj from suborder in gj.DefaultIfEmpty() select new { c.Name, suborder?.OrderId };

45. Explain the role of the Select keyword in LINQ queries.

- Select projects elements into a new form or shape.
- It can transform data by selecting specific properties or creating anonymous types.
- It is equivalent to the projection step in SQL.
- Example: var names = users.Select(u => u.Name).ToList();

1. What is inheritance?

- Inheritance is a feature of OOP that allows a class (child) to acquire properties and behaviors of another class (parent).
- It helps in code reusability by enabling child classes to use existing parent class code.
- Supports hierarchical relationships where a base class can have multiple derived classes.
- Provides a way to override parent class methods for specific behaviors in the child class.
- I used inheritance in my project to create a base service class for HTTP requests, which derived classes customized for specific APIs:

```
// Base Class
public class ApiService {
   public virtual void MakeRequest() {
      Console.WriteLine("Making API request");
   }
}

// Derived Class
public class UserService : ApiService {
   public override void MakeRequest() {
      Console.WriteLine("Making User API request");
   }
}
```

2. What is OOP?

- OOP (Object-Oriented Programming) is a programming paradigm that models concepts as "objects" with data and behavior.
- It emphasizes modularity and reusability through encapsulation, inheritance, and polymorphism.
- Encourages abstraction by hiding implementation details and exposing only relevant functionality.
- Promotes easy maintenance and scalability for software development.
- In one project, I structured an e-commerce application using OOP principles, such as encapsulating product data and providing services to handle shopping cart operations.

3. Why is the virtual keyword used in code?

- The virtual keyword allows a method, property, or event to be overridden in derived classes.
- Ensures runtime polymorphism, enabling dynamic method binding based on the object type.
- Helps in creating extensible systems by allowing behavior modifications in inherited classes.
- Provides flexibility while maintaining a contract with the base class method signatures.
- I used the virtual keyword in a .NET Core project to allow customizable logging behavior in derived logging services:

```
public class Logger {
   public virtual void Log(string message) {
      Console.WriteLine($"Log: {message}");
   }
}
public class FileLogger : Logger {
   public override void Log(string message) {
      System.IO.File.WriteAllText("log.txt", message);
   }
}
```

4. Difference between procedural and OOP?

- Procedural programming is task-oriented, focusing on functions and logic, while OOP is object-oriented, focusing on objects and their interactions.
- Procedural uses a top-down approach, whereas OOP uses a bottom-up approach.
- Data and functions are separate in procedural programming, but in OOP, they are encapsulated together.
- Procedural programming lacks features like inheritance and polymorphism, which are central to OOP.
- In a real-world example, I used procedural programming for simple scripts, like batch file processing, and OOP for larger applications like CRM systems for scalability and maintainability.

5. What is a class?

- A class is a blueprint for creating objects, defining data (fields) and behaviors (methods).
- It encapsulates data for an object and provides methods for accessing and modifying it.
- Classes enable abstraction by hiding complex implementation details from users.
- They support OOP principles like inheritance and polymorphism.
- In a project, I created a Product class to encapsulate product details for an inventory management system:

```
public class Product {
   public string Name { get; set; }
   public decimal Price { get; set; }

   public void DisplayInfo() {
      Console.WriteLine($"Product: {Name}, Price: {Price}");
   }
}
```

6. Basic features of OOPs?

- Encapsulation: Bundles data and methods to restrict direct access to object components.
- Inheritance: Allows a class to use properties and methods of another class.
- Polymorphism: Enables objects to take on multiple forms based on context.
- Abstraction: Hides implementation details, exposing only essential features.
- I applied all these features while creating a hierarchical system for different types of reports in an analytics tool, ensuring extensibility and reusability.

7. Can you inherit private members of a class?

- Private members of a class cannot be directly inherited by derived classes.
- Derived classes can only access public, protected, or internal members of the base class.
- To indirectly use private members, the base class provides access through public or protected methods.
- This ensures encapsulation, limiting access to sensitive data.
- I followed this approach in a project by exposing private member data via protected methods for derived classes:

```
public class BaseClass {
   private string secret = "Hidden";

   protected string GetSecret() {
      return secret;
   }
}

public class DerivedClass : BaseClass {
   public void DisplaySecret() {
      Console.WriteLine(GetSecret());
   }
}
```

8. What is the difference between a class and a structure?

- Classes are reference types, while structures are value types.
- Classes support inheritance, but structures do not.

- Structures are stored on the stack, while objects of classes are stored on the heap.
- Example: I used a structure for a Point type in a graphics application for efficiency and a class for Canvas to manage complex operations.

```
struct Point {
   public int X { get; set; }
   public int Y { get; set; }
}

class Canvas {
   public Point Origin { get; set; }
}
```

9. What is the relationship between a class and an object?

- A class is a blueprint or template, while an object is an instance of a class.
- Classes define the structure and behavior, while objects store actual data and execute the behavior.
- A class can create multiple objects, each with unique states.
- Example: I designed a User class in a social media app, and objects represented individual users.

```
class User {
   public string Name { get; set; }
}

User user1 = new User { Name = "Alice" };
User user2 = new User { Name = "Bob" };
```

10. What is an object bevel?

- The term "object bevel" seems unclear in the OOP context and may be misinterpreted.
- Objects in OOP represent real-world entities with properties and methods.
- It's important to clarify and use accurate terminology when discussing OOP concepts.
- Example: If you meant "object behavior," I implemented an object's behavior like methods in a game where Player objects have Jump actions.

```
class Player {
   public void Jump() => Console.WriteLine("Player jumped");
}
Player player = new Player();
player.Jump();
```

11. Explain the concept of a constructor.

- A constructor is a special method used to initialize an object when it is created.
- It shares the same name as the class and has no return type.
- Constructors can be parameterized or default, enabling object initialization flexibility.
- Example: I used a constructor in a Product class to set initial values for name and price in an e-commerce app.

```
class Product {
   public string Name { get; }
   public decimal Price { get; }

   public Product(string name, decimal price) {
      Name = name;
      Price = price;
   }
}

Product product = new Product("Laptop", 1200.00m);
```

12. What is Encapsulation?

- Encapsulation is the practice of bundling data and methods into a single unit (class).
- It restricts direct access to some components and protects the object's integrity.
- Access modifiers like private, public, and protected enforce encapsulation.
- Example: In a banking app, I encapsulated account details to prevent unauthorized access and used getters/setters for controlled access.

```
class BankAccount {
   private decimal balance;
   public decimal GetBalance() => balance;
   public void Deposit(decimal amount) => balance += amount;
}
```

13. What is Polymorphism?

- Polymorphism allows methods to take multiple forms (e.g., overriding, overloading).
- It enables objects to interact through a common interface, promoting flexibility.
- There are two types: compile-time (overloading) and runtime (overriding).
- Example: In a zoo app, polymorphism let me create an Animal base class and override a Speak method in derived classes like Dog and Cat.

```
class Animal {
   public virtual void Speak() => Console.WriteLine("Animal speaks");
}

class Dog : Animal {
   public override void Speak() => Console.WriteLine("Woof");
}
Animal myDog = new Dog();
myDog.Speak();
```

14. How could you define Abstraction in OOP?

- Abstraction hides the implementation details and exposes only the necessary functionalities.
- It simplifies complex systems by reducing the visible complexity to the user.
- Abstract classes and interfaces are used to achieve abstraction.
- Example: In a vehicle system, I used an abstract Vehicle class to define common methods like Start and Stop without showing internal engine details.

```
abstract class Vehicle {
   public abstract void Start();
   public abstract void Stop();
}

class Car : Vehicle {
   public override void Start() => Console.WriteLine("Car started");
   public override void Stop() => Console.WriteLine("Car stopped");
}
Vehicle myCar = new Car();
myCar.Start();
```

15. How can you prevent your class from being inherited further?

- Use the sealed keyword to prevent a class from being inherited.
- Sealing classes ensures no modifications or extensions can be made via inheritance.
- It can be helpful when designing security-critical or immutable classes.
- Example: I used a sealed class for LicenseKeyManager to ensure no one could alter its behavior.

```
sealed class LicenseKeyManager {
   public string GenerateKey() => "ABC123";
}

// Attempting to inherit this class will result in a compilation error.
```

16. What do you mean by Data Encapsulation?

- Data encapsulation is the bundling of data and methods that operate on the data into a single unit (class).
- It restricts direct access to an object's data, providing controlled access through methods or properties.
- This ensures that an object's internal state is protected and can only be modified in intended ways.
- Example: In a payroll system, I encapsulated employee details to ensure only authorized methods could modify salary information.

```
class Employee {
   private decimal salary;

   public decimal GetSalary() => salary;
   public void SetSalary(decimal value) {
      if (value > 0) salary = value;
   }
}
```

17. What's the difference between a method and a function in OOP context?

- In OOP, a method is a function that is defined inside a class and operates on its objects.
- Functions are independent and can exist outside of a class, while methods are inherently tied to their class or object.
- Methods typically operate on instance data or class-level data, while functions are general-purpose.
- Example: In a game, I used a CalculateScore method tied to the Player class instead of a standalone function for better modularity.

```
class Player {
   public int Score { get; set; }
   public void CalculateScore(int points) => Score += points;
}
```

18. Can you specify the accessibility modifier for methods inside an interface?

- Methods in an interface are implicitly public and cannot have any other accessibility modifier.
- This ensures that all implementing classes provide a public implementation of the methods.
- Starting from C# 8.0, default interface methods allow implementation within interfaces but are still public by default.
- Example: I created an ILogger interface with public methods to enforce consistent logging behavior across classes.

```
interface ILogger {
   void Log(string message);
}

class ConsoleLogger : ILogger {
   public void Log(string message) => Console.WriteLine(message);
}
```

19. Is it possible for a class to inherit the constructor of its base class?

- Constructors are not inherited in C#. However, a derived class can call the base class constructor using base.
- This ensures proper initialization of the base class when a derived class is instantiated.
- Overriding constructors is done by explicitly defining them in the derived class.
- Example: In a banking app, I used the base class constructor to initialize account properties in derived classes.

```
class Account {
   public Account(string accountHolder) => Console.WriteLine($"Account
created for {accountHolder}");
}

class SavingsAccount : Account {
   public SavingsAccount(string accountHolder) : base(accountHolder) { }
}
SavingsAccount sa = new SavingsAccount("John");
```

20. What are the similarities between a class and a structure?

- Both classes and structures can contain methods, properties, fields, and constructors.
- Both can implement interfaces to enforce certain behaviors.
- Both support access modifiers for encapsulating data.
- Example: I used a structure for Point and a class for Shape in a graphics tool, as both shared similar members like properties for coordinates.

```
struct Point {
   public int X { get; set; }
   public int Y { get; set; }
}
class Shape {
   public string Name { get; set; }
}
```

21. What are the different ways a method can be overloaded?

- By changing the number of parameters.
- By altering the type of parameters.
- By changing the order of parameters (only if types differ).
- Example: I implemented overloaded methods in a calculator class to handle different data types like integers and doubles.

```
class Calculator {
   public int Add(int a, int b) => a + b;
   public double Add(double a, double b) => a + b;
}
Calculator calc = new Calculator();
Console.WriteLine(calc.Add(5, 10));
Console.WriteLine(calc.Add(5.5, 10.5));
```

22. Interface or an Abstract class: which one to use?

- Use an interface to define a contract when multiple classes need to share common functionality but are unrelated.
- Use an abstract class when classes share a common base and some implementation.
- Abstract classes can have constructors and fields, but interfaces cannot.
- Example: I used an interface IMovable for vehicles and animals, and an abstract class Vehicle for shared vehicle properties.

```
interface IMovable {
   void Move();
}

abstract class Vehicle : IMovable {
   public abstract void Move();
}
```

23. What is the Unit of Work pattern?

- The Unit of Work pattern maintains a list of changes to be made and coordinates their execution as a single unit.
- It helps manage transactions by ensuring all operations either succeed or fail together.
- Commonly used with the Repository pattern in data persistence layers.
- Example: In an e-commerce app, I used Unit of Work to save orders and update inventory in a single transaction.

24. What is the difference between an Interface and an Abstract Class?

- An interface only declares methods and properties, while an abstract class can provide implementations.
- Abstract classes can have fields and constructors; interfaces cannot.
- A class can implement multiple interfaces but inherit only one abstract class.
- Example: I used interfaces for cross-cutting concerns like logging, and an abstract class for shared vehicle functionality.

```
abstract class Vehicle {
   public abstract void Drive();
}
interface ILogger {
   void Log(string message);
}
```

25. How can you prevent a class from overriding methods in C#?

- Use the sealed keyword with a method to prevent it from being overridden in derived classes.
- Sealed methods must be declared in classes that are not sealed themselves.
- This ensures no further modifications to the method's behavior.
- Example: I sealed the ProcessOrder method in an order management system to enforce standard behavior.

```
class Order {
   public virtual void ProcessOrder() => Console.WriteLine("Processing
order");
}

class SpecialOrder : Order {
   public sealed override void ProcessOrder() =>
Console.WriteLine("Processing special order");
}
```

26. What is the difference between a Virtual method and an Abstract method?

- A virtual method has a default implementation, while an abstract method does not and must be overridden.
- Virtual methods can be optionally overridden in derived classes; abstract methods must be implemented.
- Virtual methods are declared in non-abstract classes, whereas abstract methods are only in abstract classes.
- Example: In a file management app, I used a virtual OpenFile method with a default implementation and an abstract ParseFile method for specific file types.

```
abstract class File {
   public virtual void OpenFile() => Console.WriteLine("Opening file");
   public abstract void ParseFile();
}

class TextFile : File {
   public override void ParseFile() => Console.WriteLine("Parsing text file");
}
```

27. When should I use a struct instead of a class?

- Use a struct for lightweight objects that represent a single value or small group of related values.
- Structs are ideal when immutability and value-type behavior are required.
- Structs avoid heap allocation, making them more efficient for small, frequently used objects.
- Example: I used a Point struct in a graphics library for efficient manipulation of 2D coordinates.

```
struct Point {
   public int X { get; set; }
   public int Y { get; set; }
}
Point p = new Point { X = 5, Y = 10 };
```

28. What is Polymorphism, what is it for, and how is it used?

- Polymorphism allows objects to be treated as instances of their base type, enabling code generalization.
- It promotes flexibility and extensibility in systems.
- Polymorphism is implemented through method overriding or interfaces.
- Example: I used polymorphism in a payment gateway to handle various payment methods (CreditCard, PayPal) using a common interface.

```
interface IPayment {
   void ProcessPayment();
}

class CreditCardPayment : IPayment {
   public void ProcessPayment() => Console.WriteLine("Processing credit
card payment");
}
IPayment payment = new CreditCardPayment();
payment.ProcessPayment();
```

29. What are abstract classes? What are the distinct characteristics of an abstract class?

- Abstract classes cannot be instantiated and are designed to be inherited.
- They may contain abstract methods (without implementation) and concrete methods (with implementation).
- Abstract classes are used to define shared functionality while enforcing implementation in derived classes.
- Example: I used an abstract class Shape in a drawing application to define a Draw method that was implemented differently for Circle and Rectangle.

```
abstract class Shape {
   public abstract void Draw();
}
class Circle : Shape {
   public override void Draw() => Console.WriteLine("Drawing Circle");
}
```

30. State the features of an Interface.

- An interface defines a contract with no implementation.
- It supports multiple inheritance and allows classes to implement multiple interfaces.
- Interfaces cannot contain fields or constructors, only methods, properties, and events.
- Example: I used an interface ISerializable in a data export module to enforce a standard serialization method across classes.

```
interface ISerializable {
   string Serialize();
}
class Product : ISerializable {
   public string Serialize() => "Product data serialized";
}
```

31. How is method overriding different from method overloading?

- Overriding changes the behavior of a method in a derived class, while overloading provides multiple methods with the same name but different parameters.
- Overriding is achieved using virtual and override keywords; overloading doesn't require special keywords.
- Overriding is a runtime mechanism, while overloading is a compile-time mechanism.
- Example: I used overriding in a base class Animal to modify the Speak method in derived classes and overloading in a Calculator for adding integers and doubles.

```
// Overriding
class Animal {
   public virtual void Speak() => Console.WriteLine("Animal speaks");
}
class Dog : Animal {
   public override void Speak() => Console.WriteLine("Dog barks");
}

// Overloading
class Calculator {
   public int Add(int a, int b) => a + b;
   public double Add(double a, double b) => a + b;
}
```

32. What is a static constructor?

- A static constructor initializes static data or performs actions that only need to be done once for a class.
- It is called automatically before any instance is created or static members are accessed.
- It has no access modifiers and takes no parameters.
- Example: I used a static constructor in a configuration manager class to load settings once when the application started.

```
class ConfigManager {
   static ConfigManager() {
      Console.WriteLine("Loading configuration...");
   }
}
```

33. What exactly is the difference between an Interface and an Abstract class?

- Abstract classes can have concrete implementations; interfaces cannot (pre-C# 8.0).
- A class can inherit from one abstract class but implement multiple interfaces.
- Abstract classes can define fields and constructors; interfaces cannot.
- Example: I used an abstract class for shared properties in a report system and interfaces for behavior like IPrintable.

34. Differentiate between an abstract class and an interface.

- Abstract classes provide partial implementation, while interfaces provide no implementation.
- Abstract classes can contain non-public members; interfaces cannot.
- Interfaces support multiple inheritance, whereas abstract classes do not.
- Example: I used an interface for ILogging to enforce logging and an abstract class Device to share common device properties.

```
abstract class Device {
   public string Name { get; set; }
}
interface ILogging {
   void Log(string message);
}
```

35. Does .NET support multiple inheritance?

- .NET does not support multiple inheritance with classes but allows it with interfaces.
- This prevents ambiguity in inheriting members from multiple base classes.
- Mixins or interface combinations are used to achieve similar functionality.
- Example: I implemented multiple interfaces like IMovable and IDrawable in a game object.

```
interface IMovable {
   void Move();
}
interface IDrawable {
   void Draw();
}
class GameObject : IMovable, IDrawable {
   public void Move() => Console.WriteLine("Moving object");
   public void Draw() => Console.WriteLine("Drawing object");
}
```

36. What is Coupling in OOP?

- Coupling refers to the degree of dependency between different modules or classes in a program.
- Tight coupling occurs when classes are heavily dependent on each other, making the code less flexible.
- Loose coupling ensures minimal dependencies, leading to easier maintenance and scalability.
- Example: I used dependency injection to achieve loose coupling in a web application by injecting services like ILogger into controllers.

```
class Controller {
   private readonly ILogger _logger;

   public Controller(ILogger logger) {
      _logger = logger;
   }

   public void Execute() => _logger.Log("Executing action");
}
```

37. What is the difference between an abstract function and a virtual function?

- Abstract functions have no implementation and must be overridden in derived classes.
- Virtual functions have a default implementation that can be optionally overridden.
- Abstract functions are declared in abstract classes, whereas virtual functions can be in any class.
- Example: I used an abstract function Draw in a Shape class and a virtual function Display with default behavior.

```
abstract class Shape {
   public abstract void Draw();
}

class Circle : Shape {
   public override void Draw() => Console.WriteLine("Drawing Circle");
}
```

38. What is Cohesion in OOP?

- Cohesion refers to how closely related and focused the responsibilities of a single module or class are.
- High cohesion indicates that a class or module has a single, well-defined responsibility.
- Low cohesion can make a class harder to maintain and understand.
- Example: I designed a ReportGenerator class solely for generating reports, keeping responsibilities focused and cohesive.

```
class ReportGenerator {
   public void GenerateReport() => Console.WriteLine("Generating
Report");
}
```

39. Can you declare an overridden method to be static if the original method is not static?

- No, in C#, the modifier of the overridden method must match the original method's signature.
- A non-static method cannot be overridden as static, and vice versa.
- This ensures consistent behavior when invoking the method through a base class reference.
- Example: I maintained consistency by overriding non-static methods for polymorphic behavior in a user management system.

```
class User {
   public virtual void Login() => Console.WriteLine("User login");
}

class Admin : User {
   public override void Login() => Console.WriteLine("Admin login");
}
```

40. Could you explain some benefits of the Repository Pattern?

- The Repository Pattern abstracts the data access layer, providing a clean separation between business logic and data logic.

- It centralizes data access logic, making it reusable and easier to manage.
- It enables better testing by allowing mocking of the repository layer.
- Example: I implemented a UserRepository in a social networking app to manage user data interactions.

```
interface IUserRepository {
   User GetUserById(int id);
}
class UserRepository : IUserRepository {
   public User GetUserById(int id) => new User { Id = id, Name = "Sample
User" };
}
```

41. Explain the concept of Destructor.

- A destructor is used to release unmanaged resources held by an object before it is destroyed.
- In C#, destructors are declared using a tilde (~) followed by the class name.
- Destructors are automatically invoked and cannot be explicitly called.
- Example: I used a destructor in a FileHandler class to close file streams automatically when objects were no longer needed.

```
class FileHandler {
   ~FileHandler() => Console.WriteLine("FileHandler destructor called");
}
```

42. Explain different types of inheritance.

- **Single Inheritance**: A class inherits from one base class.
- **Multiple Inheritance**: A class inherits from multiple base classes (not supported directly in C#, but achievable through interfaces).
- **Multilevel Inheritance**: A class inherits from a class, which itself inherits from another class.
- **Hierarchical Inheritance**: Multiple classes inherit from a single base class.
- Example: I used hierarchical inheritance in an HR system where Employee was a base class, and Manager and Technician derived from it.

```
class Employee { }
class Manager : Employee { }
class Technician : Employee { }
```

43. What's the advantage of using getters and setters instead of simply using public fields?

- Getters and setters provide controlled access to fields, allowing validation or logic during data access.
- They help maintain encapsulation, keeping the internal state protected.
- Fields can be modified to properties without affecting external code.
- Example: I used a setter in a User class to validate email addresses before assignment.

```
class User {
   private string email;
   public string Email {
      get => email;
      set {
         if (value.Contains("@")) email = value;
      }
   }
}
```

44. How to solve Circular Reference?

- Avoid circular dependencies by redesigning relationships between classes or introducing interfaces.
- Use dependency injection to decouple classes.
- Consider breaking circular references by using events or callbacks.
- Example: I resolved a circular reference between Order and Customer classes by introducing an ICustomer interface.

```
interface ICustomer { }
class Customer : ICustomer { }
class Order {
   private ICustomer customer;
   public Order(ICustomer customer) => this.customer = customer;
}
```

45. When should I use an Interface and when should I use a Base Class?

- Use an interface for defining contracts when unrelated classes share common behavior.
- Use a base class when classes share common implementation or a logical hierarchy.
- Interfaces support multiple inheritance; base classes do not.
- Example: I used an interface IDrawable for rendering, and a base class Shape for shared geometry properties.

```
interface IDrawable {
   void Draw();
}
abstract class Shape : IDrawable {
   public abstract void Draw();
}
```

46. What is the difference between Cohesion and Coupling?

- Cohesion refers to how focused a class/module is on a single responsibility.
- Coupling refers to the level of dependency between classes/modules.
- High cohesion and low coupling are desired for maintainable and scalable code.
- Example: I achieved high cohesion by separating Invoice generation and email sending into distinct classes, reducing coupling through dependency injection.

47. What is the difference between Association, Aggregation, and Composition?

- **Association**: A general relationship between two classes, such as teacher and student.
- **Aggregation**: A "has-a" relationship where one class contains another, but they can exist independently.
- **Composition**: A stronger "has-a" relationship where the contained object cannot exist without the container.
- Example: I used aggregation for Team and Player objects and composition for Car and Engine.

```
class Engine { }
class Car {
   private Engine engine = new Engine();
}
```

48. Why doesn't C# allow static methods to implement an interface?

- Static methods are not tied to an instance, while interface methods are meant to be instance-level contracts.
- Allowing static methods in interfaces would violate the instance-based design of interfaces.
- Example: Instead of static methods, I used instance methods in the ILogger interface for logging functionality.

49. Can you provide a simple explanation of methods vs. functions in OOP context?

- A method is a function defined within a class and operates on its instance or class data.
- Functions are general and can exist outside of a class.
- Methods are context-specific to objects, while functions are context-independent.
- Example: I used methods like CalculateScore in a Player class tied to the object instance.

```
class Player {
    public int Score { get; set; }
    public void CalculateScore(int points) => Score += points;
}
```
50. Can you declare a private class in a namespace?

- No, classes declared directly in a namespace cannot have the private modifier.
- Classes within another class (nested classes) can be private.
- Example: I used a private nested class in a CacheManager for internal caching logic.

```
class CacheManager {
    private class CacheItem {
        public string Key { get; set; }
    }
}
```
51. Could you elaborate on Polymorphism, Overriding, and Overloading?

- **Polymorphism** allows the same operation to behave differently on different classes, supporting runtime flexibility.
- **Overriding** occurs when a derived class provides a new implementation for a virtual/abstract method from the base class.
- **Overloading** allows multiple methods in the same class with the same name but different parameters.
- Example: I used overriding in a PaymentProcessor to define ProcessPayment differently for CreditCard and PayPal classes, and overloading for logging messages.

```
class PaymentProcessor {
    public virtual void ProcessPayment() => Console.WriteLine("Processing
payment");
}
class CreditCardProcessor : PaymentProcessor {
    public override void ProcessPayment() => Console.WriteLine("Processing
credit card payment");
}
```
52. Why is a destructor in C# not executing?

- The garbage collector calls destructors, and execution timing isn't guaranteed.
- If the application ends before the garbage collector runs, the destructor might not execute.
- Explicit resource cleanup should use IDisposable and using blocks instead.
- Example: I implemented IDisposable in a file manager to ensure resource cleanup instead of relying on destructors.

```
class FileManager : IDisposable {
    public void Dispose() => Console.WriteLine("Releasing resources");
}
```
53. What is the difference between Mixins and Inheritance?

- Mixins provide reusable functionality through composition, often via interfaces or abstract classes.
- Inheritance is a hierarchical relationship between base and derived classes.
- Mixins allow combining behaviors from multiple sources; inheritance allows sharing structure and behavior from a single source.
- Example: I implemented mixins in a logging system by combining ILogger and IFormatter interfaces.

```
interface ILogger {
    void Log(string message);
}
interface IFormatter {
    string Format(string message);
}
class ConsoleLogger : ILogger, IFormatter {
    public void Log(string message) =>
Console.WriteLine(Format(message));
    public string Format(string message) => $"[LOG]: {message}";
}
```
54. What is LSP (Liskov Substitution Principle) and what are some examples of its use?

- LSP states that objects of a superclass should be replaceable with objects of a subclass without affecting the program.
- It ensures derived classes enhance, not modify or restrict, base class behavior.
- Violations occur when a subclass breaks expected behavior of the base class.
- Example: I applied LSP in a payment module, ensuring all payment types derived from a common Payment class adhered to shared behavior.

```
abstract class Payment {
    public abstract void Process();
}
class CreditCardPayment : Payment {
    public override void Process() => Console.WriteLine("Processing Credit
Card Payment");
}
Payment payment = new CreditCardPayment();
payment.Process(); // Works as expected
```
55. In terms that an OOP programmer would understand, what is a Monad?

- A Monad is a design pattern used to handle data transformations and chaining operations while maintaining context.
- It is commonly used in functional programming but is adaptable to OOP concepts.
- Monads ensure that operations handle additional concerns like nullability, errors, or side effects.
- Example: In a banking app, I used a Monad-like approach to handle safe chaining of nullable objects.

```
public class Maybe<T> {
    private readonly T _value;
    public Maybe(T value) => _value = value;
    public TResult Bind<TResult>(Func<T, TResult> func) => _value != null ?
func(_value) : default;
}
```
56. Why prefer Composition over Inheritance? What trade-offs are there for each approach?

- Composition promotes flexibility by combining behaviors dynamically rather than relying on static hierarchies.
- It avoids the tight coupling and fragility of deep inheritance trees.
- Inheritance is simpler for shared behavior in strongly related classes.
- Example: I used composition in a plugin system, combining IRenderer and IExporter for modular functionality.

```
interface IRenderer { void Render(); }
interface IExporter { void Export(); }
class Plugin : IRenderer, IExporter {
    public void Render() => Console.WriteLine("Rendering...");
    public void Export() => Console.WriteLine("Exporting...");
}
```
57. What does it mean to program to an Interface?

- Programming to an interface means depending on abstractions rather than concrete implementations.
- It promotes flexibility and enables easy swapping of implementations.
- This approach aligns with the Dependency Inversion Principle.
- Example: I designed a payment system by programming to the IPayment interface, allowing easy addition of new payment methods.

```
interface IPayment {
    void ProcessPayment();
}
class CreditCardPayment : IPayment {
    public void ProcessPayment() => Console.WriteLine("Credit Card
Payment Processed");
}
```

1. What is RxJS, and how does it relate to reactive programming?

- RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using Observables to handle asynchronous data streams.
- Reactive programming emphasizes handling continuous data streams and propagating changes.
- RxJS provides tools like Observables, operators, and Subjects for managing complex asynchronous workflows.
- It is commonly used in Angular for handling HTTP requests, user events, and timers in a declarative way.
- **Example:** I used RxJS in an Angular project to implement live search for fetching filtered results dynamically.

```
searchControl.valueChanges.pipe(
  debounceTime(300),
  switchMap(query => this.http.get(`/api/search?q=${query}`))
).subscribe(results => console.log(results));
```

2. Explain the differences between Observable and Subject in RxJS.

- An Observable is a lazy data producer that emits values to its subscribers only when subscribed.
- A Subject is both an Observable and an Observer, enabling multicasting to multiple subscribers.
- Observables create separate executions for each subscriber, whereas Subjects share a single execution.
- Subjects can also emit values manually, while Observables emit values according to their source logic.
- **Example:** I used a Subject in a chat application to broadcast new messages to all connected clients.

```
const messageSubject = new Subject<string>();

messageSubject.subscribe(msg => console.log('User 1 received:', msg));
messageSubject.subscribe(msg => console.log('User 2 received:', msg));

messageSubject.next('Hello, everyone!');
```

3. What are the different types of subjects in RxJS, and when would you use each?

- **Subject:** General-purpose multicast Observable, used for broadcasting values to multiple subscribers.
- **BehaviorSubject:** Emits the most recent value to new subscribers, suitable for state management.
- **ReplaySubject:** Emits a specified number of previous values to new subscribers, useful for caching or replaying events.
- **AsyncSubject:** Emits the last value upon completion, often used for one-time results or final outputs.
- **Example:** I used a BehaviorSubject in an Angular project to manage global application state and emit the current user data to components.

```
const userSubject = new BehaviorSubject<string>('Guest');

userSubject.subscribe(user => console.log('Current user:', user));

userSubject.next('Admin'); // Updates state to 'Admin'
```

4. How does the pipe method work in RxJS?

- The pipe method is used to chain multiple operators together for transforming and managing Observable streams.
- It allows for a declarative approach, improving readability and maintainability of the code.
- Operators in the pipe method are applied sequentially in the order they are defined.
- Common operators used with pipe include map, filter, mergeMap, and catchError.
- **Example:** I used pipe in an Angular project to transform HTTP response data and handle errors efficiently.

```
this.http.get('/api/data').pipe(
  map(data => data['results']),
  filter(results => results.length > 0),
  catchError(error => of([]))
).subscribe(results => console.log(results));
```

5. What are operators in RxJS, and what is the difference between creation and transformation operators?

- Operators are functions that enable transformation, filtering, and combination of Observables in RxJS.
- **Creation operators** generate new Observables, e.g., of, from, interval.
- **Transformation operators** modify emitted data or Observable behavior, e.g., map, mergeMap, switchMap.
- Operators can be chained together using the pipe method.
- **Example:** In my project, I used interval (creation) and map (transformation) to create a countdown timer.

```
import { interval } from 'rxjs';
import { map } from 'rxjs/operators';

interval(1000).pipe(
  map(seconds => 10 - seconds)
).subscribe(time => console.log(time));
```

6. Explain the difference between mergeMap, switchMap, and concatMap in RxJS.

- **mergeMap:** Projects each source value to an Observable and merges the outputs concurrently.
- **switchMap:** Projects each source value to an Observable but unsubscribes from previous ones if a new value arrives.
- **concatMap:** Projects each source value to an Observable, ensuring each completes before the next starts.
- **When to use:**
  - mergeMap: Use when concurrency is needed, e.g., processing multiple API requests simultaneously.
  - switchMap: Use for scenarios where only the latest value matters, e.g., autocomplete search.
  - concatMap: Use for ordered execution, e.g., sequential HTTP calls.
- **Example:** In my Angular project, switchMap was used for live search to cancel previous API calls when a new input was typed.

```
searchControl.valueChanges.pipe(
  debounceTime(300),
  switchMap(query => this.http.get(`/api/search?q=${query}`))
).subscribe(results => console.log(results));
```

7. What is a cold observable, and how does it differ from a hot observable?

- A **cold Observable** produces values independently for each subscriber, like a video-on-demand service.
- A **hot Observable** shares the same execution among all subscribers, like a live broadcast.
- Cold Observables start execution only when subscribed, while hot Observables may start earlier.
- Converting cold to hot can be done using Subjects or the share operator.

- **Example:** I converted a cold Observable to hot in my project using a Subject for event broadcasting.

```
const cold$ = new Observable(observer =>
observer.next(Math.random()));
const hot$ = cold$.pipe(share());

hot$.subscribe(val => console.log('Subscriber 1:', val));
hot$.subscribe(val => console.log('Subscriber 2:', val));
```

8. How do you handle errors in RxJS streams?

- Use the catchError operator to catch and handle errors in the stream.
- Use the retry or retryWhen operator to attempt recovery by retrying.
- Use the finalize operator for cleanup actions, regardless of success or error.
- Emit a fallback value or empty stream using of or EMPTY in case of an error.
- **Example:** In my project, I handled API errors gracefully by retrying once and providing a default fallback value.

```
this.http.get('/api/data').pipe(
  retry(1),
  catchError(error => of({ message: 'Fallback data' }))
).subscribe(data => console.log(data));
```

9. What is the purpose of the takeUntil operator in RxJS?

- The takeUntil operator unsubscribes from an Observable when another Observable emits a value.
- It is commonly used to manage subscriptions in components, especially to prevent memory leaks.
- Works by listening to a notifier Observable (e.g., onDestroy event) and completing the subscription.
- Improves resource management by ensuring unused streams are unsubscribed automatically.
- **Example:** I used takeUntil in my Angular project to clean up subscriptions when a component was destroyed.

```
const destroy$ = new Subject<void>();

this.someObservable.pipe(
  takeUntil(destroy$)
).subscribe(data => console.log(data));

// Trigger on component destruction
destroy$.next();
destroy$.complete();
```

10. How do you use RxJS to manage state in an Angular application?

- Use BehaviorSubject or ReplaySubject to hold and emit the current state.
- Combine multiple state streams using operators like combineLatest or withLatestFrom.
- Create a centralized state service to manage application-wide state.
- Use operators like distinctUntilChanged to prevent unnecessary emissions.
- **Example:** I used BehaviorSubject in a project to manage user authentication state and emit updates to components.

```
const authState$ = new BehaviorSubject<boolean>(false);

// Update state
authState$.next(true);

// Subscribe to state changes
authState$.subscribe(isAuthenticated => console.log(isAuthenticated));
```

11. Have you worked with RxJS in Angular? How is it used for handling asynchronous operations?

- RxJS is extensively used in Angular for managing asynchronous tasks like HTTP calls, form events, and router events.
- Observables allow reactive programming patterns, ensuring better resource management.
- Operators like mergeMap, switchMap, and catchError are commonly used for HTTP operations.
- The async pipe in templates automatically handles subscription and unsubscription of Observables.
- **Example:** I used RxJS with Angular's HTTP client to fetch data and handle errors dynamically.

```
this.http.get('/api/users').pipe(
  catchError(() => of([]))
).subscribe(users => console.log(users));
```

12. What are RxJS operators, and how do you use map, filter, and switchMap?

- Operators are functions that manipulate Observable streams to transform, filter, or combine data.
- map: Transforms each emitted value, e.g., extracting a specific property from an object.
- filter: Filters emissions based on a condition.
- switchMap: Switches to a new Observable and cancels the previous one when a new value is emitted.
- **Example:** I used map, filter, and switchMap in a live search feature to filter input and fetch data.

```
searchControl.valueChanges.pipe(
  filter(query => query.length > 2),
  map(query => query.trim()),
  switchMap(query => this.http.get(`/api/search?q=${query}`))
).subscribe(results => console.log(results));
```

13. Which RxJS operators have you used in your project?

- Commonly used operators include map, filter, mergeMap, switchMap, catchError, and combineLatest.
- Operators like takeUntil and finalize are used for managing subscriptions and cleanup.
- startWith and scan are used for initializing and reducing data streams.
- retry and debounceTime are used for error handling and event debouncing.
- **Example:** I used combineLatest to synchronize multiple state streams in a real-time dashboard project.

```
combineLatest([stream1$, stream2$]).subscribe(([value1, value2]) => {
  console.log(value1, value2);
});
```

14. What is the difference between mergeMap, switchMap, and concatMap in RxJS? Provide examples.

- **mergeMap**: Handles multiple inner Observables concurrently.

```
source$.pipe(
  mergeMap(val => this.http.get(`/api/item/${val}`))
).subscribe(result => console.log(result));
```

- **switchMap**: Cancels the previous Observable when a new one is emitted.

```
source$.pipe(
  switchMap(val => this.http.get(`/api/item/${val}`))
).subscribe(result => console.log(result));
```

- **concatMap**: Executes Observables sequentially, waiting for each to complete.

```
source$.pipe(
  concatMap(val => this.http.get(`/api/item/${val}`))
).subscribe(result => console.log(result));
```

- **Use Case:** I used switchMap in an Angular app for live search to ensure only the latest query was processed.

---

15. How does RxJS handle backpressure in streams, and what are some strategies to manage it?

- Backpressure occurs when the data producer emits values faster than the consumer can process.
- Use buffer or bufferTime to collect values before processing.
- Use throttleTime or debounceTime to limit emissions over time.
- Apply sample or auditTime to pick specific emissions from the stream.
- **Example:** I used bufferTime in a project to process batch updates instead of individual events.

```
source$.pipe(
  bufferTime(1000)
).subscribe(batch => console.log('Batch:', batch));
```

---

16. Explain the purpose of BehaviorSubject and how it differs from Subject.

- BehaviorSubject stores the latest value and emits it immediately to new subscribers, while Subject does not store any value.
- Subject starts emitting values only after a subscriber is added, whereas BehaviorSubject emits the most recent value even to late subscribers.
- BehaviorSubject requires an initial value at the time of creation.
- It is commonly used for state management in applications to share the latest state across multiple components.
- **Example:** I used BehaviorSubject in a project to maintain and broadcast the current authentication state.

```
const authState = new BehaviorSubject<boolean>(false);

authState.next(true); // Update the state
authState.subscribe(state => console.log('Auth state:', state));
```

---

17. How can you combine multiple Observables in RxJS using forkJoin? Provide a practical example.

- forkJoin combines multiple Observables and emits the last value from each when all Observables complete.
- It is suitable for scenarios where all Observables need to complete before processing the result.
- If any Observable errors, forkJoin will also emit an error.
- Useful in making parallel HTTP requests and waiting for all responses.
- **Example:** I used forkJoin in an Angular app to fetch user details and their associated posts simultaneously.

```
forkJoin({
  user: this.http.get('/api/user/1'),
  posts: this.http.get('/api/user/1/posts')
}).subscribe(({ user, posts }) => {
  console.log('User:', user);
  console.log('Posts:', posts);
});
```

---

18. What is the shareReplay operator, and when would you use it?

- shareReplay shares a single subscription among multiple subscribers and replays the specified number of emitted values to new subscribers.
- It helps avoid duplicate network calls by sharing cached values.
- Typically used in caching HTTP requests and optimizing resource usage.
- The replay count determines how many past emissions are replayed to new subscribers.
- **Example:** I used shareReplay to cache API responses for a settings page to prevent redundant HTTP calls.

```
const settings$ = this.http.get('/api/settings').pipe(
  shareReplay(1) // Cache the last value
);

settings$.subscribe(data => console.log('Subscriber 1:', data));
settings$.subscribe(data => console.log('Subscriber 2:', data));
```

---

19. How does the auditTime operator work, and how is it different from debounceTime?

- auditTime emits the most recent value after a specified time interval since the last emission.
- It ignores intermediate values during the interval but always emits the final one.
- debounceTime waits for a pause in emissions before emitting the last value.
- Use auditTime for regular sampling of events and debounceTime for handling user input.
- **Example:** I used auditTime in a project to limit UI updates while tracking mouse movements.

```
fromEvent(document, 'mousemove').pipe(
  auditTime(1000)
).subscribe(event => console.log('Mouse moved:', event));
```

---

20. Explain cold vs. hot Observables in RxJS and their implications on subscription behavior.

- **Cold Observables:** Each subscriber receives a new execution of the Observable, and data is produced only on subscription.
- **Hot Observables:** Subscribers share the same execution, and data is produced regardless of subscriptions.
- Converting cold to hot can be done using share, shareReplay, or a Subject.
- Cold Observables are used for fetching data (e.g., HTTP requests), while hot Observables suit event-based streams (e.g., WebSocket).
- **Example:** I used a Subject to convert a cold HTTP Observable to a hot Observable in a notification system.

```
const cold$ = this.http.get('/api/notifications');
const hot$ = cold$.pipe(share());

hot$.subscribe(data => console.log('Subscriber 1:', data));
hot$.subscribe(data => console.log('Subscriber 2:', data));
```

---

21. What is the role of a scheduler in RxJS, and how does it affect Observable execution?

- A scheduler controls the timing and order of Observable execution.

- Schedulers determine when emissions happen and on which thread.
- Common schedulers include asyncScheduler (asynchronous execution), queueScheduler (synchronous execution), and animationFrameScheduler (aligned with browser rendering).
- Used for performance tuning and handling specific execution contexts.
- **Example:** I used asyncScheduler in a project to schedule delayed tasks in an Observable stream.

```
of(1, 2, 3, asyncScheduler).subscribe(val => console.log(val));
```

22. How do you create a custom operator in RxJS, and what are its use cases?

- Custom operators are created by defining a function that returns an Observable.
- They are used to encapsulate reusable logic and improve code readability.
- Implement using the Observable constructor or by composing existing operators.
- Useful for scenarios like custom error handling, complex data transformations, or combining streams.
- **Example:** I created a custom operator in a project to retry an HTTP request with exponential backoff.

```
function retryWithBackoff(retryCount: number, delayMs: number) {
  return (source$: Observable<any>) => source$.pipe(
    retryWhen(errors =>
      errors.pipe(
        scan((count, error) => {
          if (count >= retryCount) throw error;
          return count + 1;
        }, 0),
        delay(delayMs)
      )
    )
  );
}
```

23. What is combineLatestWith introduced in RxJS 7, and how does it enhance Observable combination?

- combineLatestWith is an operator that combines the latest values from multiple Observables when any emits.
- It simplifies combining streams compared to the standalone combineLatest function.
- Ensures all Observables have emitted at least once before emitting a combined value.
- Commonly used for synchronizing data streams, such as combining form control states.
- **Example:** I used combineLatestWith in a project to synchronize user input and server data for a real-time dashboard.

```
input$.pipe(
  combineLatestWith(serverData$)
).subscribe(([input, data]) => console.log('Combined:', input, data));
```

24. What is the difference between exhaustMap and switchMap, and when would you use each?

- **exhaustMap** ignores new emissions from the source Observable until the current inner Observable completes.
- **switchMap** unsubscribes from the previous inner Observable and switches to a new one for every emission from the source.
- Use exhaustMap when you want to prevent overlapping inner subscriptions (e.g., handling form submissions).
- Use switchMap when only the latest emission matters (e.g., autocomplete search).
- **Example:** I used exhaustMap to handle login requests and prevent multiple submissions in a project.

```
loginButtonClick$.pipe(
  exhaustMap(() => this.authService.login())
).subscribe(response => console.log('Login successful:', response));
```

25. How does startWith operator work in RxJS, and where is it useful?

- The startWith operator emits an initial value before emitting any values from the source Observable.
- Useful for providing default values or setting an initial state.
- It is commonly used in state management or form initialization.
- Can be combined with scan or map for more complex use cases.
- **Example:** I used startWith in a project to initialize a stream with a default value for a loading spinner.

```
dataStream$.pipe(
  startWith({ loading: true })
).subscribe(state => console.log('State:', state));
```

26. What is the purpose of scan operator in RxJS, and how does it differ from reduce?

- The scan operator accumulates values over time and emits the accumulated value with each new emission.
- reduce accumulates values but emits only once when the source Observable completes.
- Use scan for tracking state updates or progressive transformations.
- Use reduce for summarizing values into a single result at the end.
- **Example:** I used scan in a project to keep a running total of user clicks dynamically.

```
clicks$.pipe(
  scan((total, _) => total + 1, 0)
).subscribe(total => console.log('Total clicks:', total));
```

27. How do you debounce user input using RxJS, and why is it important?

- Use the debounceTime operator to wait for a pause in user input before processing the value.
- It prevents unnecessary processing for every keystroke, improving performance and user experience.
- Particularly useful in scenarios like live search or form validation.
- Combine with distinctUntilChanged to emit values only if they have changed.
- **Example:** I used debounceTime in a project to optimize API calls for live search functionality.

```
searchInput$.pipe(
  debounceTime(300),
  distinctUntilChanged()
).subscribe(query => console.log('Search query:', query));
```

28. Explain the concept of multicasting in RxJS and how multicast operator is used.

- Multicasting allows sharing a single Observable execution among multiple subscribers.
- The multicast operator uses a Subject to broadcast values to all subscribers.

- Helps reduce redundant processing or network calls by sharing data.
- Typically combined with connect or used as part of higher-order operators like share.
- **Example:** I used multicast in a project to share a WebSocket stream among multiple components.

```
const sharedStream$ = source$.pipe(
  multicast(new Subject())
);

sharedStream$.connect();
```

29. What is the difference between combineLatest and withLatestFrom, and when to use each?

- combineLatest emits the latest values from all input Observables whenever any of them emits.
- withLatestFrom emits the latest value from other Observables only when the primary Observable emits.
- Use combineLatest when all Observables are equally important.
- Use withLatestFrom when the primary Observable drives emissions.
- **Example:** I used withLatestFrom in a project to combine user input with the latest server state.

```
userAction$.pipe(
  withLatestFrom(serverState$)
).subscribe(([action, state]) => console.log('Action:', action, 'State:', state));
```

30. How does retryWhen operator work, and how is it different from retry?

- retryWhen retries an Observable based on a custom logic defined in a notifier Observable.
- retry retries a fixed number of times immediately upon failure.
- Use retryWhen for more advanced retry strategies like exponential backoff.
- It listens to errors from the source Observable and determines when to resubscribe.
- **Example:** I used retryWhen in a project to implement retry logic with delays for HTTP requests.

```
this.http.get('/api/data').pipe(
  retryWhen(errors =>
    errors.pipe(delay(1000))
  )
).subscribe(data => console.log('Data:', data));
```

31. What is the purpose of the pluck operator, and how does it simplify stream data?

- The pluck operator extracts a specific property from emitted objects.
- Simplifies data transformation by directly accessing nested properties.
- Useful for working with streams of objects where only one property is needed.
- Reduces the need for custom mapping logic.
- **Example:** I used pluck in a project to extract user IDs from a stream of user objects.

```
userStream$.pipe(
  pluck('id')
).subscribe(id => console.log('User ID:', id));
```

32. How do you merge multiple Observables in RxJS, and what is the use of merge operator?

- The merge operator combines multiple Observables and emits values as they arrive from any source.
- Useful for combining events like user clicks or data streams from different sources.
- Does not wait for completion of one Observable before processing the next.
- Ideal for handling concurrent data streams.
- **Example:** I used merge in a project to handle events from multiple buttons in a single stream.

```
merge(button1Click$, button2Click$).subscribe(event =>
console.log('Event:', event));
```

33. Explain the role of zip operator in RxJS and provide a use case.

- The zip operator combines multiple Observables by pairing their emissions.
- Emits values only when all input Observables emit, combining them into arrays or objects.
- Useful for scenarios requiring synchronized data streams.
- Emits as many values as the shortest Observable.
- **Example:** I used zip in a project to combine user details and preferences fetched from different APIs.

```
zip(userDetails$, userPreferences$).subscribe(([details, preferences]) => {
  console.log('User:', details, 'Preferences:', preferences);
});
```

34. How does the groupBy operator work in RxJS, and what are its applications?

- The groupBy operator splits an Observable into multiple Observables based on a key.
- Each grouped Observable emits values corresponding to its key.
- Useful for categorizing or grouping data streams dynamically.
- Combine with mergeMap or toArray to process grouped data.
- **Example:** I used groupBy in a project to categorize incoming messages by user.

```
messages$.pipe(
  groupBy(msg => msg.userId),
  mergeMap(group$ => group$.pipe(toArray()))
).subscribe(groupedMessages => console.log('Grouped Messages:',
groupedMessages));
```

35. What is the difference between tap and do in RxJS, and how is tap commonly used?

- do was an older name for tap and has been replaced with tap in newer RxJS versions.
- tap is used to perform side effects like logging, debugging, or triggering actions without modifying the stream.
- It is a purely side-effect operator and does not alter the emitted values.
- Commonly used for logging emissions, managing external state, or triggering analytics.
- **Example:** I used tap in a project to log HTTP responses for debugging purposes.

```
this.http.get('/api/data').pipe(
  tap(data => console.log('Fetched Data:', data))
).subscribe();
```

36. How do you use interval and timer in RxJS to create scheduled Observables?

- interval creates an Observable that emits sequential numbers at specified intervals.
- timer creates an Observable that emits after a delay or at a delay and continues at regular intervals.
- Both are used for periodic tasks like polling or animations.
- Can be combined with operators like take or switchMap for specific use cases.
- **Example:** I used interval in a project to implement periodic updates for a dashboard.

```
interval(1000).pipe(
  take(5)
).subscribe(count => console.log('Count:', count));
```

---

37. How does the race operator work in RxJS, and where is it typically used?

- The race operator subscribes to multiple Observables and emits from the one that emits first.
- Subsequent emissions from other Observables are ignored.
- Used for scenarios where you want to proceed with the fastest Observable.
- Useful in fallback mechanisms or resolving multiple competing tasks.
- **Example:** I used race in a project to select the fastest response between two APIs.

```
race(api1$, api2$).subscribe(response => console.log('Fastest Response:', response));
```

---

38. What are some best practices for unsubscribing from Observables in RxJS?

- Use the takeUntil operator with a notifier Observable for controlled unsubscription.
- Leverage Angular's async pipe for template-based subscriptions, which auto-unsubscribe.
- Use Subscription objects and manually unsubscribe in ngOnDestroy.
- Prefer operators like first, take, or takeWhile for automatic completion.
- **Example:** I used takeUntil with a Subject in a project to manage cleanup in Angular components.

```
const destroy$ = new Subject<void>();

this.dataStream$.pipe(
  takeUntil(destroy$)
).subscribe(data => console.log('Data:', data));

// Trigger cleanup
destroy$.next();
destroy$.complete();
```

---

39. How does window operator work in RxJS, and what are its applications?

- The window operator splits an Observable into multiple windows (Observables), each emitting a subset of values.
- Windows are created based on another Observable or a condition.
- Useful for batching or segmenting streams dynamically.
- Combine with mergeMap or toArray to process windowed data.
- **Example:** I used window in a project to batch user clicks and process them periodically.

```
clicks$.pipe(
  window(interval(1000)),
  mergeMap(window$ => window$.pipe(toArray()))
).subscribe(batch => console.log('Click Batch:', batch));
```

---

40. What is the purpose of the catchError operator, and how is it commonly used?

- catchError handles errors in an Observable stream and provides an alternative Observable or rethrows the error.
- Prevents stream termination by gracefully recovering from errors.
- Commonly used with HTTP requests or user inputs to provide fallback data or notifications.
- Can be combined with retry mechanisms for robust error handling.
- **Example:** I used catchError in a project to handle failed API requests and return fallback data.

```
this.http.get('/api/data').pipe(
  catchError(err => {
    console.error('Error:', err);
    return of({ fallback: true });
  })
).subscribe(data => console.log('Data:', data));
```

---

41. Explain the difference between fromEvent and fromEventPattern in RxJS.

- fromEvent creates an Observable from DOM events like clicks or key presses.
- fromEventPattern is more flexible, allowing custom event binding and unbinding logic.
- Use fromEvent for standard event listeners and fromEventPattern for custom or non-DOM events.
- Both can be combined with operators like map or filter for event processing.
- **Example:** I used fromEvent in a project to track user interactions with a form.

```
fromEvent(button, 'click').subscribe(() => console.log('Button clicked'));
```

---

42. What is the purpose of defer in RxJS, and when would you use it?

- defer creates an Observable that is defined at the time of subscription.
- Useful for scenarios where Observable creation needs to be delayed or depends on runtime conditions.
- Ensures a fresh execution context for each subscription.
- Commonly used for lazy initialization or dynamic data fetching.
- **Example:** I used defer in a project to conditionally create HTTP requests based on user input.

```
const api$ = defer(() => this.http.get(`/api/data/${userId}`));
api$.subscribe(data => console.log('Data:', data));
```

---

43. How does onErrorResumeNext operator work, and how is it different from catchError?

- onErrorResumeNext continues with subsequent Observables after an error, without handling or logging the error.
- catchError handles the error and allows custom recovery logic or logging.
- Use onErrorResumeNext for resilient pipelines where errors can be ignored.
- Commonly applied in streams with non-critical tasks.
- **Example:** I used onErrorResumeNext in a project to continue processing data even if one source failed.

```
onErrorResumeNext(api1$, api2$).subscribe(data => console.log('Data:', data));
```

---

44. How does the bufferCount operator work, and what are its use cases?

- The bufferCount operator collects a specified number of values from the source Observable into an array and emits them as a batch.
- Useful for batching tasks or processing chunks of data.
- Can specify a sliding window for overlapping buffers.
- Ideal for optimizing resource-intensive operations like API calls or database writes.
- **Example:** I used bufferCount in a project to process user actions in batches of 5.

```
actionStream$.pipe(
  bufferCount(5)
).subscribe(batch => console.log('Action Batch:', batch));
```

---

45. What is the purpose of the delayWhen operator, and how is it different from delay?

- delayWhen delays emissions based on another Observable, providing dynamic delay durations.
- delay uses a fixed time duration for delaying emissions.
- Use delayWhen for scenarios requiring variable delays based on runtime conditions.
- Combine with operators like timer for advanced delay patterns.
- **Example:** I used delayWhen in a project to delay notifications based on user activity.

```
notifications$.pipe(
  delayWhen(() => timer(2000))
).subscribe(notification => console.log('Notification:', notification));
```

---

1. What is a PRIMARY KEY?

- Uniquely identifies each record in a table.
- Ensures that the column(s) contain unique, non-null values.
- There can only be one primary key per table, which may consist of single or multiple columns.
- Example: CREATE TABLE Employee (ID INT PRIMARY KEY, Name NVARCHAR(50));

2. Define a Temp Table.

- A temporary table is a table stored in the tempdb database and is available for the current session.
- It is used to store intermediate results temporarily during query execution.
- Prefixed with # for local temp tables or ## for global temp tables.
- Example: CREATE TABLE #TempTable (ID INT, Name NVARCHAR(50)); INSERT INTO #TempTable VALUES (1, 'John');

3. What is a VIEW?

- A VIEW is a virtual table based on the result of an SQL query.
- It simplifies complex queries by encapsulating them as reusable objects.
- Does not store data itself; changes in the base table reflect in the VIEW.
- Example: CREATE VIEW EmployeeView AS SELECT ID, Name FROM Employee WHERE Department = 'IT';

4. What is the DEFAULT constraint?

- The DEFAULT constraint sets a default value for a column if no value is provided.
- It helps maintain consistency in data.
- Can be applied during table creation or modification.
- Example: CREATE TABLE Employee (ID INT, Salary INT DEFAULT 5000);

5. What is the difference between Data Definition Language (DDL) and Data Manipulation Language (DML)?

- DDL deals with schema and structure changes (e.g., CREATE, ALTER, DROP).
- DML manipulates the data within tables (e.g., INSERT, UPDATE, DELETE).
- DDL operations are auto-committed, while DML requires explicit commits.
- Example: DDL: CREATE TABLE Employee (ID INT); DML: INSERT INTO Employee VALUES (1);

6. What is the difference between TRUNCATE and DELETE?

- TRUNCATE removes all rows without logging individual row deletions.
- DELETE removes rows selectively based on a condition and logs each deletion.
- TRUNCATE resets identity columns, while DELETE does not.
- Example: TRUNCATE TABLE Employee; vs. DELETE FROM Employee WHERE ID = 1;

7. What is a FOREIGN KEY?

- A FOREIGN KEY enforces referential integrity between two tables.
- It establishes a relationship by linking a column to the primary key of another table.
- Prevents invalid data entry by ensuring related records exist in the referenced table.
- Example: CREATE TABLE Orders (OrderID INT, CustomerID INT FOREIGN KEY REFERENCES Customers(CustomerID));

8. What is Normalization?

- Process of organizing database to reduce redundancy and dependency.
- Divides tables into smaller, logically connected ones.
- Ensures data integrity and efficient updates.
- Example: In my project, I normalized customer data into separate tables for Customers and Orders.

9. What is Denormalization?

- Process of combining normalized tables to optimize read performance.
- Introduces redundancy to reduce the complexity of joins.
- Used in OLAP systems for faster query execution.
- Example: Denormalized Customer and Order tables into one for reporting performance.

10. What is the difference between the WHERE clause and the HAVING clause?

- WHERE filters rows before grouping, HAVING filters after grouping.
- WHERE cannot work with aggregate functions; HAVING can.
- Both can be used together in the same query.
- Example: SELECT Department, AVG(Salary) FROM Employee GROUP BY Department HAVING AVG(Salary) > 5000;

11. What is the difference between JOIN and UNION?

- JOIN combines columns from related tables, UNION combines rows from queries.
- JOIN needs a relationship between tables, UNION works on independent results.
- UNION removes duplicates by default; UNION ALL retains all rows.
- Example: SELECT Name FROM Employee UNION SELECT Name FROM Manager;

12. What are the differences between Clustered and Non-clustered indexes?

- Clustered index sorts and stores data rows; non-clustered does not.
- A table can have only one clustered index but multiple non-clustered indexes.
- Clustered index is faster for range queries; non-clustered for exact lookups.
- Example: CREATE CLUSTERED INDEX IX_Employee ON Employee(ID);

13. How does a Hash index work?

- Uses a hash function to map keys to a location in the index.
- Efficient for equality searches but not suitable for range queries.
- Requires space for hash tables in memory or disk.
- Example: Used hash indexing for quick lookup in a large user authentication table.

14. What is the difference between INNER JOIN and OUTER JOIN?

- INNER JOIN returns matching rows from both tables.
- OUTER JOIN includes unmatched rows from one or both tables, depending on type.
- Types: LEFT OUTER, RIGHT OUTER, FULL OUTER JOIN.
- Example: SELECT e.Name, d.Name FROM Employee e LEFT JOIN Department d ON e.DeptID = d.ID;

15. What is Collation in SQL?

- Collation defines rules for text sorting and comparison.
- Includes sensitivity to case, accents, and locale-specific rules.
- Important for multilingual databases.
- Example: SELECT * FROM Employee WHERE Name COLLATE Latin1_General_BIN = 'john';

16. What's the difference between a Primary Key and a Unique Key?

- Primary Key enforces uniqueness and non-null constraints; Unique Key allows one NULL value.
- A table can have only one Primary Key but multiple Unique Keys.
- Primary Key creates a clustered index by default; Unique Key creates a non-clustered index.
- Example: CREATE TABLE Employee (ID INT PRIMARY KEY, Email NVARCHAR(50) UNIQUE);

17. How can a VIEW be used to provide a security layer for your app?

- Restricts access by exposing only necessary columns or rows.
- Encapsulates complex logic, hiding table structure from users.
- Prevents direct access to base tables and sensitive data.
- Example: CREATE VIEW PublicEmployeeData AS SELECT Name, Department FROM Employee;

18. What's the difference between Azure SQL Database and Azure SQL Managed Instance?

- Azure SQL Database is a PaaS offering for single databases; Managed Instance is for near full SQL Server compatibility.
- Managed Instance supports features like cross-database queries and SQL Agent.
- SQL Database is more cost-effective for isolated workloads; Managed Instance suits enterprise applications.
- Example: I used Azure SQL Database for an e-commerce app with independent databases.

19. How can a database index help performance?

- Speeds up data retrieval by reducing the number of rows scanned.
- Organizes data for efficient search and filtering operations.
- Impacts write operations slightly due to index maintenance.
- Example: Added indexes to optimize search queries for product inventory in a retail app.

20. Discuss INNER JOIN vs WHERE clause (with multiple FROM tables).

- INNER JOIN explicitly defines relationships; WHERE filters results.
- INNER JOIN is more readable and declarative.
- For multiple FROM tables, INNER JOIN avoids Cartesian products.
- Example: SELECT e.Name, d.Name FROM Employee e INNER JOIN Department d ON e.DeptID = d.ID;

21. Define ACID Properties.

- Atomicity ensures transactions are all-or-nothing.
- Consistency maintains database integrity.
- Isolation ensures concurrent transactions don't interfere.
- Durability guarantees persistence of committed data.
- Example: Used ACID-compliant databases to handle financial transactions securely.

22. Describe the difference between TRUNCATE and DELETE.

- TRUNCATE removes all rows without logging row deletions.
- DELETE selectively removes rows and logs each action.
- TRUNCATE resets identity columns, DELETE does not.
- Example: Used DELETE to remove expired coupons while retaining history logs.

23. What is the difference between UNION and UNION ALL?

- UNION removes duplicate rows; UNION ALL retains all rows.
- UNION requires additional processing for duplicate elimination.
- UNION ALL is faster due to no deduplication.
- Example: SELECT Name FROM Employee UNION ALL SELECT Name FROM Contractor;

24. What is the difference between INNER JOIN, OUTER JOIN, and FULL OUTER JOIN?

- INNER JOIN retrieves matching rows from both tables.
- OUTER JOIN includes unmatched rows from one table (LEFT or RIGHT).
- FULL OUTER JOIN includes unmatched rows from both tables.
- Example: SELECT e.Name, d.Name FROM Employee e FULL OUTER JOIN Department d ON e.DeptID = d.ID;

25. What is the cost of having a database index?

- Additional storage for index structure.
- Increased write operation time due to index maintenance.
- Potential performance degradation if over-indexed.
- Example: Balanced indexes to improve search times without hindering batch inserts in an analytics database.

26. What is faster, one big query or many small queries?

- One big query minimizes network overhead but can be harder to debug.
- Small queries are modular but increase network and processing overhead.
- Depends on the use case and database engine optimization.
- Example: Used one big query with CTEs for generating monthly sales reports efficiently.

27. Explain the difference between Exclusive Lock and Update Lock.

- Exclusive Lock prevents other operations from accessing the resource.
- Update Lock allows reads but prevents multiple updates.
- Update Lock reduces deadlocks compared to Exclusive Lock.
- Example: Used Update Locks to handle inventory updates without impacting read operations.

28. How does a B-trees Index work?

- B-trees organize data hierarchically for efficient retrieval.
- Balanced structure ensures uniform search time across nodes.
- Supports range queries and ordered retrieval.
- Example: Leveraged B-tree indexing to speed up searches on large customer datasets.

29. What is the difference among UNION, MINUS, and INTERSECT?

- UNION combines distinct rows from two queries.
- MINUS retrieves rows in the first query not in the second.
- INTERSECT retrieves rows common to both queries.
- Example: SELECT Name FROM Employee MINUS SELECT Name FROM RetiredEmployees;

30. What are some other types of Indexes (vs B-Trees)?

- Hash Index: Efficient for equality searches.
- Bitmap Index: Optimized for low-cardinality columns.
- Full-text Index: Specialized for textual data and keyword searches.
- Example: Used a Full-text Index for implementing search functionality in a document management system.

31. How does database indexing work?

- Indexing creates a data structure to speed up query retrieval by avoiding full table scans.
- It uses B-trees, hash tables, or other structures to map values to storage locations.
- Indexes improve read performance but can slow down write operations due to maintenance.
- Example: Added an index on the Email column to optimize user login queries.

32. What is the difference between Optimistic Locking and Pessimistic Locking?

- Optimistic Locking assumes minimal conflicts and validates data during updates.
- Pessimistic Locking prevents conflicts by locking data until the transaction completes.
- Optimistic Locking is suitable for high-concurrency systems; Pessimistic for critical updates.
- Example: Used Optimistic Locking for resolving simultaneous edits in a collaborative app.

33. Name some disadvantages of a Hash index.

- Not suitable for range queries.
- Requires extra memory for hash tables.
- Poor performance for highly skewed data distributions.
- Example: Encountered inefficiencies using hash indexing for customer age range searches.

34. What is the difference between B-Tree, R-Tree, and Hash indexing?

- B-Tree supports range queries and ordered data.

- R-Tree is optimized for multi-dimensional data like GIS applications.
- Hash indexing excels in exact match lookups but not ordered retrievals.
- Example: Used B-Tree for product searches and R-Tree for geolocation queries in my app.

35. What is Index Cardinality, and why does it matter?

- Index cardinality refers to the uniqueness of values in a column.
- High cardinality improves index efficiency for searches and lookups.
- Low cardinality may lead to performance issues due to redundancy.
- Example: Indexed high-cardinality columns like CustomerID to speed up retrievals.

36. How to select the first 5 records from a table?

- Use the TOP keyword in SQL Server or LIMIT in MySQL.
- Orders results to ensure consistent output.
- May use FETCH and OFFSET for pagination.
- Example: SELECT TOP 5 * FROM Employee ORDER BY Salary DESC;

37. Find duplicate values in a SQL table.

- Use GROUP BY and HAVING to identify duplicates.
- Aggregate the data to find repeated values.
- Use COUNT to highlight rows with duplicates.
- Example: SELECT Name, COUNT(*) FROM Employee GROUP BY Name HAVING COUNT(*) > 1;

38. How can we transpose a table using SQL (changing rows to columns or vice-versa)?

- Use PIVOT to convert rows to columns.
- Use UNPIVOT to convert columns to rows.
- Requires aggregating data for transformation.
- Example: SELECT * FROM (SELECT Department, Salary FROM Employee) AS SourceTable PIVOT (SUM(Salary) FOR Department IN ([IT], [HR], [Sales])) AS PivotTable;

39. How to generate a row number in SQL without using ROWNUM?

- Use the ROW_NUMBER() function in SQL Server.
- It assigns unique sequential numbers to each row based on a specified order.
- Supports ordering and partitioning.
- Example: SELECT ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNum, Name FROM Employee;

40. What would happen without an Index?

- Full table scans would be required for every query.
- Query performance would degrade with larger datasets.
- Increased CPU and IO utilization for retrievals.
- Example: Optimized a reporting query by adding indexes, reducing runtime from minutes to seconds.

41. Delete duplicate values in a SQL table.

- Use CTE or subqueries to identify duplicates.
- Delete duplicates while retaining the desired rows.
- Use ROW_NUMBER() to flag duplicates.
- Example:

```
WITH CTE AS (
  SELECT Name, ROW_NUMBER() OVER (PARTITION BY Name ORDER BY ID)
AS RowNum
  FROM Employee
)
DELETE FROM CTE WHERE RowNum > 1;
```

42. How do TRUNCATE and DELETE operations affect Identity columns?

- TRUNCATE resets the identity seed to the default value.
- DELETE does not affect the identity seed.
- Use DBCC CHECKIDENT to reset identity manually.
- Example: Used TRUNCATE on a test table to reset identity during data preparation.

43. How can I do an UPDATE statement with a JOIN in SQL?

- Use JOIN in the UPDATE statement to modify data based on related tables.
- Specify the target table and the join condition.
- Include a SET clause to define updates.
- Example:

```
UPDATE e
SET e.Salary = e.Salary * 1.1
FROM Employee e
INNER JOIN Department d ON e.DeptID = d.ID
WHERE d.Name = 'IT';
```

44. Select the first row in each GROUP BY group (greatest-n-per-group problem).

- Use ROW_NUMBER() or RANK() to rank rows within groups.
- Filter rows where the rank equals 1.
- Useful for deduplication or summary queries.
- Example:

```
WITH CTE AS (
  SELECT Name, Department, ROW_NUMBER() OVER (PARTITION BY
Department ORDER BY Salary DESC) AS RowNum
  FROM Employee
)
SELECT * FROM CTE WHERE RowNum = 1;
```

45. How can we efficiently manage indexing in a database?

- Regularly monitor index usage with DMVs or tools.
- Remove unused or redundant indexes.
- Rebuild or reorganize fragmented indexes.
- Example: Scheduled monthly index maintenance to ensure optimal query performance.

1. Is it possible to rename a database? If so, how would you rename the database?

- Yes, it is possible to rename a database in SQL Server.
- You can use the ALTER DATABASE command or the SQL Server Management Studio (SSMS) interface.
- The database must be in a state where no connections are active, and it should not be involved in replication.
- Example:

```
ALTER DATABASE OldDatabaseName MODIFY NAME = NewDatabaseName;
-- I used this command during a database migration to reflect the updated naming convention.
```

2. What is TOP in T-SQL?

- TOP is used to limit the number of rows returned by a query.
- It works with SELECT, INSERT INTO, and DELETE operations.
- You can specify a fixed number or a percentage of rows using TOP (n) or TOP (n) PERCENT.
- Example:

```
SELECT TOP (5) * FROM Employees ORDER BY Salary DESC;
-- I used this to extract the top 5 highest-paid employees from an employee table.
```

3. What are T-SQL Window functions?

- Window functions perform calculations across a set of table rows related to the current row.
- These include ROW_NUMBER, RANK, DENSE_RANK, and NTILE.
- They are often used with the OVER clause for partitioning or ordering.
- Example:

```
SELECT Name, Salary, RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS Rank
FROM Employees;
-- I applied window functions in a performance dashboard to rank employees within departments.
```

4. When should I use a primary key or an index?

- Use a primary key to uniquely identify each row in a table.
- Use an index to improve the performance of queries.
- Primary keys automatically create a unique clustered index unless specified otherwise.
- Example:

```
CREATE INDEX idx_employee_name ON Employees (Name);
-- I created an index to speed up searches on employee names in a large database.
```

5. Could you explain the difference between Primary Key and Unique Index?

- A primary key enforces both uniqueness and the "not null" constraint.
- A unique index ensures uniqueness but allows multiple nulls.
- Each table can have only one primary key but multiple unique indexes.
- Example:

```
-- Primary Key
CREATE TABLE Products (ID INT PRIMARY KEY, Name NVARCHAR(50));

-- Unique Index
CREATE UNIQUE INDEX idx_unique_email ON Customers (Email);
-- I used unique indexes to enforce email uniqueness without making it a primary key.
```

6. What are the three ways that Dynamic SQL can be issued?

- Using EXEC to execute a query string directly.
- Using sp_executesql for parameterized dynamic SQL.
- Generating and running the dynamic query with application logic.
- Example:

```
DECLARE @SQL NVARCHAR(MAX) = 'SELECT * FROM ' + QUOTENAME(@TableName);
EXEC sp_executesql @SQL;
-- I used dynamic SQL to create flexible reports where table names were input dynamically.
```

7. What are the new error handling commands introduced with SQL Server 2005 and beyond?

- TRY...CATCH for structured error handling.
- THROW to re-raise errors with custom messages.
- Enhanced ERROR_NUMBER, ERROR_MESSAGE, and other error functions.
- Example:

```
BEGIN TRY
  INSERT INTO Orders (ID, OrderDate) VALUES (1, NULL);
END TRY
BEGIN CATCH
  PRINT ERROR_MESSAGE();
END CATCH;
-- I implemented TRY...CATCH to log failed transactions into an error log table.
```

8. Name 5 commands that can be used to manipulate text in T-SQL.

- LEN, CHARINDEX, LEFT, RIGHT, and REPLACE are commonly used.
- These commands help in finding, extracting, and replacing text.
- They are vital for ETL tasks and handling unstructured data.
- Example:

```
SELECT REPLACE('SQL Server', 'Server', 'Database') AS ModifiedText;
-- I used these functions in a project for cleaning and formatting customer names.
```

9. What are the two commands to remove all of the data from a table? Are there any implications with the specific commands?

- TRUNCATE and DELETE can remove all rows.
- TRUNCATE is faster and resets identity columns but doesn't log individual row deletions.
- DELETE logs row deletions and can include conditions.
- Example:

```
DELETE FROM Employees WHERE Department = 'HR';
TRUNCATE TABLE TempLogs;
-- I used TRUNCATE to quickly clean up staging tables in a data warehouse project.
```

10. What is a Subquery in T-SQL?

- A subquery is a query nested inside another query.
- It can return scalar, column, or table data.
- Subqueries can be used in SELECT, FROM, or WHERE clauses.
- Example:

```
SELECT Name FROM Employees WHERE DepartmentID = (SELECT ID FROM Departments WHERE Name = 'HR');
-- I used subqueries in a project to filter data based on dynamic criteria.
```

11. What are the differences between SQL and T-SQL?

- SQL is a standard query language; T-SQL is Microsoft's extension of SQL.
- T-SQL includes procedural programming constructs like variables and loops.
- T-SQL supports advanced features like error handling and window functions.
- Example:

```
DECLARE @Message NVARCHAR(50) = 'Hello T-SQL';
PRINT @Message;
-- T-SQL features like variables helped me automate tasks in ETL processes.
```

12. What is the difference between Data Definition Language (DDL) and Data Manipulation Language (DML)?

- DDL deals with schema creation and modification (CREATE, ALTER, DROP).
- DML handles data operations like SELECT, INSERT, UPDATE, and DELETE.

- DDL changes are auto-committed, while DML can be transactional.
- Example:

```
-- DDL
CREATE TABLE Employees (ID INT, Name NVARCHAR(50));

-- DML
INSERT INTO Employees VALUES (1, 'John Doe');
-- I used DDL and DML extensively during schema and data migrations.
```

13. What are the limitations of the IDENTITY column?

- Cannot be updated once set.
- Sequential values can have gaps if transactions fail.
- Only one IDENTITY column per table is allowed.
- Example:

```
CREATE TABLE Orders (OrderID INT IDENTITY(1,1), OrderDate DATE);
-- I faced gaps in IDENTITY columns when handling high transaction volumes and resolved them using SEQUENCE.
```

14. What is Blocking in SQL?

- Blocking occurs when one transaction locks resources, preventing other transactions from accessing them.
- It is temporary and resolved once the locking transaction completes.
- Excessive blocking impacts system performance.
- Example:

```
-- Identify blocking
EXEC sp_who2;
-- I addressed blocking by optimizing queries and reducing transaction time in critical systems.
```

15. What is the OFFSET-FETCH filter in T-SQL?

- Used for pagination by skipping and fetching specific rows.
- Available in SQL Server 2012 and later.
- Works with ORDER BY to ensure predictable results.
- Example:

```
SELECT * FROM Employees ORDER BY Name OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;
-- I used OFFSET-FETCH to implement server-side paging in web applications.
```

16. What's the difference between a Local Temp Table and a Global Temp Table?

- Local temp tables (#TempTable) are visible only within the session that created them.
- Global temp tables (##TempTable) are visible to all sessions but are deleted once all sessions using them are closed.
- Local temp tables are ideal for session-specific operations, while global temp tables work well for shared data between sessions.
- Example:

```
-- Local Temp Table
CREATE TABLE #TempTable (ID INT, Name NVARCHAR(50));
INSERT INTO #TempTable VALUES (1, 'John');

-- Global Temp Table
CREATE TABLE ##TempTable (ID INT, Name NVARCHAR(50));
INSERT INTO ##TempTable VALUES (1, 'John');
-- I used local temp tables during complex data transformations in a stored procedure.
```

17. In what version of SQL Server were synonyms released, what do synonyms do, and when could you make the case for using them?

- Synonyms were introduced in SQL Server 2005.
- They provide an alias for database objects like tables, views, or stored procedures.
- Synonyms simplify cross-database queries and improve code portability.
- Example:

```
CREATE SYNONYM SalesTable FOR [SalesDB].[dbo].[Orders];
SELECT * FROM SalesTable;
-- I used synonyms to simplify queries in applications accessing multiple databases.
```

18. What are bitwise operators and what is the value from a database design perspective?

- Bitwise operators like &, |, ^, and ~ perform operations at the bit level.
- They are used for flags or binary data manipulation.
- They allow compact storage of multiple boolean states.
- Example:

```
SELECT 5 & 3 AS BitwiseAnd, 5 | 3 AS BitwiseOr;
-- I used bitwise operations to store and retrieve user permissions efficiently.
```

19. What are uncommittable transactions?

- A transaction enters an uncommittable state when it encounters an error after the BEGIN TRANSACTION.
- In this state, the transaction cannot be committed but must be rolled back.
- Error handling, like TRY...CATCH, is crucial to manage such cases.
- Example:

```
BEGIN TRANSACTION
UPDATE Orders SET OrderDate = NULL WHERE ID = 1; -- This causes an error
IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
-- I encountered uncommittable transactions during bulk updates and handled them with rollback logic.
```

20. What's the difference between Azure SQL Database and Azure SQL Managed Instance?

- Azure SQL Database is a fully managed PaaS offering for single databases or elastic pools.
- Azure SQL Managed Instance offers near-complete SQL Server compatibility with managed features.
- Managed Instance supports SQL Server Agent and cross-database queries, unlike Azure SQL Database.
- Example:

```
-- Migration scenario
-- I used Azure SQL Database for lightweight applications and Managed Instance for lifting legacy SQL Server workloads.
```

21. What does the T-SQL command IDENT_CURRENT do?

- IDENT_CURRENT returns the last identity value generated for a specified table, regardless of session or scope.
- Unlike @@IDENTITY, it is table-specific.
- Careful use is required in concurrent environments to avoid misleading results.
- Example:

```
SELECT IDENT_CURRENT('Orders') AS LastOrderID;
-- I used this command to track the last inserted ID in audit processes.
```

22. What are the practical differences between COALESCE() and ISNULL() in T-SQL?

- COALESCE() supports multiple arguments and returns the first non-null value.
- ISNULL() accepts only two arguments and returns a replacement for a single null.
- COALESCE() conforms to the ANSI SQL standard, while ISNULL() is T-SQL-specific.
- Example:

```
SELECT COALESCE(NULL, 'Default') AS CoalesceResult, ISNULL(NULL, 'Default') AS IsNullResult;
-- I used COALESCE in reporting to handle multiple fallback values.
```

23. Is there a difference between T-SQL linked server and a synonym?

- A linked server connects SQL Server to external data sources, enabling cross-server queries.
- A synonym is a local alias for database objects and does not connect to external servers.

- Linked servers are for interoperability, while synonyms improve object accessibility.
- Example:

```
-- Linked Server
SELECT * FROM [LinkedServer].[Database].[Schema].[Table];

-- Synonym
CREATE SYNONYM LocalTable FOR
[LinkedServer].[Database].[Schema].[Table];
SELECT * FROM LocalTable;
```

-- *I used linked servers for ETL processes and synonyms for simplifying application queries.*

24. What are the Join Types in T-SQL?

- Inner Join: Retrieves matching rows from both tables.
- Left Join: Retrieves all rows from the left table and matching rows from the right table.
- Right Join: Retrieves all rows from the right table and matching rows from the left table.
- Full Join: Retrieves all rows when there's a match in either table.
- Example:

```
SELECT e.Name, d.Name AS Department
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID = d.ID;
```

-- *I used joins extensively for consolidating relational data in reporting systems.*

25. What are the types of XML indexes in SQL Server?

- Primary XML Index: Required for creating other XML indexes.
- Secondary XML Indexes: Includes PATH, VALUE, and PROPERTY indexes for specific query patterns.
- These indexes improve the performance of XML data retrieval.
- Example:

```
CREATE PRIMARY XML INDEX px_index ON Products(XmlData);
CREATE SECONDARY XML INDEX sx_path ON Products(XmlData) USING
XML INDEX px_index FOR PATH;
```

-- *I applied XML indexes for efficient querying of XML-based configuration data.*

26. What two commands were released in SQL Server 2005 related to comparing data sets from two or more separate SELECT statements?

- EXCEPT: Returns rows from the first query that are not in the second query.
- INTERSECT: Returns rows common to both queries.
- They simplify comparing datasets without using JOIN or NOT IN.
- Example:

```
SELECT Name FROM Employees
EXCEPT
SELECT Name FROM FormerEmployees;

SELECT Name FROM Employees
INTERSECT
SELECT Name FROM ProjectMembers;
```

-- *I used these commands for data reconciliation in migration projects.*

27. What are ROLLUP and CUBE in T-SQL?

- ROLLUP: Generates subtotals and a grand total for a hierarchical group.
- CUBE: Generates subtotals and a grand total for all possible combinations of groups.
- Both are used in GROUP BY for advanced aggregation.
- Example:

```
SELECT Department, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY ROLLUP(Department);
```

-- *I used ROLLUP to create summary reports for hierarchical data structures.*

28. How can you delete duplicate records in a table with no primary key?

- Use ROW_NUMBER() with a CTE to identify duplicates.
- Delete rows where ROW_NUMBER > 1.
- Ensure proper ordering to retain the desired record.
- Example:

```
WITH CTE AS (
  SELECT *, ROW_NUMBER() OVER (PARTITION BY Name ORDER BY ID) AS
RowNum
  FROM Employees
)
DELETE FROM CTE WHERE RowNum > 1;
```

-- *I used this method for cleaning legacy data in a customer table.*

29. How do I perform an IF...THEN in a SQL SELECT statement?

- Use a CASE statement within the SELECT clause.
- It provides conditional logic to transform data dynamically.
- Simplifies conditional column values or derived calculations.
- Example:

```
SELECT Name,
    CASE WHEN Salary > 50000 THEN 'High' ELSE 'Low' END AS
SalaryCategory
FROM Employees;
```

-- *I used CASE for categorizing employee salary ranges in reports.*

30. What are the advantages of using Stored Procedures in SQL Server?

- Precompiled execution improves performance.
- Reusability simplifies code management and reduces redundancy.
- Security enhances by encapsulating business logic.
- Example:

```
CREATE PROCEDURE GetEmployeeDetails
  @EmployeeID INT
AS
BEGIN
  SELECT * FROM Employees WHERE ID = @EmployeeID;
END;
```

-- *I used stored procedures for consistent and secure data access in a multi-tier application.*

31. What's the difference between TRUNCATE and DELETE in SQL?

- TRUNCATE removes all rows from a table and resets identity columns.
- DELETE removes rows one by one and logs each deletion.
- TRUNCATE cannot include a WHERE clause, while DELETE can.
- Example:

```
DELETE FROM Employees WHERE Department = 'HR';
TRUNCATE TABLE TempLogs;
```

-- *I used TRUNCATE to quickly clean up staging tables and DELETE for selective row removal.*

32. What is a Cursor and how does it work?

- A cursor allows row-by-row processing of query results.
- It supports operations like FETCH, UPDATE, or DELETE on individual rows.
- Cursors are less efficient than set-based operations and should be used sparingly.
- Example:

```
DECLARE EmployeeCursor CURSOR FOR SELECT ID, Name FROM
Employees;
OPEN EmployeeCursor;
FETCH NEXT FROM EmployeeCursor INTO @ID, @Name;
```

-- *I used cursors in legacy systems for row-level operations that lacked set-based solutions.*

33. Explain Function vs. Stored Procedure in SQL Server.

- Functions return a value (scalar or table), while procedures execute code without necessarily returning a value.
- Functions are used in SELECT and other expressions; procedures cannot be used like this.

- Functions cannot have side effects like modifying data; procedures can.
- Example:

```
CREATE FUNCTION GetEmployeeSalary(@ID INT) RETURNS INT
AS
BEGIN
   RETURN (SELECT Salary FROM Employees WHERE ID = @ID);
END;
```

-- *I used functions for calculations and procedures for complex business logic.*

34. What are Row Constructors in SQL Server?

- Row constructors allow multiple rows of values to be inserted in a single INSERT statement.
- Useful for quickly inserting data without repeating the column list.
- Supported from SQL Server 2008 onwards.
- Example:

```
INSERT INTO Employees (ID, Name, Salary) VALUES
(1, 'John', 50000),
(2, 'Jane', 60000);
```

-- *I used row constructors to initialize lookup tables with static values.*

35. What is a Linked Server in SQL Server?

- A linked server connects SQL Server to external databases or data sources.
- Enables cross-database queries without data migration.
- Configured using sp_addlinkedserver.
- Example:

```
SELECT * FROM
[LinkedServerName].[DatabaseName].[Schema].[TableName];
```

-- *I used linked servers to integrate data from Oracle and SQL Server systems for a BI project.*

36. What is a Filegroup in SQL Server?

- A filegroup is a logical storage unit for grouping data files.
- It helps manage and allocate storage across multiple physical disks.
- Primary filegroups store metadata, while secondary ones store user data.
- Example:

```
CREATE DATABASE MyDB
ON PRIMARY
(NAME = MyDB_Data, FILENAME = 'C:\MyDBData.mdf')
LOG ON
(NAME = MyDB_Log, FILENAME = 'C:\MyDBLog.ldf');
```

-- *I used filegroups to optimize performance by distributing data across disks.*

37. Is it possible to import data directly from T-SQL commands without using SQL Server Integration Services? If so, what are the commands?

- Yes, you can use commands like BULK INSERT, OPENROWSET, or linked server queries.
- BULK INSERT is efficient for loading large data files.
- OPENROWSET allows querying external files.
- Example:

```
BULK INSERT Employees
FROM 'C:\Data\Employees.csv'
WITH (FIRSTROW = 2, FIELDTERMINATOR = ',', ROWTERMINATOR = '\n');
```

-- *I used BULK INSERT to load large CSV datasets into staging tables.*

38. What is the difference between PARTITION BY and GROUP BY in SQL?

- GROUP BY aggregates rows into groups, reducing the number of rows in the result.
- PARTITION BY provides row-wise aggregation without reducing rows, often used with window functions.
- Both organize data but serve different purposes in aggregation.
- Example:

```
SELECT Name, Salary, RANK() OVER (PARTITION BY Department ORDER BY
Salary DESC) AS Rank
FROM Employees;
```

-- *I used PARTITION BY for generating rank-based reports without losing details.*

39. What do Clustered and Non-Clustered indexes actually mean?

- A clustered index sorts and stores data rows in the table based on the index key.
- A non-clustered index creates a separate structure pointing to table rows.
- Each table can have one clustered index but multiple non-clustered indexes.
- Example:

```
CREATE CLUSTERED INDEX idx_employee_id ON Employees(ID);
CREATE NONCLUSTERED INDEX idx_employee_name ON
Employees(Name);
```

-- *I used a clustered index on primary keys and non-clustered for search columns.*

40. Name some types of Triggers in SQL Server.

- **DML Triggers**: Execute on INSERT, UPDATE, or DELETE.
- **DDL Triggers**: Fire on schema changes like CREATE or ALTER.
- **Logon Triggers**: Execute during user logins.
- Example:

```
CREATE TRIGGER trg_employee_update
ON Employees
AFTER UPDATE
AS
   PRINT 'Employee record updated';
```

-- *I used triggers to log changes in sensitive tables.*

41. What is the difference between EXEC vs sp_executesql in SQL Server?

- EXEC executes a SQL string but does not support parameters.
- sp_executesql supports parameterized queries, reducing SQL injection risks.
- sp_executesql allows query plan reuse, improving performance.
- Example:

```
DECLARE @SQL NVARCHAR(MAX) = 'SELECT * FROM Employees WHERE ID
= @ID';
DECLARE @ID INT = 1;
EXEC sp_executesql @SQL, N'@ID INT', @ID;
```

-- *I used sp_executesql to dynamically query user-specific data securely.*

42. What is the use of GO in Transact SQL?

- GO indicates the end of a batch of T-SQL statements.
- It is not a SQL command but a batch separator in tools like SSMS.
- Statements before GO are executed together, helping structure scripts.
- Example:

```
PRINT 'Batch 1';
GO
PRINT 'Batch 2';
```

-- *I used GO to organize long scripts for database deployment.*

43. Is it correct/best practice to have the TRY/CATCH block inside the transaction or should the transaction be inside the TRY block?

- Best practice is to have the transaction inside the TRY block.
- This ensures proper rollback in case of errors.
- Avoid committing or rolling back outside of TRY/CATCH.
- Example:

```
BEGIN TRY
   BEGIN TRANSACTION
   INSERT INTO Orders (ID, OrderDate) VALUES (1, NULL);
   COMMIT TRANSACTION;
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION;
```

```
  PRINT ERROR_MESSAGE();
END CATCH;
```
-- *I used this approach for ensuring data consistency in critical operations.*

44. What are the best practices for using a GUID as a primary key, specifically regarding performance?

- Use GUIDs as non-clustered indexes to avoid fragmentation.
- Prefer sequential GUIDs (NEWSEQUENTIALID) for clustered indexes.
- Monitor performance impact with large datasets.
- Example:

```
CREATE TABLE Products (ID UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(), Name NVARCHAR(50));
```
-- *I used GUIDs as primary keys in distributed systems requiring global uniqueness.*

45. What is the native system stored procedure to issue a command against all databases?

- Use sp_MSforeachdb to run a command across all databases.
- It simplifies tasks like collecting information or applying updates.
- Example:

```
EXEC sp_MSforeachdb 'USE [?]; SELECT DB_NAME() AS DatabaseName, COUNT(*) FROM sys.tables';
```
-- *I used this procedure to audit table counts across multiple databases.*

46. Why should you never use GUIDs as part of a clustered index?

- GUIDs are non-sequential, leading to high fragmentation in clustered indexes.
- This affects insert performance and storage efficiency.
- Use sequential GUIDs or surrogate keys instead.
- Example:

```
CREATE TABLE Products (ID UNIQUEIDENTIFIER DEFAULT NEWID(), Name NVARCHAR(50));
CREATE NONCLUSTERED INDEX idx_product_id ON Products(ID);
```
-- *I avoided clustered GUID indexes after observing fragmentation issues in large datasets.*

47. From a T-SQL perspective, how would you prevent T-SQL code from running on a production SQL Server?

- Use a check for the server name or environment variable at the beginning of the script.
- Abort execution if the server is production.
- Example:

```
IF @@SERVERNAME = 'ProdServer'
BEGIN
  PRINT 'This script cannot run on production!';
  RETURN;
END;
```
-- *I used this safeguard to prevent accidental deployments on production environments.*

48. How can you capture the length of a column when it is a Text, NText, or Image data type?

- Use the DATALENGTH() function to return the length in bytes of these data types.
- LEN() does not work with TEXT, NTEXT, or IMAGE.
- Consider converting TEXT/NTEXT to VARCHAR or NVARCHAR before using LEN() for string length.
- Example:

```
SELECT DATALENGTH(ColumnName) FROM MyTable;
```
-- *I used DATALENGTH() to analyze large text data in logging systems.*

49. Provide an example of Left Outer Join with Exclusions.

- A LEFT OUTER JOIN returns all rows from the left table and matching rows from the right table; non-matching rows on the right will have NULL.
- To exclude non-matching rows, use WHERE clause filtering.
- Example:

```
SELECT A.Name, B.Address
FROM Customers A
LEFT OUTER JOIN Orders B ON A.CustomerID = B.CustomerID
WHERE B.OrderID IS NULL;
```
-- *I used this query to find customers who have never placed an order.*

50. How do I UPDATE from a SELECT in SQL Server?

- You can perform an UPDATE based on the result of a SELECT query using a JOIN or WHERE clause.
- This is useful for updating rows based on data from another table or query result.
- Example:

```
UPDATE A
SET A.Salary = B.NewSalary
FROM Employees A
JOIN SalaryUpdates B ON A.EmployeeID = B.EmployeeID;
```
-- *I used this technique for updating salaries based on a bulk upload of new salary data.*

51. How do TRUNCATE and DELETE operations affect Identity columns?

- TRUNCATE resets the identity column to its seed value.
- DELETE does not affect the identity column and continues the count from the last inserted value.
- Be cautious when using TRUNCATE as it may affect auto-increment behavior.
- Example:

```
TRUNCATE TABLE Products;
```
-- *I used TRUNCATE when I needed to clear tables and reset identity values in staging environments.*

52. How to insert the results of a stored procedure into a temporary table?

- You can insert the result of a stored procedure into a temporary table using INSERT INTO #TempTable.
- Ensure the stored procedure returns a result set.
- Example:

```
CREATE TABLE #TempResults (ID INT, Name NVARCHAR(50));
INSERT INTO #TempResults
EXEC GetEmployeeDetails;
```
-- *I used this approach to capture procedure results for further analysis without re-running the procedure.*

1. What is the difference between .ts and .tsx extensions in TypeScript?

- .ts files are used for standard TypeScript code without JSX syntax.
- .tsx files support TypeScript code with JSX, enabling React component development.
- JSX stands for JavaScript XML, used for defining React elements and components.
- Example: In a React project, .tsx is used to define components:

```
const Button: React.FC = () => <button>Click Me</button>;
```

---

2. Do we need to compile TypeScript files and why?

- TypeScript must be compiled because browsers understand only JavaScript.
- The TypeScript compiler converts .ts files into plain JavaScript.
- Compilation helps catch type-related errors during development.
- Example: While creating a Node.js server, I compiled .ts files using tsc:

```
tsc server.ts
```

---

3. What are the benefits of TypeScript?

- Provides static typing, reducing runtime errors.
- Supports modern JavaScript features, making code maintainable.
- Enhances developer productivity with autocompletion and tooling.
- Example: Static typing helped me avoid null issues in a React app:

```
const greet = (name: string): string => `Hello, ${name}`;
```

---

4. What is TypeScript, and why would I use it in place of JavaScript?

- TypeScript is a superset of JavaScript that adds static typing.
- It improves code quality by detecting issues at compile time.
- Enhances teamwork through clear contracts and documentation.
- Example: I migrated a JS codebase to TypeScript for better type safety:

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

---

5. How to call a base class constructor from a child class in TypeScript?

- Use the super keyword to invoke the base class constructor.
- Ensure super is called before accessing this in the child class.
- Pass necessary arguments to the super function as needed.
- Example: I used super to extend a user class in an auth module:

```
class User {
  constructor(public name: string) {}
}

class Admin extends User {
  constructor(name: string, public adminLevel: number) {
    super(name);
  }
}
```

---

6. What is TypeScript, and why do we need it?

- TypeScript adds strong typing to JavaScript for better error detection.
- It supports ES6+ features and compiles them to ES5 for browser compatibility.
- Boosts productivity with features like interfaces, modules, and type inference.
- Example: Using interfaces improved consistency in a team project:

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

---

7. What is TypeScript, and why should one use it?

- TypeScript provides advanced IDE support, catching errors at compile time.
- Helps manage complex projects with type annotations and strict rules.
- Bridges the gap between JavaScript's flexibility and robust programming practices.
- Example: I implemented TypeScript in a REST API to enforce typing for request bodies:

```
interface RequestBody {
  username: string;
  password: string;
}
```

---

8. How to perform string interpolation in TypeScript?

- Use template literals with backticks (`).
- Embed expressions inside ${} within template literals.
- Supports multiline strings and dynamic content seamlessly.
- Example: I used string interpolation for dynamic greeting messages:

```
const greet = (name: string) => `Hello, ${name}!`;
```

---

9. What are Modules in TypeScript?

- Modules encapsulate code into reusable and manageable units.
- TypeScript supports ES6 modules with import and export syntax.
- Helps avoid global scope pollution and simplifies dependency management.
- Example: I used modules to organize components in an Angular project:

```
export const greet = (name: string) => `Hello, ${name}`;
import { greet } from './greet';
```

---

10. Explain generics in TypeScript.

- Generics provide a way to write reusable, type-safe functions, or classes.
- They allow specifying types at the time of usage, maintaining flexibility.
- Useful for functions or data structures like arrays, maps, etc.
- Example: I used generics in a utility function to type arrays dynamically:

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

---

11. List the built-in types in TypeScript.

- TypeScript includes number, string, boolean, null, undefined, and more.
- Advanced types: any, unknown, never, void, and object.
- Supports array and tuple types for collection management.
- Example: I used tuple types to represent fixed-length arrays in a project:

```
let userInfo: [string, number] = ["John", 25];
```

12. What is Optional Chaining in TypeScript?

- Optional chaining (?.) safely accesses properties on nullish values.
- Avoids runtime errors when accessing nested object properties.
- Returns undefined instead of throwing an error for nullish objects.
- Example: I used optional chaining to check for nested data in an API response:

```
const userCity = user?.address?.city;
```

13. How can we use optional chaining in TypeScript?

- Use ?. to safely access object properties, methods, or array elements.
- Combine with nullish coalescing (??) to provide fallback values.
- Reduces the need for repetitive null checks.
- Example: Optional chaining prevented crashes in dynamic data handling:

```
const zipCode = user?.address?.postalCode ?? 'Not Available';
```

14. How to make arrays that can only be specific types in TypeScript?

- Use type annotations with arrays like string[] or Array<number>.
- Employ union types for arrays that accept multiple types.
- Leverage tuple types for arrays with fixed-length and specific types.
- Example: I used a tuple type to enforce structure in a data-processing task:

```
let response: [number, string] = [200, "OK"];
```

15. Describe what conditional types are in TypeScript.

- Conditional types allow type selection based on conditions.
- They follow the T extends U ? X : Y syntax for type evaluation.
- Useful for creating flexible and reusable types.
- Example: I implemented conditional types for narrowing based on input:

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

16. What does the pipe symbol mean in TypeScript?

- The pipe symbol (|) is used to define union types.
- Union types allow a variable to hold multiple possible types.
- It ensures flexibility while maintaining type safety.
- Example: I used a union type for a function parameter to accept string or number:

```
function format(input: string | number): string {
  return input.toString();
}
```

17. How do we create an enum with string values in TypeScript?

- Use the enum keyword with string assignments for each member.
- String enums allow descriptive and readable values.
- Access enum members via their names or string values.
- Example: I used a string enum to define API status responses:

```
enum Status {
  SUCCESS = "Success",
  ERROR = "Error",
  PENDING = "Pending"
}
```

18. What is the difference between types String and string in TypeScript?

- string is a primitive type representing text.
- String is an object type, which wraps the primitive type.
- Prefer string for type annotations to avoid unnecessary overhead.
- Example: I consistently used string in my project for simplicity:

```
let name: string = "John Doe";
```

19. What is a TypeScript Map file?

- A Map file links TypeScript code to its JavaScript output for debugging.
- It allows developers to trace errors in the original TypeScript code.
- Generated with the sourceMap compiler option in tsconfig.json.
- Example: I enabled sourceMap to debug issues in a compiled project:

```
{
  "compilerOptions": {
    "sourceMap": true
  }
}
```

20. What is the purpose of the Nullish Coalescing operator in TypeScript?

- The ?? operator provides a default value for null or undefined.
- It prevents false positives with falsy values like 0 or an empty string.
- Combines well with optional chaining for clean error handling.
- Example: I used ?? to set a default username in a form:

```
const username = userInput ?? "Guest";
```

21. What are assertion functions in TypeScript?

- Assertion functions ensure certain conditions are met during runtime.
- Use asserts to refine the type within specific code paths.
- Helpful for narrowing down complex types.
- Example: I used assertion functions to validate API response structures:

```
function assertIsString(value: any): asserts value is string {
  if (typeof value !== "string") {
    throw new Error("Not a string!");
  }
}
```

22. Which access modifiers are implied when not specified in TypeScript?

- Members without explicit modifiers are considered public.
- Public members are accessible anywhere.
- private and protected restrict access to specific scopes.
- Example: I relied on default public access for a utility class:

```
class Utils {
  calculateSum(a: number, b: number): number {
    return a + b;
  }
}
```

23. What is Type Erasure in TypeScript?

- Type erasure removes type annotations during compilation to JavaScript.

- Ensures TypeScript's type safety doesn't impact runtime performance.
- Enables compatibility with plain JavaScript environments.
- Example: Type annotations in this function are erased in the output:

```
function greet(name: string): string {
  return `Hello, ${name}`;
}
```

24. What is the difference between Classes and Interfaces in TypeScript?

- Classes define behavior and implementation; interfaces specify structure.
- Interfaces cannot contain implementation logic, only type definitions.
- Classes support inheritance, while interfaces allow multiple type extensions.
- Example: I used interfaces to enforce structure in class implementation:

```
interface IUser {
  id: number;
  name: string;
}

class User implements IUser {
  constructor(public id: number, public name: string) {}
}
```

25. What are Decorators in TypeScript?

- Decorators are special functions used to modify classes or methods.
- They are applied using the @ syntax before class or method declarations.
- Requires enabling the experimentalDecorators compiler option.
- Example: I used a decorator for logging method calls in a service class:

```
function Log(target: any, propertyName: string, descriptor:
PropertyDescriptor) {
  const originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`Method ${propertyName} called with arguments:`, args);
    return originalMethod.apply(this, args);
  };
}

class Service {
  @Log
  fetchData() {
    return "Data fetched";
  }
}
```

26. How could you check for null and undefined in TypeScript?

- Use strict equality (===) to differentiate between null and undefined.
- Combine nullish coalescing (??) with conditional checks for fallback values.
- TypeScript's strictNullChecks flag enhances null/undefined type safety.
- Example: I implemented a utility function to validate input values:

```
function validateInput(value: any): boolean {
  return value !== null && value !== undefined;
}
```

27. Could we use TypeScript on the backend, and how?

- Yes, TypeScript works seamlessly with Node.js and other backend frameworks.
- Helps build type-safe APIs and maintainable backend codebases.
- Requires compilation to JavaScript using the TypeScript compiler.
- Example: I created a REST API using TypeScript with Express:

```
import express, { Request, Response } from 'express';
const app = express();
app.get('/api', (req: Request, res: Response) => res.send('Hello,
TypeScript!'));
app.listen(3000);
```

28. What are the differences between TypeScript and JavaScript?

- TypeScript adds static typing; JavaScript is dynamically typed.
- TypeScript compiles to JavaScript; JavaScript runs directly in browsers.
- TypeScript supports advanced features like interfaces and generics.
- Example: I migrated a JS codebase to TS for enhanced maintainability:

```
let isActive: boolean = true; // Static typing
```

29. What is an Interface in TypeScript?

- Interfaces define the structure of objects, ensuring consistent properties.
- They are purely compile-time constructs with no runtime output.
- Supports optional and readonly properties for better flexibility.
- Example: I used an interface for type-safe API responses:

```
interface ApiResponse {
  data: string;
  status: number;
}
```

30. Does TypeScript support all object-oriented principles?

- TypeScript supports encapsulation, inheritance, polymorphism, and abstraction.
- Classes, interfaces, and access modifiers implement these principles.
- Enables robust OOP design patterns in JavaScript projects.
- Example: I used OOP principles to design a library management system:

```
class Book {
  constructor(public title: string, public author: string) {}
}
```

31. How to implement class constants in TypeScript?

- Use the readonly modifier for class fields to make them immutable.
- Define constants directly inside the class or as static readonly for shared access.
- TypeScript enforces immutability at compile time for readonly fields.

- Example: I used readonly to define a constant in a configuration class:

```
class Config {
  static readonly APP_NAME = "MyApp";
}
console.log(Config.APP_NAME); // MyApp
```

32. When to use interfaces and when to use classes in TypeScript?

- Use interfaces to define structures without implementation logic.
- Use classes when defining both structure and behavior.
- Interfaces are ideal for contracts, while classes encapsulate functionality.
- Example: I used interfaces for API types and classes for service logic:

```
interface IUser {
  id: number;
  name: string;
}

class UserService {
  getUser(id: number): IUser {
    return { id, name: "John Doe" };
  }
}
```

33. What is the purpose of getters/setters in TypeScript?

- Getters and setters control access to class properties.
- They provide a mechanism to encapsulate data and validate inputs.
- Ensures separation of concerns by managing logic inside accessors.
- Example: I used getters/setters to manage user details securely:

```
class User {
  private _name: string = "";

  get name(): string {
    return this._name;
  }

  set name(value: string) {
    if (!value) throw new Error("Invalid name");
    this._name = value;
  }
}
```

34. Which object-oriented terms are supported by TypeScript?

- TypeScript supports inheritance, polymorphism, encapsulation, and abstraction.
- Implements these through classes, interfaces, and access modifiers.
- Enhances OOP practices with optional static typing.
- Example: I used inheritance and polymorphism in a vehicle system:

```
class Vehicle {
  drive(): string {
    return "Driving";
  }
}

class Car extends Vehicle {
  drive(): string {
    return "Car driving";
  }
}
```

35. What are the use cases for a const assertion in TypeScript?

- Prevents type widening by making a value immutable and its type literal.
- Ensures stricter type checking with fixed values.
- Useful for immutable configuration objects or literal-based enums.
- Example: I used const assertion to fix an array type:

```
const roles = ["Admin", "User", "Guest"] as const;
type Role = typeof roles[number]; // "Admin" | "User" | "Guest"
```

36. What are some use cases of template literal types in TypeScript?

- Enables dynamic string types based on existing literal types.
- Useful for creating string-based patterns or configurations.
- Commonly used in utility types and API request definitions.
- Example: I defined dynamic route parameters with template literal types:

```
type Route = `/user/${string}`;
const userProfile: Route = "/user/123";
```

37. What is a Mixin Class in TypeScript?

- Mixins allow combining multiple behaviors into a single class.
- They provide a flexible way to reuse functionality without traditional inheritance.
- Implemented using functions that extend base classes.
- Example: I used mixins for reusable logging functionality:

```
class Logger {
  log(message: string) {
    console.log(message);
  }
}

function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach((baseCtor) => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach((name) => {
      derivedCtor.prototype[name] = baseCtor.prototype[name];
    });
  });
}

class App {}
applyMixins(App, [Logger]);
```

38. List a few rules of private fields in TypeScript.

- Private fields start with # and are truly private to the class.
- Cannot be accessed or modified outside their containing class.
- Different from private, as they are not accessible even via prototype.
- Example: I used private fields to enforce strict encapsulation:

```
class User {
  #password: string;

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

39. How to choose between never, unknown, and any types in TypeScript?

- Use never for unreachable code or impossible states.
- Use unknown for values of an uncertain type, requiring runtime checks.
- Use any when type safety is not a concern (not recommended).
- Example: I used unknown for runtime type checking in a utility function:

```
function process(value: unknown): void {
  if (typeof value === "string") {
    console.log(value.toUpperCase());
  }
}
```

40. Explain how and why we could use property decorators in TypeScript.

- Property decorators modify or annotate class properties.
- Useful for metadata generation, validation, or dependency injection.
- Requires enabling experimentalDecorators in tsconfig.json.
- Example: I used a decorator to validate property values:

```
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,
    });
  };
}

class User {
  @MinLength(5)
  username: string = "";
}
```

41. What does Short-Circuiting mean in TypeScript?

- Short-circuiting stops further evaluation if a condition is already resolved.
- Common with logical operators like && and ||.
- Useful for performance optimization and conditional execution.
- Example: I used short-circuiting to set default values in a function:

```
const getValue = (value?: string): string => value || "Default";
```

42. What is the unique symbol used for in TypeScript?

- The unique symbol ensures a symbol is globally unique and immutable.
- Used for creating strongly typed properties or constants.
- Prevents accidental name clashes in large applications.
- Example: I used unique symbol to define private keys in an API:

```
const UNIQUE_KEY: unique symbol = Symbol("UNIQUE_KEY");
```

43. How to make a readonly tuple type in TypeScript?

- Use readonly before tuple types for immutability.
- Prevents modification of tuple elements after initialization.
- Enhances type safety for fixed-length, immutable arrays.
- Example: I used readonly tuples for a configuration array:

```
const settings: readonly [string, number] = ["Theme", 1];
```

44. What is the fundamental difference between Optional Chaining and Non-null assertion operator in TypeScript?

- Optional chaining (?.) safely accesses properties, returning undefined if nullish.
- Non-null assertion (!) explicitly tells TypeScript a value is non-null.
- Optional chaining avoids errors, while non-null assertion risks runtime failures.
- Example: I used optional chaining for accessing API data safely:

```
const city = user?.address?.city;
```

45. Explain Project References and its benefits in TypeScript.

- Project references enable modular TypeScript project compilation.
- Facilitates faster builds by reusing compiled outputs of referenced projects.
- Encourages code reusability and better project structure.
- Example: I used project references in a monorepo to manage dependencies:

```
{
  "references": [{ "path": "./shared" }]
}
```

46. How to check the type of a variable or constant in TypeScript?

- Use the typeof operator to check primitive types at runtime.
- For custom types, use instanceof for class instances.
- Type guards and user-defined type predicates can also be used.
- Example: I implemented type checks for a flexible utility function:

```
function process(input: unknown) {
  if (typeof input === "string") {
    console.log(input.toUpperCase());
  } else if (input instanceof Array) {
    console.log(input.length);
  }
}
```

47. How TypeScript is an optionally statically typed language?

- TypeScript allows optional typing, meaning variables can be untyped (any).
- Developers can use static types where necessary or avoid them entirely.
- This flexibility helps in gradual migration from JavaScript.
- Example: I used optional typing for a dynamic configuration loader:

```
let config: any = {};
config = { mode: "production" }; // Valid due to dynamic typing
```

48. What is the default access modifier for members of a class in TypeScript?

- The default access modifier for class members is public.
- Members are accessible from any part of the program unless specified otherwise.
- Explicitly specifying modifiers is a best practice for clarity.
- Example: I relied on the default public behavior for a shared utility class:

```
class User {
  name: string; // public by default
  constructor(name: string) {
    this.name = name;
  }
}
```

49. What are the different components of TypeScript?

- **Type System**: Provides static typing to catch errors at compile time.
- **Compiler (tsc)**: Transpiles TypeScript into JavaScript.
- **Language Features**: Includes OOP features like classes, interfaces, generics.
- Example: I used TypeScript components to build a maintainable enterprise app.

50. How to use external plain JavaScript libraries in TypeScript?

- Install type definitions using @types package if available.
- Use the declare keyword to define types manually for unsupported libraries.
- Import the library and use it as per its defined API.
- Example: I integrated a JavaScript charting library with TypeScript:

```
import Chart from "chart.js";

const ctx = document.getElementById("myChart") as
HTMLCanvasElement;
const chart = new Chart(ctx, { type: "bar", data: { labels: [], datasets: [] } });
```

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

- Type can alias primitive, union, intersection, or tuple types.
- Interface is used to define object shapes and is extensible.
- Interfaces are more commonly used for object types.
- Example: I used a type for union types and an interface for object contracts:

```
type ID = string | number;
interface User {
  id: ID;
  name: string;
}
```

52. How to add types to an interface from another interface or extend types in TypeScript?

- Use the extends keyword to inherit from another interface.
- This enables reusability and modular design in type definitions.
- Combine multiple interfaces for composite types.
- Example: I extended interfaces to define related models:

```
interface Person {
  name: string;
}

interface Employee extends Person {
  employeeId: number;
}
```

53. Does TypeScript support function overloading?

- Yes, TypeScript allows function overloading with multiple type signatures.
- The implementation must match one of the declared overloads.
- Useful for defining functions with multiple valid input/output types.
- Example: I used overloading for a data processing utility:

```
function process(value: string): string;
function process(value: number): number;
function process(value: any): any {
  return typeof value === "string" ? value.toUpperCase() : value * 2;
}
```

54. What is the difference between Private and Protected variables in TypeScript?

- Private variables are accessible only within the class they are defined.
- Protected variables are accessible in the class and its subclasses.
- Protected is useful for extending functionality while keeping some scope restricted.
- Example: I used protected fields for shared behavior in derived classes:

```
class Base {
  protected id: number;
  constructor(id: number) {
    this.id = id;
  }
}

class Derived extends Base {
  displayId() {
    console.log(this.id);
  }
}
```

55. What is Typings in TypeScript?

- Typings are definition files (.d.ts) that describe the types in libraries.
- Allow TypeScript to understand JavaScript libraries during compilation.
- Available through @types packages or manually written.
- Example: I added typings for a legacy library in a project:

```
declare module "legacy-lib" {
  export function legacyMethod(): void;
}
```

56. What is the difference between enum and const enum in TypeScript?

- Enum is fully compiled to JavaScript and can be used dynamically.
- Const enum is inlined at compile time, reducing runtime overhead.
- Use const enum for performance-critical applications.
- Example: I used const enum to improve performance in a mapping function:

```
const enum Direction {
  Up,
  Down,
}
console.log(Direction.Up); // Compiles to 0
```

57. Why do we need to use the abstract keyword for classes and their methods in TypeScript?

- Abstract classes define shared behavior but cannot be instantiated directly.
- Abstract methods must be implemented in derived classes.
- Used to enforce structure while allowing flexibility.
- Example: I used an abstract class for a shared vehicle interface:

```
abstract class Vehicle {
  abstract drive(): void;
}

class Car extends Vehicle {
  drive() {
    console.log("Driving a car");
  }
}
```

58. What is Structural Typing in TypeScript?

- Structural typing focuses on the shape of an object rather than its explicit type.
- Enables compatibility between objects with matching structures.
- Used for flexible and duck-typing-friendly designs.
- Example: I relied on structural typing for third-party API data handling:

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

let pt: Point = { x: 10, y: 20 };
```

59. How can you allow classes defined in a module to be accessible outside of the module?

- Use the export keyword to make classes accessible outside the module.
- Import the class where needed using import statements.
- Encapsulates functionality while enabling modularity.
- Example: I exported a utility class for use in multiple modules:

```
export class Helper {
  static greet() {
    console.log("Hello!");
  }
}
```

60. Explain what is Currying in TypeScript.

- Currying transforms a function with multiple arguments into a series of unary functions.
- Helps in creating reusable, partial applications.
- Common in functional programming and cleaner callback management.
- Example: I used currying to create flexible query builders:

```
function multiply(a: number) {
  return (b: number) => a * b;
}

const double = multiply(2);
console.log(double(5)); // 10
```

61. How to exclude a property from a type in TypeScript?

- Use the Omit utility type to exclude specific properties.
- Pass the base type and the property key(s) to Omit.
- This creates a new type without the excluded property.
- Example: I excluded sensitive fields from a user data model:

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

type PublicUser = Omit<User, "password">;

const user: PublicUser = { id: 1, name: "Alice" }; // Valid
```

62. How to define a TypeScript class with an index signature?

- Use an index signature to define a class that allows dynamic properties.
- Specify the property key type and value type in the index signature.
- Ensure other members align with the dynamic property definition.
- Example: I created a class for flexible configuration storage:

```
class Config {
  [key: string]: string;
  appName = "MyApp";
}

const config = new Config();
config.theme = "dark";
```

63. Why do we need Index Signature in TypeScript?

- Index signatures allow defining types for dynamic object keys.
- Useful when the object structure isn't fixed or keys are runtime-defined.
- Enforces type safety for dynamic properties.
- Example: I used index signatures for a translation dictionary:

```
interface Translation {
  [key: string]: string;
}

const en: Translation = { hello: "Hello", bye: "Goodbye" };
```

64. What is the difference between unknown and any type in TypeScript?

- unknown is a safer version of any and requires type assertions or checks before usage.
- any bypasses type checking entirely, potentially causing runtime errors.
- Use unknown for uncertain types to enforce type safety.
- Example: I used unknown to validate API response types:

```
function process(data: unknown) {
  if (typeof data === "string") {
    console.log(data.toUpperCase());
  }
}
```

65. Why is the infer keyword needed in TypeScript?

- infer is used in conditional types to infer types based on a condition.
- Enables extraction or manipulation of types at compile time.
- Simplifies complex type computations in generic scenarios.
- Example: I used infer to extract the return type of a function:

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

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

type ValueType = ReturnType<typeof getValue>; // string
```

66. Explain what is the never datatype in TypeScript.

- never represents values that never occur (e.g., function throws or infinite loops).
- It's used for exhaustive checks in conditional types or switch cases.
- Helps enforce complete case handling in logic.
- Example: I used never to ensure exhaustive case handling:

```
function handle(value: "a" | "b") {
  switch (value) {
    case "a":
      console.log("A");
      break;
    case "b":
      console.log("B");
      break;
    default:
      const _exhaustive: never = value;
  }
}
```

67. What is dynamic import expression in TypeScript?

- Dynamic import loads modules asynchronously using import().
- Useful for code-splitting and lazy-loading in applications.
- Returns a promise that resolves to the module.
- Example: I used dynamic import for loading a configuration module:

```
async function loadConfig() {
  const config = await import("./config");
  console.log(config.default);
}
loadConfig();
```

68. What is the difference between interface and type statements in TypeScript?

- Interface is mainly for defining object shapes and can be extended.
- Type is more flexible, supporting unions, intersections, and primitives.
- Interfaces are better for class-based designs.
- Example: I combined both for a flexible type system:

```
interface Person {
  name: string;
}

type PersonWithAge = Person & { age: number };
```

69. What is Mixin Constructor Type in TypeScript?

- A mixin constructor type combines multiple classes into one.
- Helps in achieving reusable behavior across classes.
- Uses generics to extend a base constructor.
- Example: I used mixins for a feature-rich component class:

```
type Constructor<T = {}> = new (...args: any[]) => T;

function Timestamped<T extends Constructor>(Base: T) {
  return class extends Base {
    timestamp = Date.now();
  };
}

class Entity {}
const TimestampedEntity = Timestamped(Entity);
```

70. How does the override keyword work in TypeScript?

- The override keyword ensures the method overrides a base class method.
- Prevents accidental method shadowing when the base method doesn't exist.
- Increases code clarity and avoids runtime errors.
- Example: I used override to override a base logging method:

```
class Base {
  log() {
    console.log("Base log");
  }
}

class Derived extends Base {
  override log() {
    console.log("Derived log");
  }
}
```

71. Explain when to use the declare keyword in TypeScript.

- Use declare to describe types or variables defined elsewhere (e.g., global scope).
- It avoids TypeScript compilation errors for external or global declarations.
- Commonly used for type definitions of external libraries.
- Example: I used declare for a global configuration object:

```
declare const CONFIG: { apiUrl: string };
console.log(CONFIG.apiUrl);
```

72. Is it possible to generate TypeScript declaration files from a JavaScript library?

- Yes, use the --declaration flag with tsc to generate .d.ts files.
- This helps TypeScript users consume the library with type support.
- Combine it with --allowJs for JavaScript-based projects.
- Example: I generated typings for a shared utility library:

```
tsc --declaration --allowJs --emitDeclarationOnly
```

73. What does the tsconfig option lib do?

- The lib option specifies the TypeScript standard libraries to include.
- It helps restrict or expand the available built-in APIs.
- Useful for targeting specific environments like ES5 or DOM.
- Example: I configured lib to include ES2020 and DOM APIs:

```
{
  "compilerOptions": {
    "lib": ["ES2020", "DOM"]
  }
}
```

74. How to make a union type from a type alias or interface properties in TypeScript?

- Use keyof and Extract to create a union type from properties.
- Helps in dynamically deriving types based on an interface or alias.
- Useful for utility types or type-safe operations.
- Example: I extracted keys for a validation utility:

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

type UserKeys = keyof User; // "id" | "name"
```

75. What are Ambients in TypeScript and when to use them?

- Ambients declare global types, variables, or modules in .d.ts files.
- Used for external libraries or APIs without native TypeScript support.
- Helps in integrating TypeScript with JavaScript ecosystems.
- Example: I defined ambient types for a third-party analytics script:

```
declare module "analytics" {
  export function track(event: string): void;
}
```

76. What is the benefit of import assertions features in TypeScript?

- Import assertions validate the type of imported modules (e.g., JSON, CSS).
- Prevents runtime errors by enforcing expected module formats.
- Supports modern workflows like JSON modules or WebAssembly.
- Example: I used import assertions to load a JSON configuration:

```
import config from "./config.json" assert { type: "json" };
console.log(config.apiUrl);
```

77. What is one thing you would change about TypeScript?

- Enhance type inference for deeply nested object types.
- Provide better tooling for debugging complex types in large codebases.
- Include built-in support for runtime type validation.
- Example: While working with deeply nested APIs, explicit types were often cumbersome.

78. Explain the difference between declare enum vs declare const enum.

- declare enum defines an external enum in ambient declarations.
- declare const enum creates optimized enums by inlining values.
- Use declare const enum for performance-sensitive scenarios.
- Example: I used declare const enum for configuration keys:

```
declare const enum Config {
  BaseUrl = "https://api.example.com"
}

const url = Config.BaseUrl; // Inlined as string
```

79. What are the differences between the private keyword and private fields in TypeScript?

- private keyword enforces access only within the class or subclasses (TypeScript-specific).
- #private fields are part of JavaScript and accessible only within the declaring class.
- #private ensures runtime-level encapsulation.
- Example: I used #private for secure internal state handling:

```
class Secure {
  #token: string = "secret";

  getToken() {
    return this.#token;
  }
}
```

80. How the never datatype can be useful in TypeScript?

- Represents functions or expressions that never produce a value.
- Ensures exhaustive checks in conditional types or control flows.
- Helps catch unhandled cases at compile time.
- Example: I used never to ensure no unhandled enum cases:

```
type Colors = "Red" | "Blue";

function getColorName(color: Colors): string {
  switch (color) {
    case "Red":
      return "Red Color";
    case "Blue":
      return "Blue Color";
    default:
      const exhaustiveCheck: never = color;
      return exhaustiveCheck;
  }
}
```

81. What is the need for the incremental flag in TypeScript?

- The --incremental flag enables faster compilation by caching.
- It compiles only changed files, improving development efficiency.
- Creates a .tsbuildinfo file to store metadata for reuse.
- Example: I enabled it for a large project with frequent changes:

```
{
  "compilerOptions": {
    "incremental": true
  }
}
```

82. Is there a way to check for both null and undefined in TypeScript?

- Use == or === comparisons for strict or loose checks.
- Combine null and undefined in a single comparison using x == null.
- Optional chaining and nullish coalescing simplify handling.
- Example: I checked for null/undefined during data validation:

```
function process(data?: string | null) {
  if (data == null) {
    console.log("Data is null or undefined");
  }
}
```

83. How to make an array with a specific length or elements in TypeScript?

- Use Array constructor or tuple types for specific lengths.
- Define exact element types for precise constraints.
- Combine generics for flexible yet strict array types.
- Example: I defined a tuple for fixed-length RGB values:

```
type RGB = [number, number, number];
const color: RGB = [255, 0, 0];
```

84. What's wrong with that code?

- Analyze issues like type mismatches, missing return types, or unsafe operations.
- Debug by reviewing errors or enabling strict compiler options.
- Use TypeScript tooling (e.g., tsc or IDE) for detailed diagnostics.
- Example: I debugged an incorrect return type in a utility function:

```
function add(a: number, b: number): string {
  return a + b; // Error: Type 'number' is not assignable to type 'string'
}
```

85. Are strongly-typed functions as parameters possible in TypeScript?

- Yes, use function types or interfaces to define parameter signatures.
- Enforce input and return types for type safety.
- Useful for callbacks, event handlers, or higher-order functions.
- Example: I used a strongly-typed callback in a utility:

```
type Callback = (value: number) => void;

function process(callback: Callback) {
  callback(42);
}
```

86. Is that TypeScript code valid? Explain why.

- Validate code against TypeScript rules like strict typing and access modifiers.
- Check for compliance with compiler options in tsconfig.json.
- Ensure compatibility with TypeScript's type system.
- Example: I debugged invalid parameter types in a function:

```
function multiply(a: string, b: string): number {
  return parseInt(a) * parseInt(b); // Fix: Ensure inputs are numeric
}
```

87. What will be the result of this code execution?

- Analyze behavior based on runtime and type system rules.
- Understand how TypeScript translates to JavaScript for execution.
- Predict results considering strict mode and implicit conversions.
- Example: I evaluated a conditional operation in TypeScript:

```
const value = undefined ?? "default"; // Result: "default"
```

88. In the expression a?.b.c, if a.b is null or undefined, will a.b.c evaluate to undefined?

- No, if a?.b evaluates to null or undefined, a?.b.c will not execute.
- Optional chaining halts further property access on null or undefined.
- Avoids runtime errors by safely accessing nested properties.
- Example: I used this for deep API response validation:

```
const response = { user: null };
console.log(response?.user?.name); // undefined
```

89. What does the const assertion mean in TypeScript?

- const assertion freezes the type as literal and prevents widening.
- Useful for defining immutable values.
- Simplifies narrowing of inferred types.
- Example: I used const assertion for exact API responses:

```
const config = {
  apiUrl: "https://example.com",
} as const;

config.apiUrl = "newUrl"; // Error: Cannot assign to 'apiUrl'
```

90. Explain why that code is marked as WRONG?

- Identify issues like mismatched types, incorrect scopes, or access violations.
- Ensure adherence to strict typing and declared members.
- Example: I corrected wrong variable access in a nested function:

```
function outer() {
  let count = 0;
  function inner() {
    count++; // Correct: Ensure access to `count` is valid
  }
}
```

91. How would you overload a class constructor in TypeScript?

- Define multiple constructor signatures for flexibility.
- Use a single implementation to handle different parameter combinations.
- Employ if or switch to process arguments based on type or count.
- Example: I implemented constructor overloading for a User class:

```
class User {
  name: string;
  age: number;

  constructor(name: string);
  constructor(name: string, age: number);
  constructor(name: string, age?: number) {
    this.name = name;
    this.age = age ?? 0;
  }
}

const user1 = new User("Alice");
const user2 = new User("Bob", 30);
```

92. What are the use cases for the keyof operator in TypeScript?

- Retrieves keys of a type as a union of string literals.
- Useful for creating generic functions that operate on object keys.
- Enables type-safe property access and key validation.
- Example: I used keyof for a reusable utility function:

```
type User = { name: string; age: number };
type UserKeys = keyof User; // "name" | "age"

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

const user: User = { name: "Alice", age: 30 };
```

```
console.log(getProperty(user, "name")); // Alice
```

93. How to use mapped types in TypeScript?

- Mapped types transform properties of an existing type.
- Apply operations like making properties optional, readonly, or modifying values.
- Use keyof and indexed access to iterate over keys.
- Example: I created a utility type for partial objects:

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};

type User = { name: string; age: number };
type PartialUser = Partial<User>;

const user: PartialUser = { name: "Alice" };
```

94. What are type guards in TypeScript?

- Type guards narrow types using conditions or functions.
- Built-in guards include typeof and instanceof.
- Custom guards use functions returning arg is Type.
- Example: I implemented a type guard for API responses:

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

function printLength(input: unknown) {
  if (isString(input)) {
    console.log(input.length);
  }
}
```

95. What is the purpose of a type predicate in TypeScript?

- A type predicate narrows a type in conditional checks.
- Declared in the return type as paramName is Type.
- Enables type-safe code paths after checks.
- Example: I used a predicate for validating user inputs:

```
function isUser(input: any): input is { name: string; age: number } {
  return input && typeof input.name === "string" && typeof input.age ===
"number";
}

const obj = { name: "Alice", age: 30 };
if (isUser(obj)) {
  console.log(obj.name); // Safe access
}
```

96. How do template literal types work in TypeScript?

- Combine string literals and unions to define complex string patterns.
- Useful for creating strict string formats or dynamic keys.
- Extendable with generic and mapped types.
- Example: I created strict route paths using template literals:

```
type Route = `/user/${string}`;

const validRoute: Route = "/user/123";
const invalidRoute: Route = "/product/123"; // Error
```

97. How to combine multiple interfaces in TypeScript?

- Use intersection types or extends for composition.
- Intersection types merge properties into a single type.
- Ensure compatibility for overlapping property types.
- Example: I combined interfaces for modular design:

```
interface Address {
  street: string;
  city: string;
}

interface User {
  name: string;
}

type UserWithAddress = User & Address;

const user: UserWithAddress = {
  name: "Alice",
  street: "Main St",
  city: "Wonderland",
};
```

98. What are type assertions in TypeScript?

- Force a value to a specific type using as or <type>.
- Bypass compiler checks when confident about the type.
- Use cautiously to avoid runtime errors.
- Example: I asserted a value during DOM manipulation:

```
const input = document.getElementById("username") as
HTMLInputElement;
input.value = "Alice";
```

99. How does the infer keyword work in TypeScript?

- Extracts types within conditional types.
- Enables reusability and dynamic inference of complex types.
- Commonly used in utility types like ReturnType or Parameters.
- Example: I inferred the return type of a function:

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

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

type GreetReturnType = GetReturnType<typeof greet>; // string
```

100. What are the benefits of TypeScript's strict mode?

- Enforces better code quality and error prevention.
- Includes features like noImplicitAny, strictNullChecks, and more.
- Reduces runtime issues through strict type-checking.
- Example: I enabled strict mode for a safer codebase:

```
{
  "compilerOptions": {
    "strict": true
  }
}
```

101. How do you create a readonly tuple in TypeScript?

- Use readonly keyword before tuple definition.
- Prevents reassigning elements in the tuple.
- Immutable tuples enhance safety in complex data structures.
- Example: I used readonly tuples for API constants:

```
const settings: readonly [number, string] = [10, "Light"];
settings[0] = 20; // Error
```

102. Explain how utility types like Omit work in TypeScript.

- Omit removes specified keys from a type.
- Simplifies type transformations in reusable code.
- Often combined with other utility types for flexibility.
- Example: I omitted sensitive fields in a response type:

```
type User = { name: string; age: number; password: string };
type PublicUser = Omit<User, "password">;

const user: PublicUser = { name: "Alice", age: 30 };
```

103. How does TypeScript handle function overloading?

- Define multiple signatures for varying parameter combinations.
- Use a single implementation to fulfill all overloads.
- Compiler enforces parameter types based on signature.
- Example: I overloaded a string processing function:

```
function process(input: string): string;
function process(input: string[]): string[];
function process(input: any): any {
  return Array.isArray(input) ? input.map(i => i.toUpperCase()) :
input.toUpperCase();
}

console.log(process("test")); // TEST
console.log(process(["a", "b"])); // ["A", "B"]
```

104. What are conditional types in TypeScript?

- Enable type transformations based on conditions.
- Use syntax T extends U ? X : Y.
- Flexible for creating advanced generic utilities.
- Example: I applied conditional types for array handling:

```
type ElementType<T> = T extends Array<infer U> ? U : T;

type StringArray = ElementType<string[]>; // string
type NumberType = ElementType<number>; // number
```

105. How does TypeScript ensure type safety in promise handling?

- Promises are strongly typed with their resolved value.
- Ensures proper chaining and error handling.
- Use async/await for clean syntax and type safety.
- Example: I enforced type-safe API calls with promises:

```
async function fetchData(): Promise<string> {
  return "Data";
}

fetchData().then(data => console.log(data)); // Typed as string
```

1. What is ASP.NET Web API?

- ASP.NET Web API is a framework for building HTTP services that can be consumed by a broad range of clients like browsers, mobile devices, and other applications.
- It supports RESTful services, which means it can return data based on HTTP verbs (GET, POST, PUT, DELETE).
- It can return data in various formats such as JSON, XML, and plain text, making it highly flexible.
- I used ASP.NET Web API to create RESTful endpoints for a cloud-based storage solution, which allowed multiple client apps to interact with the service.

```
public class ProductsController : ApiController
{
   public IEnumerable<Product> Get()
   {
      return productService.GetAllProducts();
   }
}
```

2. What is the difference between ApiController and Controller?

- ApiController is specifically designed for building RESTful services, whereas Controller is used for handling MVC web applications.
- ApiController automatically serializes the data to JSON or XML format, while Controller returns views.
- The routing mechanism differs; ApiController uses attribute routing, whereas Controller uses conventional routing.
- I used ApiController in a microservices project for building stateless APIs, which were consumed by Angular front-end applications.

```
[Route("api/products")]
public class ProductsController : ApiController
{
   [HttpGet]
   public IEnumerable<Product> GetAll() { /* logic */ }
}
```

3. What are the main return types supported in Web API?

- Web API supports various return types, such as IHttpActionResult, HttpResponseMessage, and strongly typed objects.
- IHttpActionResult simplifies unit testing and provides flexibility in returning HTTP responses.
- HttpResponseMessage allows greater control over the response, including headers and status codes.
- I used IHttpActionResult in an e-commerce application to handle errors gracefully and standardize HTTP responses.

```
public IHttpActionResult GetProduct(int id)
{
   var product = productService.GetProduct(id);
   if (product == null)
      return NotFound();
   return Ok(product);
}
```

4. What are the advantages of using ASP.NET Web API?

- It is lightweight and ideal for building RESTful services.
- It supports multiple media types like JSON, XML, and plain text.
- It integrates seamlessly with .NET framework and supports Dependency Injection (DI).
- I utilized Web API in a banking application to expose RESTful endpoints for internal systems and third-party integrations.

```
public IHttpActionResult GetAccounts()
{
   var accounts = accountService.GetAllAccounts();
   return Ok(accounts);
}
```

5. Which status code is used for all uncaught exceptions by default?

- By default, Web API returns status code **500 Internal Server Error** for uncaught exceptions.
- This can be customized using exception filters.
- Exception handling can be improved using middleware or global exception filters.
- I implemented a custom exception filter in a financial application to log errors and provide user-friendly messages.

```
public class CustomExceptionFilter : ExceptionFilterAttribute
{
   public override void OnException(HttpActionExecutedContext context)
   {
      // Log exception and return custom response
      context.Response = new
HttpResponseMessage(HttpStatusCode.InternalServerError);
   }
}
```

6. Explain the usage of HttpResponseMessage.

- HttpResponseMessage is used to create a detailed HTTP response with status codes, headers, and content.
- It provides flexibility when returning custom messages or setting headers.
- It is useful when you need to manipulate the response before sending it to the client.
- I used HttpResponseMessage to set custom headers in a reporting module, which enhanced client-side caching.

```
public HttpResponseMessage GetReport(int id)
{
   var response = Request.CreateResponse(HttpStatusCode.OK,
reportService.GetReport(id));
   response.Headers.Add("Custom-Header", "value");
   return response;
}
```

7. What new features are introduced in ASP.NET Web API 2.0?

- Attribute Routing for better control over URL patterns.
- IHttpActionResult for simplified responses and improved testability.
- CORS support for handling cross-origin requests.
- I used attribute routing in an enterprise project to create clean, readable URLs for RESTful endpoints.

```
[Route("api/orders/{id}")]
public IHttpActionResult GetOrder(int id)
{
   var order = orderService.GetOrderById(id);
   return Ok(order);
}
```

8. What exactly is OAuth (Open Authorization)?

- OAuth is a protocol for authorization, allowing users to grant third-party access to their resources without sharing credentials.
- It uses access tokens instead of credentials to access APIs.
- OAuth supports different flows like authorization code, implicit, and client credentials.
- I implemented OAuth in a SaaS application to allow external partners to securely access APIs.

```
services.AddAuthentication("OAuth").AddJwtBearer("OAuth", options =>
{
   options.TokenValidationParameters = new TokenValidationParameters
   {
      ValidateIssuer = true,
      ValidateAudience = true
   };
});
```

9. Explain the difference between WCF RESTful Service and ASP.NET Web API.

- WCF supports both SOAP and RESTful services, while Web API is built specifically for RESTful services.

- WCF requires more configuration and is heavyweight compared to Web API.
- Web API is more flexible, supporting multiple media types like JSON and XML out of the box.
- I migrated a legacy WCF REST service to Web API to simplify maintenance and improve performance in a healthcare application.

```
public class ProductService : IProductService
{
   public IEnumerable<Product> GetAllProducts() { /* logic */ }
}
```

10. What is Attribute Routing in ASP.NET Web API 2.0?

- Attribute Routing allows you to define routes directly on controller actions using attributes.
- It provides more control and flexibility over URL patterns compared to conventional routing.
- Attribute Routing supports parameters and constraints for cleaner URL management.
- I used attribute routing in a microservices project to create RESTful APIs with descriptive URLs.

```
[Route("api/customers/{id:int}")]
public IHttpActionResult GetCustomer(int id)
{
   var customer = customerService.GetCustomerById(id);
   return Ok(customer);
}
```

11. Is it true that ASP.NET Web API has replaced WCF?

- Web API has not entirely replaced WCF; it depends on the use case.
- WCF is still suitable for scenarios requiring SOAP, security, and reliable messaging.
- Web API is preferred for building RESTful services with lightweight communication.
- I used WCF for internal SOAP services and Web API for external-facing REST APIs in a finance system.

```
public class MyWcfService : IService
{
   public string GetData(int value) { return $"You entered: {value}"; }
}
```

12. What are the differences between Web API and Web API 2?

- Web API 2 introduced Attribute Routing, making it easier to define custom routes.
- It added CORS support, allowing cross-origin requests.
- The IHttpActionResult interface was introduced for cleaner and more testable code.
- I used Web API 2 to manage CORS in a multi-client project that required API access from different domains.

```
[EnableCors(origins: "*", headers: "*", methods: "*")]
public class ProductsController : ApiController
{
   public IEnumerable<Product> GetProducts() { return 
productService.GetAll(); }
}
```

13. Explain the difference between MVC and ASP.NET Web API.

- MVC is designed for web applications that return HTML views, while Web API is for creating RESTful services that return data.
- MVC controllers inherit from the Controller class, while Web API controllers inherit from ApiController.
- Web API supports content negotiation, returning data in JSON, XML, etc., based on client needs.
- I used MVC for a user-facing web portal and Web API for exposing data to mobile apps in a logistics project.

```
public class HomeController : Controller
{
   public ActionResult Index() { return View(); }
}
```

14. Compare WCF and ASP.NET Web API.

- WCF is versatile, supporting SOAP, TCP, and HTTP protocols, while Web API only supports HTTP.
- WCF requires configuration for REST services, whereas Web API is RESTful by default.
- Web API is lightweight and easier to use for simple RESTful services.
- I used Web API in a project requiring mobile-friendly JSON responses and WCF for internal secure messaging.

```
public class MyService : IService
{
   public string GetMessage() { return "Hello WCF"; }
}
```

15. Name types of Action Results in Web API 2.

- OkResult (200 OK)
- NotFoundResult (404 Not Found)
- BadRequestResult (400 Bad Request)
- I used NotFoundResult in an e-commerce project to handle missing product IDs gracefully.

```
public IHttpActionResult GetProduct(int id)
{
   var product = productService.GetProduct(id);
   if (product == null) return NotFound();
   return Ok(product);
}
```

16. In OOP, what is the difference between the Repository Pattern and a Service Layer?

- The Repository Pattern abstracts database operations, handling CRUD logic.
- The Service Layer handles business logic and acts as a bridge between controllers and repositories.
- Using both patterns promotes separation of concerns and clean code architecture.
- I implemented these patterns in a microservice project to separate data access from business logic.

```
public class ProductService
{
   private readonly IProductRepository _repository;
   public ProductService(IProductRepository repository) { _repository = 
repository; }
   public Product GetProduct(int id) { return _repository.Get(id); }
}
```

17. Why are the FromBody and FromUri attributes needed in ASP.NET Web API?

- FromBody is used to bind complex types from the request body.
- FromUri is used to bind simple types from the query string or URL.
- Using these attributes provides clarity and flexibility in data binding.
- I used FromBody to accept JSON payloads in a project where clients submitted data through REST APIs.

```
public IHttpActionResult Post([FromBody] Product product)
{
   productService.AddProduct(product);
   return Ok();
}
```

18. What is ASP.NET Web API Data?

- ASP.NET Web API Data refers to handling data in RESTful APIs, including serialization, model binding, and validation.
- It uses JSON or XML serializers to convert objects to data formats.
- Data handling can be customized using formatters and filters.

- I customized data serialization in a project to handle special date formats for client applications.

```
config.Formatters.JsonFormatter.SerializerSettings.DateFormatString =
"yyyy-MM-dd";
```

19. What's the difference between OpenID and OAuth?

- OpenID is used for authentication, verifying user identity.
- OAuth is for authorization, granting access to resources.
- They can be used together for secure login and resource access.
- I implemented both protocols in a project to allow secure login and third-party API access.

```
services.AddAuthentication("OAuth").AddJwtBearer("OAuth", options => {
/* config */ });
```

20. What is a Delegating Handler in ASP.NET Web API?

- A Delegating Handler is a custom message handler that processes HTTP requests before they reach the controller.
- It can be used for tasks like logging, authentication, and modifying requests/responses.
- Delegating Handlers can be chained together to form a pipeline.
- I used a Delegating Handler in a project to log API requests for auditing purposes.

```
public class LoggingHandler : DelegatingHandler
{
   protected override async Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken
cancellationToken)
   {
      // Log request
      var response = await base.SendAsync(request, cancellationToken);
      return response;
   }
}
```

21. How to register an exception filter globally?

- Exception filters handle exceptions thrown during the execution of an action.
- To register globally, add the filter to the GlobalConfiguration.Configuration.Filters collection.
- This ensures all controllers and actions use the filter.
- I registered a global exception filter in a Web API project to log exceptions and return custom error messages.

```
GlobalConfiguration.Configuration.Filters.Add(new
CustomExceptionFilter());
```

22. How to return a view from an ASP.NET Web API method?

- Web API is designed to return data, not views.
- However, it is possible by combining Web API with MVC, returning a ViewResult.
- Use a regular MVC controller for actions requiring views.
- I used this approach in a hybrid application where some endpoints returned JSON, and others returned HTML views.

```
public ActionResult Index()
{
   return View("MyView");
}
```

23. Can we use Web API with ASP.NET Web Forms?

- Yes, Web API can be integrated with Web Forms by adding API routes in Global.asax.
- It allows you to create RESTful endpoints in a Web Forms project.
- Web Forms pages and Web API controllers can coexist.
- I added Web API to a legacy Web Forms project to expose data to modern front-end clients.

```
GlobalConfiguration.Configure(WebApiConfig.Register);
```

24. Explain briefly Cross-Origin Resource Sharing (CORS).

- CORS is a security feature that allows or restricts resources from being requested from a different domain.
- It requires server-side configuration to enable cross-origin requests.
- Web API 2 has built-in support for CORS using the EnableCors attribute.
- I enabled CORS in a multi-client project to allow requests from multiple domains.

```
[EnableCors(origins: "http://example.com", headers: "*", methods: "*")]
public class ProductsController : ApiController
{
   public IEnumerable<Product> GetProducts() { return
productService.GetAll(); }
}
```

25. Explain advantages/disadvantages of using HttpModule vs Delegating Handler.

- HttpModule processes requests at the IIS pipeline level, affecting all incoming requests.
- Delegating Handler operates within the Web API pipeline and is specific to Web API requests.
- Delegating Handlers are more flexible and easier to configure for API-specific logic.
- I used Delegating Handlers in a project to implement custom authentication for APIs without affecting the entire application.

```
public class CustomHandler : DelegatingHandler
{
   protected override async Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken
cancellationToken)
   {
      // Custom logic
      return await base.SendAsync(request, cancellationToken);
   }
}
```

26. Explain briefly OWIN (Open Web Interface for .NET) self-hosting.

- OWIN decouples web applications from the server, allowing self-hosting without IIS.
- It provides a lightweight, flexible way to host Web APIs.
- Self-hosting is useful for console applications and Windows services.
- I used OWIN self-hosting to run a background task API in a Windows service.

```
WebApp.Start<Startup>("http://localhost:9000");
```

27. Could you clarify what is the best practice with Web API error management?

- Use exception filters to handle exceptions centrally.
- Return meaningful HTTP status codes for different error types.
- Log errors for diagnostics and monitoring.
- I implemented global error handling in a financial application to ensure consistent error responses across APIs.

```
public class GlobalExceptionFilter : ExceptionFilterAttribute
{
   public override void OnException(HttpActionExecutedContext context)
   {
      context.Response = new
HttpResponseMessage(HttpStatusCode.InternalServerError);
   }
}
```

28. What is the difference between WCF, Web API, WCF REST, and Web Service?

- WCF supports SOAP and multiple protocols, Web API is REST-focused and HTTP-only.
- WCF REST requires configuration, whereas Web API is RESTful by default.
- Traditional Web Services (ASMX) are legacy SOAP-based services.

- I chose Web API for a project requiring lightweight, JSON-based communication and WCF for SOAP interoperability.

```
[ServiceContract]
public interface IService
{
   [OperationContract]
   string GetData(int value);
}
```

29. Why should I use IHttpActionResult instead of HttpResponseMessage?

- IHttpActionResult simplifies unit testing by abstracting the HTTP response creation.
- It provides a cleaner, more readable API by encapsulating response details.
- It allows built-in helper methods like Ok(), NotFound(), etc.
- I used IHttpActionResult in a project to streamline controller code and improve readability.

```
public IHttpActionResult GetProduct(int id)
{
   var product = productService.GetProduct(id);
   if (product == null) return NotFound();
   return Ok(product);
}
```

30. How to restrict access to a Web API method to specific HTTP verbs?

- Use HTTP method attributes like [HttpGet], [HttpPost], etc.
- This enforces the correct HTTP verb for each action.
- Combining with routing provides fine-grained control over API behavior.
- I used verb restrictions in a project to separate read and write operations in a RESTful service.

```
[HttpGet]
public IEnumerable<Product> GetProducts() { return
productService.GetAll(); }
```

31. How can we provide an alias name for an ASP.NET Web API action?

- Use the [ActionName] attribute to give an action a different name in the route.
- This helps avoid name conflicts and improves API readability.
- Aliases are useful for creating user-friendly endpoints.
- I used action aliases in a project to create intuitive, readable API routes.

```
[ActionName("List")]
public IEnumerable<Product> GetAllProducts() { return
productService.GetAll(); }
```