

3

4 1. You are designing a billing system. How would you enforce that every bill must
implement CalculateTotal() method without providing any default behavior?

5

6 Answer:

7 I would create an interface like this:

8

9 interface IBill

10 {

11 double CalculateTotal();

12 }

13

14 Now, every billing class like Invoice or PurchaseOrder must implement the
CalculateTotal() method explicitly.

15

16 Tip : This ensures flexibility and forces a contract.

17

18

19

20

21

22 2. In a real-world app, you want to initialize database connections only once for all
users. How will you implement it?

23

24 Answer:

25 I would use the Singleton pattern like this:

26

27 class DBConnection

28 {

29 private static DBConnection instance;

30 private DBConnection() {}

31

32 public static DBConnection Instance

33 {

34 get

35 {

36 if (instance == null)

37 instance = new DBConnection();

38 return instance;

39 }

40 }

41 }

42

43 Note: This ensures a single connection object reused across the application.

44

45

46

47

48 3. How will you make sure that a Customer object is immutable once created?

49

50 Answer:

51 I would make all properties readonly and initialize them inside the constructor:

52

53 class Customer

54 {

55 public string Name { get; }

56 public int Age { get; }

57

58 public Customer(string name, int age)

59 {

60 Name = name;

61 Age = age;

62 }

63 }

64

65 Note: Once constructed, object values can't be changed.

66

4. You have a Shape base class and Circle, Rectangle as derived classes. How would you design a method Draw() differently for each?

Answer:

Use virtual and override keywords properly like below:

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing Shape");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing Circle");
    }
}

class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing Rectangle");
    }
}
```

Note: Different derived classes give different implementations dynamically (Polymorphism).

5. In an E-commerce platform, you want different products to have different price calculation logic. How will you design it?

Answer:

Use the Strategy Pattern by creating different price calculators:

```
interface IPriceStrategy
{
    decimal CalculatePrice(Product p);
}

class DiscountPrice : IPriceStrategy
{
    public decimal CalculatePrice(Product p)
    {
        return p.BasePrice * 0.9m;
    }
}

class PremiumPrice : IPriceStrategy
{
    public decimal CalculatePrice(Product p)
    {
        return p.BasePrice * 1.2m;
    }
}
```

Note: Different pricing logic can be injected at runtime without changing product code.

133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201

6. How can you restrict a class from being inherited further?

Answer:

Use the sealed keyword in C#.

Example:

```
sealed class BankAccount
{
    public void DisplayBalance()
    {
        Console.WriteLine("Balance shown.");
    }
}
```

Note: sealed prevents any other class from extending BankAccount.

7. How do you enforce multiple inheritance in C# without using classes?

Answer:

C# doesn't support multiple inheritance with classes but it supports it via interfaces.

Example:

```
interface IPrintable
{
    void Print();
}

interface IScannable
{
    void Scan();
}

class MultiFunctionDevice : IPrintable, IScannable
{
    public void Print()
    {
        Console.WriteLine("Printing...");
    }

    public void Scan()
    {
        Console.WriteLine("Scanning...");
    }
}
```

Note: Multiple interfaces allow a class to inherit multiple behaviors.

8. What would you use to define fixed data like Status {Pending, Approved, Rejected}?

Answer:

I would use an enum for this type of scenario.

Example:

```
enum Status
```

```
202 {
203     Pending,
204     Approved,
205     Rejected
206 }
207
208 Note: Enums provide meaningful names to integral values, improving code readability.
209
210 9. You need to provide method overloading to handle both int and float types for the
    same function AddNumbers(). How would you design it?
211
212 Answer:
213 I would implement two overloaded versions of the method.
214
215 Example:
216
217 class Calculator
218 {
219     public int AddNumbers(int a, int b)
220     {
221         return a + b;
222     }
223
224     public float AddNumbers(float a, float b)
225     {
226         return a + b;
227     }
228 }
229
230 Note: Method Overloading allows multiple methods with the same name but different
    parameter types.
231
232
233
234
235
236 10. Can we overload constructors? Why is it useful in real-time applications?
237
238 Answer:
239 Yes, we can overload constructors to allow different ways of object initialization.
240
241 Example:
242
243 class Student
244 {
245     public string Name;
246     public int Age;
247
248     public Student()
249     {
250         Name = "Unknown";
251         Age = 0;
252     }
253
254     public Student(string name)
255     {
256         Name = name;
257         Age = 0;
258     }
259
260     public Student(string name, int age)
261     {
262         Name = name;
263         Age = age;
264     }
265 }
266
267 Note: Real-world use: Providing flexibility to create objects with full or partial
    information.
```

268
269
270 11. In a library management system, you need to ensure that once a book is issued, it cannot be modified. How would you design it?

271
272 Answer:
273 I would create an immutable class for the book issue record.
274
275 Example:
276
277 class IssuedBook
278 {
279 public string BookName { get; }
280 public string IssuedTo { get; }
281 public DateTime IssuedDate { get; }
282
283 public IssuedBook(string bookName, string issuedTo, DateTime issuedDate)
284 {
285 BookName = bookName;
286 IssuedTo = issuedTo;
287 IssuedDate = issuedDate;
288 }
289 }
290
291 Note: This way, once created, IssuedBook details cannot be changed.
292
293
294
295
296
297 12. How would you implement a base class that forces child classes to override a method but still allows base functionality?

298
299 Answer:
300 Use an abstract class with abstract methods.
301
302 Example:
303
304 abstract class Report
305 {
306 public void PrintHeader()
307 {
308 Console.WriteLine("Company Confidential Report");
309 }
310
311 public abstract void PrintDetails();
312 }
313
314 class SalesReport : Report
315 {
316 public override void PrintDetails()
317 {
318 Console.WriteLine("Sales Details Printed");
319 }
320 }
321
322 Note: This enforces PrintDetails() to be overridden, but allows reuse of PrintHeader().
323
324
325
326
327
328 13. In a payment gateway integration, how would you implement dynamic switching between CreditCard and PayPal payments?

329
330 Answer:
331 I would use interface-based strategy switching.
332
333 Example:

```

334
335 interface IPaymentGateway
336 {
337     void ProcessPayment(double amount);
338 }
339
340 class CreditCardPayment : IPaymentGateway
341 {
342     public void ProcessPayment(double amount)
343     {
344         Console.WriteLine($"CreditCard Payment done for {amount}");
345     }
346 }
347
348 class PayPalPayment : IPaymentGateway
349 {
350     public void ProcessPayment(double amount)
351     {
352         Console.WriteLine($"PayPal Payment done for {amount}");
353     }
354 }
355
356 Note: Depending on user choice, dynamically assign the appropriate payment method
      object.
357
358
359
360
361
362 14. Explain what happens when you mark a method as virtual but don't override it in
      child class?
363
364 Answer:
365 If a method is marked virtual in the base class but not overridden in the derived class,
      the base class version of the method is called.
366
367 Example:
368
369 class Employee
370 {
371     public virtual void DisplayRole()
372     {
373         Console.WriteLine("Employee Role");
374     }
375 }
376
377 class Manager : Employee
378 {
379     // No override here
380 }
381
382 When you call DisplayRole() on Manager, it prints:
383 Employee Role
384
385 Note: Only if you override, the derived class version is called.
386
387
388
389
390 15. In a chat application, you want users to be able to "SendMessage" differently based
      on platform (Mobile/Desktop). How would you design it?
391
392 Answer:
393 I would define an interface and provide different implementations.
394
395 Example:
396
397 interface IChatPlatform
398 {

```

```

399     void SendMessage(string message);
400 }
401
402 class MobileChat : IChatPlatform
403 {
404     public void SendMessage(string message)
405     {
406         Console.WriteLine($"Sending via Mobile: {message}");
407     }
408 }
409
410 class DesktopChat : IChatPlatform
411 {
412     public void SendMessage(string message)
413     {
414         Console.WriteLine($"Sending via Desktop: {message}");
415     }
416 }

```

Note: At runtime, based on device type, create appropriate object and send the message.

16. How would you design a system where certain functionality should only be accessible to Admin users at runtime?

Answer:

I would use interface segregation and role-based checks.

Example:

```

431 interface IUserFunctions
432 {
433     void ViewProfile();
434 }
435
436 interface IAdminFunctions : IUserFunctions
437 {
438     void DeleteUser();
439     void AddUser();
440 }
441
442 class AdminUser : IAdminFunctions
443 {
444     public void ViewProfile()
445     {
446         Console.WriteLine("Viewing Admin Profile");
447     }
448
449     public void DeleteUser()
450     {
451         Console.WriteLine("Deleting User");
452     }
453
454     public void AddUser()
455     {
456         Console.WriteLine("Adding User");
457     }
458 }

```

Note: Based on role, we decide which interface methods are exposed to the user.

17. How would you reuse logic across multiple unrelated classes without inheritance?

467 Answer:
468 Use Composition rather than inheritance.

469
470 Example:

```
471  
472 class Logger  
473 {  
474     public void Log(string message)  
475     {  
476         Console.WriteLine($"Log: {message}");  
477     }  
478 }  
479  
480 class OrderService  
481 {  
482     private Logger _logger = new Logger();  
483  
484     public void PlaceOrder()  
485     {  
486         _logger.Log("Order Placed");  
487     }  
488 }  
489  
490 class PaymentService  
491 {  
492     private Logger _logger = new Logger();  
493  
494     public void ProcessPayment()  
495     {  
496         _logger.Log("Payment Processed");  
497     }  
498 }  
499
```

500 Note: Composition promotes code reuse without tying classes together rigidly.

501
502
503
504
505 18. What is the use of base keyword? Give a real example.

506
507 Answer:
508 base is used to access members of the base class from within a derived class.

509
510 Example:

```
511  
512 class Vehicle  
513 {  
514     public void Start()  
515     {  
516         Console.WriteLine("Vehicle started");  
517     }  
518 }  
519  
520 class Car : Vehicle  
521 {  
522     public void StartCar()  
523     {  
524         base.Start();  
525         Console.WriteLine("Car is ready to drive");  
526     }  
527 }  
528
```

529 Note: base.Start() allows reuse of the Vehicle's Start method inside Car without redefining.

530
531
532
533
534

535 19. What is the benefit of using abstract classes compared to interfaces?

536

537 Answer:

538

539 Abstract classes allow you to provide partial implementation (methods with body +
methods without body).

540

541 Interfaces only allow method declarations (until C# 8.0).

542

543 Example:

544

545 abstract class Animal

546 {

547 public void Breathe()

548 {

549 Console.WriteLine("Breathing...");

550 }

551

552 public abstract void MakeSound();

553 }

554

555 class Dog : Animal

556 {

557 public override void MakeSound()

558 {

559 Console.WriteLine("Barks");

560 }

561 }

562

563 Note: Abstract classes allow code reuse plus enforce important abstract methods.

564

565

566

567

568

569 20. In a microservices project, services should be loosely coupled but must agree on
contracts. How would you enforce this?

570

571 Answer:

572 I would define shared interfaces across services.

573

574 Example:

575

576 interface IUserService

577 {

578 string GetUserName(int userId);

579 }

580

581 Each microservice can have different implementations but must respect the interface
contract.

582

583 Note: Helps in loosely coupled but well-defined architecture.

584

585

586

587

588

589 21. In a banking system, you want to hide sensitive fields like AccountBalance but still
allow displaying AccountHolderName. How would you design it?

590

591 Answer:

592 I would use access modifiers and properties.

593

594 Example:

595

596 class BankAccount

597 {

598 public string AccountHolderName { get; set; }

599 private double AccountBalance { get; set; }

```
600
601     public BankAccount(string name, double balance)
602     {
603         AccountHolderName = name;
604         AccountBalance = balance;
605     }
606 }
607
608 Note: AccountBalance cannot be accessed directly outside the class.
```

```
609
610
611
612
613
614 22. Explain the difference between override and new keywords in method overriding.
```

```
615
616 Answer:
617
618 override extends the base class method and provides new behavior.
619
620 new hides the base class method intentionally (not polymorphic).
```

```
621
622 Example:
623
624 class Base
625 {
626     public virtual void Show()
627     {
628         Console.WriteLine("Base Show");
629     }
630 }
631
632 class Derived : Base
633 {
634     public new void Show()
635     {
636         Console.WriteLine("Derived Show");
637     }
638 }
639
```

```
640 Note: If you use a base class reference, it calls Base's Show().
641 Note: For polymorphism, override should be used, not new.
```

```
642
643 23. You need to implement loose coupling between UI layer and Business Logic layer. How
will you achieve it?
```

```
644
645 Answer:
646 I would use interfaces or dependency injection (DI).
```

```
647
648 Example:
649
650 interface IEmployeeService
651 {
652     string GetEmployeeName(int id);
653 }
654
655 class EmployeeService : IEmployeeService
656 {
657     public string GetEmployeeName(int id)
658     {
659         return $"Employee_{id}";
660     }
661 }
662
```

```
663 In UI Layer:
```

```
664
665 class EmployeeUI
666 {
667     private IEmployeeService _employeeService;
```

```

668
669     public EmployeeUI(IEmployeeService service)
670     {
671         _employeeService = service;
672     }
673
674     public void ShowName(int id)
675     {
676         Console.WriteLine(_employeeService.GetEmployeeName(id));
677     }
678 }
679

```

Note: This way, UI depends only on interface, not on concrete classes.

24. Can you create an object of an abstract class? How is it useful?

Answer:

No, you cannot create an object of an abstract class directly.

However, abstract classes are useful for defining a base template.

Example:

```

693
694 abstract class Document
695 {
696     public abstract void Print();
697 }
698
699 class PdfDocument : Document
700 {
701     public override void Print()
702     {
703         Console.WriteLine("Printing PDF Document");
704     }
705 }
706

```

Note: Forces derived classes to implement important methods like Print().

25. Explain Interface Segregation Principle (ISP) with example.

Answer:

ISP says – don't force a class to implement methods it doesn't use.

Better to split into smaller interfaces.

Example (Bad Design):

```

720
721 interface IMachine
722 {
723     void Print();
724     void Fax();
725     void Scan();
726 }
727
728 class OldPrinter : IMachine
729 {
730     public void Print()
731     {
732         Console.WriteLine("Printed");
733     }
734
735     public void Fax()
736     {

```

```
737         throw new NotImplementedException();
738     }
739
740     public void Scan()
741     {
742         throw new NotImplementedException();
743     }
744 }
```

746 Note: Problem: OldPrinter is forced to implement Fax() and Scan() unnecessarily.

748 Good Design:

```
749
750 interface IPrinter
751 {
752     void Print();
753 }
754
755 interface IScanner
756 {
757     void Scan();
758 }
759
760 class SimplePrinter : IPrinter
761 {
762     public void Print()
763     {
764         Console.WriteLine("Printed");
765     }
766 }
```

768 Note: Classes implement only what they need.

775 26. When would you prefer an abstract class over an interface?

777 Answer:

778 Use an abstract class when:

780 You want to provide default implementation for some methods.

782 You want to define common fields or constructors.

784 Example:

```
785
786 abstract class Animal
787 {
788     public string Name;
789
790     public Animal(string name)
791     {
792         Name = name;
793     }
794
795     public void Breathe()
796     {
797         Console.WriteLine($"{Name} is breathing");
798     }
799
800     public abstract void MakeSound();
801 }
802
803 class Dog : Animal
804 {
805     public Dog(string name) : base(name) {}
806 }
```

```
806
807     public override void MakeSound()
808     {
809         Console.WriteLine($"{Name} barks");
810     }
811 }
```

812
813 Note: Abstract classes are perfect when some shared logic needs to be inherited.

814
815
816
817
818
819
820 27. What happens if you don't implement all methods of an interface in your class?

821
822 Answer:

823
824 The compiler will throw an error.

825
826 To fix it, the class must either be declared abstract, or all methods must be implemented.

827
828 Example:

```
829
830 interface IShape
831 {
832     void Draw();
833     void Resize();
834 }
835
836 class Circle : IShape
837 {
838     public void Draw()
839     {
840         Console.WriteLine("Drawing Circle");
841     }
842
843     // Forgot to implement Resize() - ✗ Compiler Error
844 }
```

845
846 Note: Must implement both Draw() and Resize() or declare Circle as abstract.

847
848
849
850
851
852 28. Why can't a constructor be virtual in C#?

853
854 Answer:

855
856 Constructors are not inherited.

857
858 Since virtual relates to method overriding in derived classes, constructors cannot be virtual.

859
860 Example:

```
861
862 // This is invalid:
863 // public virtual MyClass() { } // ✗ Not allowed
864
```

865 Note: Instead, constructor chaining using base() can be used to control construction flow across base and derived classes.

866
867 Example:

```
868
869 class Base
870 {
871     public Base()
```

```

872     {
873         Console.WriteLine("Base constructor");
874     }
875 }
876
877 class Derived : Base
878 {
879     public Derived() : base()
880     {
881         Console.WriteLine("Derived constructor");
882     }
883 }

```

884
885
886
887

888 29. How would you handle a situation where a derived class needs to extend behavior but also call base behavior?

889

890 Answer:

891 Use base.MethodName() inside overridden methods.

892

893 Example:

894

```

895 class Notification
896 {
897     public virtual void Send()
898     {
899         Console.WriteLine("Sending basic notification");
900     }
901 }
902
903 class EmailNotification : Notification
904 {
905     public override void Send()
906     {
907         base.Send();
908         Console.WriteLine("Sending Email notification additionally");
909     }
910 }

```

911

912 Note: First calls base logic, then extends behavior.

913

914

915

916

917

918 30. What is a real-world example of using sealed keyword practically?

919

920 Answer:

921 If you create a utility/helper class like MathHelper, and you don't want others to override or extend it accidentally, mark it as sealed.

922

923 Example:

924

```

925 sealed class MathHelper
926 {
927     public static int Add(int a, int b)
928     {
929         return a + b;
930     }
931 }

```

932

933 // class ExtendedHelper : MathHelper // ✗ Not allowed. Compile-time error.

934

935 Note: sealed ensures no one can inherit and misuse/change base behavior.

936

937

938

31. How do you call a base class constructor explicitly from a derived class?

Answer:

Use : base(parameters) in the derived class constructor.

Example:

```
class Person
{
    public string Name;

    public Person(string name)
    {
        Name = name;
    }
}

class Employee : Person
{
    public Employee(string name) : base(name)
    {
        Console.WriteLine($"Employee Name: {Name}");
    }
}
```

Note: Helps reusing initialization logic defined in base classes.

32. How do you achieve polymorphism using interfaces?

Answer:

By referring to objects using their interface type, not their concrete class.

Example:

```
interface IShape
{
    void Draw();
}

class Circle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing Circle");
    }
}

class Square : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing Square");
    }
}
```

Usage:

```
IShape shape = new Circle();
shape.Draw();

shape = new Square();
shape.Draw();
```

1008
1009 Note: Same method Draw() behaves differently based on object type.
1010
1011
1012
1013

1014
1015
1016 33. Explain the concept of explicit interface implementation.
1017

1018 Answer:

1019 When two interfaces have methods with same signature, you can explicitly implement to
1020 avoid conflicts.

1021 Example:

```
1022 interface IPrinter
1023 {
1024     void Print();
1025 }
1026
1027 interface IScanner
1028 {
1029     void Print();
1030 }
1031
1032
1033 class MultiFunctionMachine : IPrinter, IScanner
1034 {
1035     void IPrinter.Print()
1036     {
1037         Console.WriteLine("Printing Document");
1038     }
1039
1040     void IScanner.Print()
1041     {
1042         Console.WriteLine("Scanning Document");
1043     }
1044 }
```

1045
1046 Usage:

```
1047
1048 IPrinter printer = new MultiFunctionMachine();
1049 printer.Print();
```

```
1050
1051 IScanner scanner = new MultiFunctionMachine();
1052 scanner.Print();
```

1053
1054 Note: Explicit Implementation clearly separates behavior.
1055
1056
1057
1058
1059
1060

1061 34. How would you prevent a method in base class from being overridden?
1062

1063 Answer:

1064 Declare the method as sealed in the derived class while overriding.
1065

1066 Example:

```
1067
1068 class Parent
1069 {
1070     public virtual void Greet()
1071     {
1072         Console.WriteLine("Hello from Parent");
1073     }
1074 }
1075
```



```
1076 class Child : Parent
1077 {
1078     public sealed override void Greet()
1079     {
1080         Console.WriteLine("Hello from Child");
1081     }
1082 }
1083
1084 // class GrandChild : Child { override Greet() ✗ Not Allowed }
1085
1086 Note: sealed override prevents further overriding in the inheritance chain.
```

```
1087
1088
1089
1090
1091
1092 35. How would you implement Dependency Injection manually in C#?
```

```
1093
1094 Answer:
1095 You can inject dependencies via constructor without any framework.
```

```
1096
1097 Example:
1098
1099 interface INotification
1100 {
1101     void Send(string message);
1102 }
1103
1104 class EmailNotification : INotification
1105 {
1106     public void Send(string message)
1107     {
1108         Console.WriteLine($"Sending Email: {message}");
1109     }
1110 }
1111
1112 class UserService
1113 {
1114     private INotification _notification;
1115
1116     public UserService(INotification notification)
1117     {
1118         _notification = notification;
1119     }
1120
1121     public void RegisterUser(string username)
1122     {
1123         Console.WriteLine($"{username} Registered");
1124         _notification.Send("Welcome to our app!");
1125     }
1126 }
1127
```

```
1128 Usage:
1129
1130 INotification notifier = new EmailNotification();
1131 UserService service = new UserService(notifier);
1132 service.RegisterUser("Karthik");
1133
1134 Note: Manual DI improves testability and flexibility.
```

```
1135
1136
1137 36. What is method hiding in C#? How is it different from overriding?
```

```
1138
1139 Answer:
1140
1141 Hiding uses the new keyword.
1142
1143 Overriding uses the override keyword and works via polymorphism.
```

```
1144
```

1145 Example of method hiding:

```
1146
1147 class Base
1148 {
1149     public void Display()
1150     {
1151         Console.WriteLine("Base Display");
1152     }
1153 }
1154
1155 class Derived : Base
1156 {
1157     public new void Display()
1158     {
1159         Console.WriteLine("Derived Display");
1160     }
1161 }
```

1162 Usage:

```
1163 Base obj = new Derived();
1164 obj.Display();
```

1165 Note: Output: Base Display (because hiding does not change base behavior with base reference)

1166

1167

1168

1169 37. Can you override a non-virtual method in C#?

1170

1171 Answer:

1172 No, you cannot override a non-virtual method.

1173 Only methods declared as virtual, abstract, or override can be overridden.

1174

1175 Example:

1176

```
1177 class Parent
```

```
1178 {
1179     public void SayHello()
1180     {
1181         Console.WriteLine("Hello from Parent");
1182     }
1183 }
```

1184

```
1185 class Child : Parent
```

```
1186 {
1187     // public override void SayHello() ✕ Compile-time Error
1188 }
```

1189

1190 Note: Always mark the method virtual first if you want it to be overridden.

1191

1192

1193

1194

1195

1196

1197

1198 38. What are extension methods? Why are they useful?

1199

1200 Answer:

1201

1202 Extension Methods allow you to add methods to existing types without modifying them.

1203

1204 Very useful for working with external or sealed classes.

1205

1206 Syntax:

1207

```
1208 public static class StringExtensions
```

```
1209 {
1210     public static int WordCount(this string str)
```

1211

```

1213     {
1214         return str.Split(' ').Length;
1215     }
1216 }
1217

```

1218 Usage:

```

1219
1220 string message = "Hello Karthik How Are You";
1221 int count = message.WordCount();
1222 Console.WriteLine(count);
1223

```

1224 Note: Adds behavior without changing original class definition!

1225
1226
1227
1228
1229 39. What is the difference between abstract class and interface?

1230	1231	1232	1233	1234	1235	1236	1237
		Feature	Abstract Class	Interface			
		=====	=====	=====			
		Inheritance	Single inheritance	Multiple inheritance			
		Constructors	Allowed	Not allowed			
		Access Modifiers	Can have public/private/protected	Always public methods			
		(before C#8)					
		Default Implementation	Possible (Concrete methods)	Not possible (until C#8)			

1238
1239 Note: Abstract classes are partially implemented templates.

1240
1241 Interfaces are pure contracts.

1242
1243
1244
1245
1246
1247 40. How can you prevent inheritance of a class?

1248
1249 Answer:

1250 Use the sealed keyword.

1251
1252 Example:

```

1253
1254 sealed class BankAccount
1255 {
1256     public void Deposit()
1257     {
1258         Console.WriteLine("Deposit Done");
1259     }
1260 }
1261
1262 // class SavingsAccount : BankAccount ✗ Cannot inherit
1263

```

1264 Note: sealed classes are used when you want final implementations without any child class overriding or modifying the behavior.

1265
1266
1267
1268
1269
1270 41. What is multiple inheritance? Does C# support it?

1271
1272 Answer:

1273
1274 Multiple inheritance means a class inherits from more than one class.

1275
1276 C# does NOT support multiple class inheritance to avoid ambiguity (like Diamond Problem).

1277
1278 However, C# supports multiple interface inheritance.

1279

```
1280 Example:
1281
1282 interface IPrinter
1283 {
1284     void Print();
1285 }
1286
1287 interface IScanner
1288 {
1289     void Scan();
1290 }
1291
1292 class MultiFunctionMachine : IPrinter, IScanner
1293 {
1294     public void Print()
1295     {
1296         Console.WriteLine("Printing...");
1297     }
1298
1299     public void Scan()
1300     {
1301         Console.WriteLine("Scanning...");
1302     }
1303 }
```

1305 Note: So C# achieves multiple behavior inheritance via interfaces, not classes.

1311 42. Why would you make a class abstract if it has no abstract methods?

1313 Answer:

1314 Even without abstract methods, marking a class as abstract is useful when:

1316 You don't want the class to be instantiated directly.

1318 You want it to act as a base only for derived classes.

1320 Example:

```
1322 abstract class Shape
1323 {
1324     public void Move()
1325     {
1326         Console.WriteLine("Shape moved");
1327     }
1328 }
```

1330 Note: Shape is not supposed to be instantiated directly.

1336 43. What is the use of readonly fields in a class?

1338 Answer:

1340 readonly fields can only be assigned during declaration or inside constructor.

1342 Once assigned, they cannot be changed outside.

1344 Example:

```
1346 class Vehicle
1347 {
1348     public readonly string EngineType;
```

```
1349
1350     public Vehicle(string engineType)
1351     {
1352         EngineType = engineType;
1353     }
1354 }
1355
1356 Usage:
1357
1358 Vehicle v = new Vehicle("Petrol");
1359 // v.EngineType = "Diesel"; ✗ Not allowed
1360
1361 Note: Ensures immutability after object creation.
```

1367 44. Can abstract classes have constructors? If yes, why?

1369 Answer:

1370 Note: Yes, abstract classes can have constructors!

1372 Constructors are used to initialize fields common to derived classes.

1374 Example:

```
1376 abstract class Employee
1377 {
1378     protected string Name;
1379
1380     public Employee(string name)
1381     {
1382         Name = name;
1383     }
1384 }
1385
1386 class Manager : Employee
1387 {
1388     public Manager(string name) : base(name) {}
1389 }
```

1391 Note: Derived classes can call base constructor to reuse initialization logic.

1398 45. How to force derived classes to implement certain methods?

1400 Answer:

1401 Declare those methods as abstract in the base abstract class.

1403 Example:

```
1405 abstract class Animal
1406 {
1407     public abstract void MakeSound();
1408 }
1409
1410 class Dog : Animal
1411 {
1412     public override void MakeSound()
1413     {
1414         Console.WriteLine("Dog Barks");
1415     }
1416 }
```

1418 Note: Now Dog must implement MakeSound().
1419 Otherwise, the compiler will throw an error.

1420
1421
1422
1423

1424 46. What is the difference between const and readonly in C#?

1425

1426 Answer:

1427

1428

Feature	const	readonly
=====	=====	=====
Assignment constructor) allowed	At compile time only	At runtime (in
Type default)	Static by default	Instance level (by
Modifiable inside constructor	Cannot modify even inside constructor	Can modify once

1434

1435 Example:

1436

1437 class Example

1438 {

1439 public const double Pi = 3.14;

1440 public readonly int RollNumber;

1441

1442 public Example(int rollNumber)

1443 {

1444 RollNumber = rollNumber;

1445 }

1446 }

1447

1448 Note: const is for fixed values.

1449 Note: readonly is for values decided during object creation.

1450

1451

1452

1453

1454 47. What is shadowing or method hiding in C#?

1455

1456 Answer:

1457

1458 Shadowing means redefining a method in a derived class without overriding.

1459

1460 Use new keyword.

1461

1462 Example:

1463

1464 class Base

1465 {

1466 public void Display()

1467 {

1468 Console.WriteLine("Base Display");

1469 }

1470 }

1471

1472 class Derived : Base

1473 {

1474 public new void Display()

1475 {

1476 Console.WriteLine("Derived Display");

1477 }

1478 }

1479

1480 Usage:

1481

1482 Base obj = new Derived();

1483 obj.Display(); // Output: Base Display

1484
1485 Note: In shadowing, polymorphism does NOT happen.
1486

1487
1488
1489
1490 48. What is constructor chaining in C#?
1491

1492 Answer:
1493 Constructor chaining means calling one constructor from another constructor of the same
class or base class.

1494
1495 Example:

```
1496 class Student
1497 {
1498     public string Name;
1500     public int Age;
1501
1502     public Student() : this("Unknown", 0)
1503     {
1504     }
1505
1506     public Student(string name, int age)
1507     {
1508         Name = name;
1509         Age = age;
1510     }
1511 }
```

1512
1513 Note: Improves code reuse and reduces duplication.
1514

1515
1516
1517
1518
1519 49. What is an example of real-world polymorphism in C#?
1520

1521 Answer:
1522 When you have common behavior with different implementations.
1523

1524 Example:

```
1525
1526 abstract class Payment
1527 {
1528     public abstract void MakePayment();
1529 }
1530
1531 class CreditCardPayment : Payment
1532 {
1533     public override void MakePayment()
1534     {
1535         Console.WriteLine("Payment done through Credit Card");
1536     }
1537 }
1538
1539 class PayPalPayment : Payment
1540 {
1541     public override void MakePayment()
1542     {
1543         Console.WriteLine("Payment done through PayPal");
1544     }
1545 }
```

1546
1547 Usage:

```
1548
1549 Payment payment = new CreditCardPayment();
1550 payment.MakePayment();
1551
```

```
1552 payment = new PayPalPayment();
1553 payment.MakePayment();
1554
```

1555 Note: Different classes respond differently to the same method call!

```
1556
1557
1558
1559
1560
1561
```

1562 50. What is encapsulation? How is it implemented in C#?

```
1563
1564
```

Answer:

```
1565
1566
```

Encapsulation is hiding internal data and exposing only necessary parts via public properties/methods.

```
1567
1568
```

Protects data integrity.

```
1569
1570
```

Example:

```
1571
1572
```

```
class Employee
```

```
1573 {
```

```
    private int salary;
```

```
1575
```

```
    public int Salary
```

```
1577 {
```

```
        get { return salary; }
```

```
1579        set
```

```
        {
```

```
1581            if (value > 0)
```

```
                salary = value;
```

```
1583        }
```

```
    }
```

```
1585 }
```

```
1586
```

Note: No direct access to salary.

```
1588
```

Note: Data is controlled via property logic.

```
1589
1590
```

```
1591
1592
```

```
1593
1594
```

1595 51. What is the difference between early binding and late binding in C#?

```
1596
1597
```

Answer:

```
1598
```

Early Binding: Method calls are resolved at compile time. (static typing)

```
1600
```

Late Binding: Method calls are resolved at runtime. (dynamic typing or reflection)

```
1602
```

Example of early binding:

```
1604
1605
```

```
class Calculator
```

```
1606 {
```

```
    public int Add(int a, int b)
```

```
1608    {
```

```
        return a + b;
```

```
1610    }
```

```
1611 }
```

```
1612
```

Usage:

```
1614
```

```
Calculator calc = new Calculator();
```

```
1616 int result = calc.Add(3, 5);
```

```
1617
```

Note: Compiler knows the method Add() at compile time.

```
1619
```



```

1620 Example of late binding using dynamic:
1621
1622 dynamic obj = new Calculator();
1623 int result = obj.Add(4, 6);
1624
1625 Note: Method resolution happens at runtime.
1626 Note: Useful when method availability is dynamic (ex: COM, plugins, reflection).
1627
1628
1629
1630

```

```

1631 52. What is the difference between dynamic and var in C#?
1632

```

```

1633 Answer:
1634

```

```

1635
1636 Feature                var                dynamic
1637 =====
1638 When resolved          Compile-time          Runtime
1639 Flexibility            Type cannot change after assigned    Type can change during execution
1640 Errors caught          At compile-time          At runtime
1641 Example:

```

```

1642
1643 var name = "Karthik";
1644 // name = 100; // ✗ Compile-time Error
1645
1646 dynamic obj = "Karthik";
1647 obj = 100; // Note: No error until runtime
1648

```

```

1649 Note: Use dynamic when you want maximum flexibility, but lose type safety.
1650

```

```

1651
1652
1653
1654 53. What is a sealed class in C#? When should you use it?
1655

```

```

1656 Answer:
1657

```

```

1658 A sealed class cannot be inherited.
1659

```

```

1660 Use sealed classes when you want to restrict modification and ensure final
1661 implementation.

```

```

1662 Example:
1663

```

```

1664 sealed class Logger
1665 {
1666     public void Log(string message)
1667     {
1668         Console.WriteLine(message);
1669     }
1670 }
1671
1672 // class ExtendedLogger : Logger ✗ Error
1673

```

```

1674 Note: Typical usage: Helper classes, utility classes, or security reasons.
1675

```

```

1676
1677
1678
1679
1680 54. Explain shallow copy vs deep copy in C#.
1681

```

```

1682 Answer:
1683

```

```

1684 Shallow Copy: Copies only reference addresses (not new objects).
1685

```

```

1686 Deep Copy: Copies entire new objects recursively.
1687

```

1688 Example of shallow copy using MemberwiseClone():

```
1689
1690 class Person
1691 {
1692     public string Name;
1693     public Person Clone()
1694     {
1695         return (Person)this.MemberwiseClone();
1696     }
1697 }
```

1698
1699 Note: If you modify the reference type field, both original and clone are affected!

1700 Note: Deep copy means you create separate copies of reference objects manually.

1702
1703
1704

1705
1706
1707
1708 55. How do you implement a singleton class in C#?

1709
1710 Answer:
1711 Singleton ensures only one instance of a class is created.

1712
1713 Implementation:

```
1714
1715 class Singleton
1716 {
1717     private static Singleton instance;
1718     private static readonly object locker = new object();
1719
1720     private Singleton() {}
1721
1722     public static Singleton GetInstance()
1723     {
1724         if (instance == null)
1725         {
1726             lock (locker)
1727             {
1728                 if (instance == null)
1729                 {
1730                     instance = new Singleton();
1731                 }
1732             }
1733         }
1734         return instance;
1735     }
1736 }
```

1737
1738 Note: Ensures thread-safe, lazy initialization, and single instance.

1739
1740

1741
1742
1743
1744
1745
1746 56. What is the difference between override and new keyword in C#?

1747
1748 Answer:

Feature	override	new
Purpose	Replace base class method logic	Hide base class method
Runtime behavior	Polymorphism (dynamic dispatch)	No polymorphism
When used	Base method must be virtual	Base method need not be virtual

```
1757 Example of override:
1758
1759 class Base
1760 {
1761     public virtual void Show()
1762     {
1763         Console.WriteLine("Base Show");
1764     }
1765 }
1766
1767 class Derived : Base
1768 {
1769     public override void Show()
1770     {
1771         Console.WriteLine("Derived Show");
1772     }
1773 }
1774
```

1775 Example of new:

```
1776
1777 class DerivedNew : Base
1778 {
1779     public new void Show()
1780     {
1781         Console.WriteLine("Derived New Show");
1782     }
1783 }
1784
```

1785 Note: Use override for polymorphism, new for hiding methods.

1786
1787
1788
1789
1790
1791 57. How do you create a private constructor? When do you use it?

1792
1793 Answer:

1794 Private constructors are used in Singleton Pattern or Utility classes.

1795 Prevents object creation from outside.

1796
1797 Example:

```
1800
1801 class DatabaseConnection
1802 {
1803     private static DatabaseConnection _instance = new DatabaseConnection();
1804
1805     private DatabaseConnection() {}
1806
1807     public static DatabaseConnection GetInstance()
1808     {
1809         return _instance;
1810     }
1811 }
1812
```

1813 Note: You control the instance creation manually.

1814
1815
1816
1817
1818
1819
1820 58. Can a class implement multiple interfaces with same method names?

1821
1822 Answer:

1823 Note: Yes, but you have to use explicit implementation to resolve ambiguity.

1824
1825 Example:

```

1826
1827 interface IReadable
1828 {
1829     void Display();
1830 }
1831
1832 interface IWritable
1833 {
1834     void Display();
1835 }
1836
1837 class Document : IReadable, IWritable
1838 {
1839     void IReadable.Display()
1840     {
1841         Console.WriteLine("Reading Document");
1842     }
1843
1844     void IWritable.Display()
1845     {
1846         Console.WriteLine("Writing Document");
1847     }
1848 }
1849
1850 Usage:
1851
1852 IReadable readDoc = new Document();
1853 readDoc.Display();
1854
1855 IWritable writeDoc = new Document();
1856 writeDoc.Display();
1857
1858 Note: Perfect separation of behaviors even if method names match.
1859
1860
1861
1862
1863
1864

```

59. What is the difference between interface and abstract class with examples?

Answer:

Feature	Interface	Abstract Class
Methods	Only signatures (until C#8)	Can have implementations
Fields	No fields	Can have fields and properties
Inheritance	Multiple interfaces supported	Only single class inheritance
Constructor	No constructor allowed	Constructors allowed

Example Interface:

```

1879 interface IVehicle
1880 {
1881     void Start();
1882 }
1883

```

Example Abstract Class:

```

1886 abstract class Vehicle
1887 {
1888     public abstract void Start();
1889     public void Horn()
1890     {
1891         Console.WriteLine("Beep Beep");
1892     }
1893 }
1894

```

1895 Note: Use interface for behavior contract,
1896 Note: Use abstract class for shared behavior + contract.

1897

1898

1899

1900

1901

1902 60. What happens if you don't provide implementation for all interface methods?

1903

1904 Answer:

1905

1906 If a class does not implement all interface methods,

1907

1908 The compiler throws an error unless the class is declared abstract.

1909

1910 Example:

1911

1912 interface IAnimal

1913 {

1914 void Eat();

1915 void Sleep();

1916 }

1917

1918 class Dog : IAnimal

1919 {

1920 public void Eat()

1921 {

1922 Console.WriteLine("Dog eating");

1923 }

1924

1925 // Missing Sleep() => Compiler Error }

1926

1927 Note: Always implement all interface methods unless you make the class abstract.

1928

1929

1930 61. Can an interface inherit another interface? Can a class inherit multiple interfaces?

1931

1932 Answer:

1933 Note: Yes, interfaces can inherit other interfaces.

1934 Note: Yes, a class can implement multiple interfaces.

1935

1936 Example:

1937

1938 interface IFirst

1939 {

1940 void MethodA();

1941 }

1942

1943 interface ISecond : IFirst

1944 {

1945 void MethodB();

1946 }

1947

1948 class Implementation : ISecond

1949 {

1950 public void MethodA()

1951 {

1952 Console.WriteLine("MethodA executed");

1953 }

1954

1955 public void MethodB()

1956 {

1957 Console.WriteLine("MethodB executed");

1958 }

1959 }

1960

1961 Note: Interfaces can extend multiple interfaces too: interface ICombined : IFirst,
1962 IOther.

1963 62. What is the difference between method overloading and method overriding?

1964

1965 Answer:

1966

1967

Feature	Method Overloading	Method Overriding
=====		
Purpose behavior	Same name, different parameters	Modify inherited method
Compile time/runtime	Compile-time polymorphism	Runtime polymorphism
Keyword used	None	override, virtual, abstract

1973

1974 Example of overloading:

1975

1976 class Calculator

1977 {

1978 public int Add(int a, int b) => a + b;

1979 public float Add(float a, float b) => a + b;

1980 }

1981

1982 Example of overriding:

1983

1984 class Parent

1985 {

1986 public virtual void Show()

1987 {

1988 Console.WriteLine("Parent Show");

1989 }

1990 }

1991

1992 class Child : Parent

1993 {

1994 public override void Show()

1995 {

1996 Console.WriteLine("Child Show");

1997 }

1998 }

1999

2000 Note: Overloading: same method name, different arguments.

2001 Note: Overriding: same method signature, different behavior.

2002

2003

2004

2005

2006

2007 63. What is boxing and unboxing in C#?

2008

2009 Answer:

2010 Note: Boxing: Converting a value type (e.g., int) into an object.

2011 Note: Unboxing: Extracting the value type from an object.

2012

2013 Example:

2014

2015 int number = 10;

2016 object obj = number; // Boxing

2017 int result = (int)obj; // Unboxing

2018

2019 Note: Boxing moves value type to heap (object type),

2020 Note: Unboxing extracts back the original value.

2021

2022 64. What is an indexer in C#?

2023

2024 Answer:

2025

2026 Indexers allow an object to be indexed like an array.

2027

2028 Syntax similar to array but at the object level.

2029

2030 Example:

```
2031
2032 class SampleCollection
2033 {
2034     private string[] data = new string[5];
2035
2036     public string this[int index]
2037     {
2038         get { return data[index]; }
2039         set { data[index] = value; }
2040     }
2041 }
2042
2043 Usage:
2044
2045 SampleCollection collection = new SampleCollection();
2046 collection[0] = "Hello";
2047 Console.WriteLine(collection[0]);
2048
2049 Note: Indexers help in custom array-like behavior for your own classes.
2050
2051
2052
2053
2054 65. What is a delegate in C#? How is it different from an event?
2055
2056 Answer:
2057
2058 A delegate is a function pointer (type-safe).
2059
2060 An event is a wrapper over a delegate to restrict direct invocation.
2061
2062 Example:
2063
2064 delegate void Notify();
2065
2066 class Process
2067 {
2068     public static void Task()
2069     {
2070         Console.WriteLine("Process Started");
2071     }
2072 }
2073
2074 Usage:
2075
2076 Notify notify = Process.Task;
2077 notify();
2078
2079 Note: Delegates point to methods.
2080 Note: Events control access so that only subscribers can trigger.
2081
2082 Example Event:
2083
2084 class Alarm
2085 {
2086     public event Notify Ring;
2087
2088     public void Trigger()
2089     {
2090         Ring?.Invoke();
2091     }
2092 }
2093
2094 Note: Events restrict outsiders from accidentally invoking the delegate.
2095
2096
2097
2098
2099
```

2100 66. What is a multicast delegate in C#?

2101

2102 Answer:

2103 Note: A multicast delegate points to multiple methods.

2104 Note: When invoked, it calls all the methods sequentially.

2105

2106 Example:

2107

2108 delegate void Notify();

2109

2110 class MulticastExample

2111 {

2112 public static void Method1()

2113 {

2114 Console.WriteLine("Method1 Called");

2115 }

2116

2117 public static void Method2()

2118 {

2119 Console.WriteLine("Method2 Called");

2120 }

2121 }

2122

2123 Usage:

2124

2125 Notify notify = MulticastExample.Method1;

2126 notify += MulticastExample.Method2;

2127 notify();

2128

2129 Note: Both Method1 and Method2 get called when notify() is invoked.

2130

2131

2132

2133

2134

2135

2136 67. What are anonymous methods in C#?

2137

2138 Answer:

2139

2140 Anonymous Methods are methods without a name.

2141

2142 They are assigned directly to a delegate.

2143

2144 Example:

2145

2146 delegate void Notify(string message);

2147

2148 class Program

2149 {

2150 static void Main()

2151 {

2152 Notify notify = delegate(string msg)

2153 {

2154 Console.WriteLine("Notification: " + msg);

2155 };

2156

2157 notify("Task Completed!");

2158 }

2159 }

2160

2161 Note: Anonymous methods are helpful for small inline tasks without creating separate methods.

2162

2163

2164

2165

2166

2167

2168 68. What are lambda expressions in C#?
2169
2170 Answer:
2171
2172 Lambda expressions are shorthand for anonymous methods.
2173
2174 Syntax: (parameters) => expression
2175

2176 Example:

```
2177 delegate int Square(int num);  
2178  
2179 class Program  
2180 {  
2181     static void Main()  
2182     {  
2183         Square square = x => x * x;  
2184         Console.WriteLine(square(5));  
2185     }  
2186 }  
2187
```

2188
2189 Note: Lambda makes the code cleaner and readable.
2190
2191
2192
2193
2194
2195
2196
2197

2198 69. What is the difference between Func, Action, and Predicate?
2199

2200 Answer:

Type	Signature	Return type
Func	Takes parameters, returns value	Value (int, string, etc.)
Action	Takes parameters, returns nothing (void)	Void
Predicate	Takes one parameter, returns bool	bool (true/false)

2208
2209 Examples:

```
2210  
2211 Func<int, int, int> add = (a, b) => a + b;  
2212 Action<string> greet = name => Console.WriteLine("Hello " + name);  
2213 Predicate<int> isEven = num => num % 2 == 0;  
2214
```

2215 Usage:

```
2216  
2217 Console.WriteLine(add(3, 4));  
2218 greet("Karthik");  
2219 Console.WriteLine(isEven(10));  
2220
```

2221 Note: Func: Useful for computations,
2222 Note: Action: Useful for performing actions,
2223 Note: Predicate: Useful for conditions/checks.
2224
2225
2226
2227
2228

2229 70. What is an event handler delegate signature in C#?
2230

2231 Answer:

2232 The standard pattern for .NET event handlers:

2233
2234 Return type: void

2235
2236 Parameters: object sender, EventArgs e

2237
2238 Example:
2239
2240 public delegate void EventHandler(object sender, EventArgs e);
2241

2242 Example of usage:

2243
2244 class Alarm
2245 {
2246 public event EventHandler Ring;
2247
2248 public void Trigger()
2249 {
2250 if (Ring != null)
2251 {
2252 Ring(this, EventArgs.Empty);
2253 }
2254 }
2255 }

2256
2257 Note: Follows the EventHandler delegate signature in all standard .NET events.
2258
2259
2260

2261
2262
2263 71. What is covariance and contravariance in C#?
2264

2265 Answer:

2266 Note: Covariance: Allows a method to return a more derived type than originally specified.

2267 Note: Contravariance: Allows a method to accept parameters of less derived types.

2268
2269 Example of Covariance with return types:

2270
2271 IEnumerable<string> names = new List<string>();
2272 IEnumerable<object> objects = names; // Note: Covariance allowed because string → object
2273

2274 Example of Contravariance with parameters:

2275
2276 Action<object> actObject = (obj) => Console.WriteLine(obj);
2277 Action<string> actString = actObject; // Note: Contravariance allowed because string is
a derived type of object

2278
2279 Note: Covariance → "Output flexibility"

2280 Note: Contravariance → "Input flexibility"
2281
2282
2283
2284
2285

2286 72. What is the difference between Task and Thread in C#?
2287

2288 Answer:

2289
2290
2291 Feature Task Thread
2292 =====
2293 Managed by Task Scheduler OS Thread Pool
2294 Lightweight Yes No
2295 Use case For async operations For manual thread management
2296 Creation cost Less More
2297

2298 Example:

2299 Task.Run(() => Console.WriteLine("Task Running"));
2300

2301 vs

2302
2303 Thread thread = new Thread(() => Console.WriteLine("Thread Running"));

```
2304     thread.Start();
2305
2306     Note:   Prefer Task for modern, scalable, async operations.
2307
2308
2309
2310
2311
2312     73. What is deadlock? How do you avoid it in C#?
2313
2314     Answer:
2315     Note:   Deadlock occurs when two or more threads are waiting for each other's resources,
2316             causing an infinite wait.
2317
2318     Example of Deadlock:
2319
2320     object lock1 = new object();
2321     object lock2 = new object();
2322
2323     Thread t1 = new Thread(() =>
2324     {
2325         lock (lock1)
2326         {
2327             Thread.Sleep(1000);
2328             lock (lock2) { }
2329         }
2330     });
2331
2332     Thread t2 = new Thread(() =>
2333     {
2334         lock (lock2)
2335         {
2336             lock (lock1) { }
2337         }
2338     });
2339
2340     t1.Start();
2341     t2.Start();
2342
2343     Note:   Avoid deadlocks by:
2344
2345     Always locking resources in the same order.
2346
2347     Using timeout patterns (e.g., Monitor.TryEnter()).
2348
2349
2350
2351
2352     74. What is async and await in C#?
2353
2354     Answer:
2355
2356     async enables a method to be asynchronous (non-blocking).
2357
2358     await pauses method execution until awaited Task completes.
2359
2360     Example:
2361
2362     async Task<int> GetNumberAsync()
2363     {
2364         await Task.Delay(1000);
2365         return 5;
2366     }
2367
2368     Usage:
2369
2370     var result = await GetNumberAsync();
2371     Console.WriteLine(result);
```

2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439

Note: Helps improve scalability without blocking threads.

75. Explain the difference between IEnumerable and IQueryable.

Answer:

Feature	IEnumerable	IQueryable
Where evaluated	In memory (client-side)	At database (server-side)
When evaluated	Deferred execution	Deferred execution
Suitable for	In-memory collections (List, Array)	Remote data sources

Example:

```
IEnumerable<int> localQuery = new List<int> {1,2,3,4}.Where(x => x > 2);
```

vs

```
IQueryable<int> dbQuery = dbContext.Employees.Where(e => e.Id > 100);
```

Note: Use IEnumerable for in-memory data,

Note: Use IQueryable for database-efficient queries.

76. What is a static class in C#? Can it have constructors?

Answer:

Note: A static class:

Cannot be instantiated.

Can only have static members.

Can have a static constructor (executed once when class is loaded).

Example:

```
static class MathHelper
{
    static MathHelper()
    {
        Console.WriteLine("Static Constructor Called");
    }

    public static int Add(int a, int b)
    {
        return a + b;
    }
}
```

Usage:

```
Console.WriteLine(MathHelper.Add(3, 4));
```

Note: Static classes are used for utility/helper methods.

77. What is reflection in C#? How can you use it?

```

2440
2441 Answer:
2442 Note: Reflection is the ability to inspect metadata (types, methods, properties) at
runtime.
2443
2444 Example:
2445
2446 Type type = typeof(string);
2447 Console.WriteLine("Type Name: " + type.Name);
2448 foreach (var method in type.GetMethods())
2449 {
2450     Console.WriteLine("Method: " + method.Name);
2451 }
2452
2453 Note: Reflection is used for:
2454
2455 Dynamic loading,
2456
2457 Plugin architectures,
2458
2459 Inspecting attributes.
2460
2461 Be careful: Reflection is slow compared to normal access.
2462
2463
2464
2465
2466
2467
2468 78. What is the difference between early binding and late binding using reflection?
2469
2470 Answer:
2471
2472
2473 Feature                Early Binding                Late Binding (Reflection)
2474 =====
2475 Method known           At compile time           Only at runtime
2476 Speed                  Fast                       Slower
2477 Flexibility            Rigid (fixed types)       Very flexible (dynamic types)
2478
2479
2480 Example Early Binding:
2481
2482 string text = "Hello";
2483 Console.WriteLine(text.ToUpper());
2484
2485 Example Late Binding:
2486
2487 object textObj = "Hello";
2488 Type t = textObj.GetType();
2489 MethodInfo method = t.GetMethod("ToUpper");
2490 object result = method.Invoke(textObj, null);
2491 Console.WriteLine(result);
2492
2493 Note: Reflection allows calling methods without knowing their name at compile time.
2494
2495
2496
2497
2498
2499 79. How do you create a custom attribute in C#?
2500
2501 Answer:
2502 Note: Custom attributes allow you to attach metadata to classes, methods, properties.
2503
2504 Example:
2505
2506 [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
2507 public class AuthorAttribute : Attribute

```

```
2508 {
2509     public string Name { get; set; }
2510     public AuthorAttribute(string name)
2511     {
2512         Name = name;
2513     }
2514 }
```

2515
2516 Usage:

```
2517
2518 [Author("Karthik")]
2519 class MyClass
2520 {
2521     public void DoWork() { }
2522 }
2523
```

2524 Note: Attributes can later be retrieved via Reflection.

2525
2526
2527
2528
2529
2530 80. How do you read custom attributes using reflection?

2531
2532 Answer:

2533 Note: After defining custom attributes, you can read them dynamically.

2534
2535 Example:

```
2536
2537 Type type = typeof(MyClass);
2538 object[] attributes = type.GetCustomAttributes(typeof(AuthorAttribute), true);
2539 foreach (AuthorAttribute attr in attributes)
2540 {
2541     Console.WriteLine("Author: " + attr.Name);
2542 }
2543
```

2544 Note: Reflection fetches metadata and enables dynamic behaviors.

2545
2546
2547
2548
2549
2550
2551 81. What is a partial class in C#?

2552
2553 Answer:

2554 Note: A partial class allows a class to be split across multiple files.

2555 Note: At compile time, all parts are combined into a single class.

2556
2557 Example:

2558
2559 First file (Person1.cs):

```
2560
2561 public partial class Person
2562 {
2563     public string FirstName;
2564 }
2565
```

2566
2567 Second file (Person2.cs):

```
2568 public partial class Person
2569 {
2570     public string LastName;
2571 }
2572
```

2573 Usage:

```
2574
2575 Person p = new Person();
2576 p.FirstName = "Karthik";
```

2577 p.LastName = "Muthukrishnan";

2578

2579 Note: Partial classes are helpful for:

2580

2581 Large classes,

2582

2583 Auto-generated code (e.g., Designer files in WinForms).

2584

2585

2586

2587

2588

2589

2590 82. What is a partial method in C#?

2591

2592 Answer:

2593 Note: A partial method is a special method inside a partial class:

2594

2595 It can be optionally implemented.

2596

2597 If not implemented, the compiler removes its call (no error, no code).

2598

2599 Example:

2600

2601 In first file:

2602

2603 public partial class Demo

2604 {

2605 partial void Log(string message);

2606

2607 public void Run()

2608 {

2609 Log("Running Demo");

2610 }

2611 }

2612

2613 In second file:

2614

2615 public partial class Demo

2616 {

2617 partial void Log(string message)

2618 {

2619 Console.WriteLine("Log: " + message);

2620 }

2621 }

2622

2623 Note: Partial methods are lightweight hooks for optional logic.

2624

2625

2626

2627

2628

2629

2630

2631 83. What is serialization in C#? What are types of serialization?

2632

2633 Answer:

2634 Note: Serialization is the process of converting an object into a format that can be persisted (file, database) or transmitted (over network).

2635

2636 Types of Serialization:

2637

2638 1. Binary Serialization (compact, .bin file)

2639

2640 2. XML Serialization (human-readable)

2641

2642 3. JSON Serialization (modern, APIs)

2643

2644 4. SOAP Serialization (older web services)

```

2645
2646 Example JSON serialization:
2647
2648 using System.Text.Json;
2649
2650 Employee emp = new Employee { Id = 1, Name = "Karthik" };
2651 string jsonString = JsonSerializer.Serialize(emp);
2652 Console.WriteLine(jsonString);
2653
2654 Note: Serialization is important for data storage, API communication, and distributed
      systems.

```

```

2655
2656
2657
2658
2659
2660
2661
2662

```

2663 84. What is the difference between shallow copy and deep copy in serialization?

```

2664
2665 Answer:

```

```

2666
2667

```

Feature	Shallow Copy	Deep Copy
=====	=====	=====
What is copied	Only references are copied	Entire objects are copied
Effect	Changes affect both objects	Independent copies
How to achieve	MemberwiseClone()	Serialization/Manual copy

```

2673
2674 Example Deep Copy using JSON:

```

```

2675
2676 string json = JsonSerializer.Serialize(originalObject);
2677 var deepCopy = JsonSerializer.Deserialize<Employee>(json);
2678

```

2679 Note: Deep copy ensures full independence between copies.

```

2680
2681
2682
2683
2684
2685

```

2686 85. How do you create a custom exception in C#?

```

2687
2688 Answer:

```

2689 Note: Create a class that inherits from Exception.

```

2690
2691 Example:

```

```

2692
2693

```

```

2693 public class InvalidAgeException : Exception
2694 {
2695     public InvalidAgeException(string message) : base(message)
2696     {
2697     }
2698 }

```

```

2699
2700 Usage:

```

```

2701
2702

```

```

2702 int age = 15;
2703 if (age < 18)
2704 {
2705     throw new InvalidAgeException("Age must be 18 or older.");
2706 }

```

```

2707
2708 Note: Custom exceptions are useful for domain-specific error handling.

```

```

2709
2710
2711
2712

```


86. What is the difference between String and StringBuilder in C#?

Answer:

Feature	String	StringBuilder
=====		
Mutability	Immutable (modifications create new object)	Mutable (modifications happen in-place)
Performance	Poor for frequent modifications	High performance for modifications
Usage	Small and simple text	Large and dynamic text changes

Example of String:

```
string str = "Hello";  
str += " World";  
Console.WriteLine(str);
```

Example of StringBuilder:

```
StringBuilder sb = new StringBuilder("Hello");  
sb.Append(" World");  
Console.WriteLine(sb.ToString());
```

Note: Prefer StringBuilder for heavy text modifications like loops and concatenations.

87. What are extension methods in C#?

Answer:

Note: Extension methods allow you to "add" methods to existing types without modifying them.

Example:

```
public static class StringExtensions  
{  
    public static int WordCount(this string str)  
    {  
        return str.Split(' ').Length;  
    }  
}
```

Usage:

```
string text = "Hello from Karthik";  
Console.WriteLine(text.WordCount());
```

Note: Extension methods make your code more readable and fluent.

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

```

2778
2779 Answer:
2780
2781
2782 Feature      IEnumerable      IEnumerator
2783 =====
2784 Purpose      Collection that can be iterated      Cursor that iterates the collection
2785 Method        GetEnumerator()      MoveNext(), Current, Reset()
2786 Used with     foreach loops      Inside foreach internals
2787
2788 Example:
2789
2790 class Numbers : IEnumerable
2791 {
2792     public IEnumerator GetEnumerator()
2793     {
2794         yield return 1;
2795         yield return 2;
2796         yield return 3;
2797     }
2798 }
2799
2800 Usage:
2801
2802 Numbers numbers = new Numbers();
2803 foreach (int n in numbers)
2804 {
2805     Console.WriteLine(n);
2806 }
2807
2808 Note: IEnumerable exposes enumeration behavior,
2809 Note: IEnumerator controls the navigation logic.
2810
2811
2812
2813
2814
2815
2816
2817 89. What is Dependency Injection (DI) in C#?
2818
2819 Answer:
2820 Note: Dependency Injection is a design pattern to inject an object's dependencies from
2821 outside instead of creating them inside the class.
2822
2823 Three common types:
2824
2825 1. Constructor Injection
2826
2827 2. Setter Injection
2828
2829 3. Method Injection
2830
2831 Example - Constructor Injection:
2832
2833 class Service
2834 {
2835     public void Serve()
2836     {
2837         Console.WriteLine("Service Called");
2838     }
2839 }
2840
2841 class Client
2842 {
2843     private Service _service;
2844
2845     public Client(Service service)
2846     {

```

```

2846         _service = service;
2847     }
2848
2849     public void Start()
2850     {
2851         _service.Serve();
2852     }
2853 }

```

2854 Usage:

```

2856 Service service = new Service();
2857 Client client = new Client(service);
2858 client.Start();

```

2859 Note: DI improves testability, decoupling, and maintainability.

2868 90. What is the difference between Singleton and Static Class?

2871 Answer:

2874 Feature	2874 Singleton	2874 Static Class
2875 =====		
2876 Object creation	2876 Only one instance allowed	2876 No instance allowed
2877 Memory management	2877 Created on demand	2877 Always loaded in memory
2878 Inheritance	2878 Can implement interfaces	2878 Cannot inherit interfaces

2881 Example Singleton:

```

2883 class Singleton
2884 {
2885     private static Singleton _instance;
2886     private Singleton() {}
2887
2888     public static Singleton Instance
2889     {
2890         get
2891         {
2892             if (_instance == null)
2893                 _instance = new Singleton();
2894             return _instance;
2895         }
2896     }
2897 }

```

2899 Note: Singleton gives controlled object creation,

2900 Note: Static class gives grouped static methods without object management.

2910 91. What is the use of 'yield' keyword in C#?

2912 Answer:

2913 Note: The yield keyword allows you to iterate items one-by-one, maintaining the state between calls without creating a collection.

```

2914
2915 Example:
2916
2917 public static IEnumerable<int> GetNumbers()
2918 {
2919     yield return 1;
2920     yield return 2;
2921     yield return 3;
2922 }
2923
2924 Usage:
2925
2926 foreach (var number in GetNumbers())
2927 {
2928     Console.WriteLine(number);
2929 }
2930
2931 Note: yield return simplifies creating custom iterators without manual IEnumerator
implementation.

```

```

2932
2933
2934
2935
2936
2937
2938 92. What is the 'lock' statement in C#?
2939

```

```

2940 Answer:
2941 Note: The lock statement ensures that a critical section of code is executed by only
one thread at a time.

```

```

2942
2943 Example:
2944
2945 private static readonly object _lockObject = new object();
2946
2947 public void CriticalMethod()
2948 {
2949     lock (_lockObject)
2950     {
2951         // Only one thread can execute here at a time
2952         Console.WriteLine("Critical Section Accessed");
2953     }
2954 }
2955

```

```

2956 Note: lock prevents race conditions in multi-threaded environments.
2957
2958
2959
2960
2961
2962

```

```

2963
2964 93. What is the difference between 'const', 'readonly', and 'static readonly'?
2965

```

```

2966 Answer:
2967

```

Feature	const	readonly	static readonly
Value Set	At compile-time	At runtime (in constructor)	At runtime (only once in static constructor)
Modifiability	No	No	No
Context	Static by default	Instance-level or Static	Static

```

2974
2975 Example of const:
2976

```

```

2977 public const int MaxItems = 100;
2978

```

```

2979 Example of readonly:

```

```
2980
2981 public readonly int CreatedAt = DateTime.Now.Year;
2982
2983 Example of static readonly:
2984
2985 public static readonly int StaticValue;
2986
2987 static ClassName()
2988 {
2989     StaticValue = 10;
2990 }
2991
2992 Note: Use:
2993
2994 const for pure compile-time constants,
2995
2996 readonly for instance constants,
2997
2998 static readonly for class-level constants set at runtime.
2999
3000
3001
3002
3003
3004
3005
3006 94. What is method hiding in C#?
3007
3008 Answer:
3009 Note: Method hiding happens when a derived class defines a new method with the same
3010 name as in base class, but does not override it.
3011
3012 Use new keyword to indicate method hiding.
3013
3014 Example:
3015
3016 class Base
3017 {
3018     public void Display()
3019     {
3020         Console.WriteLine("Base Display");
3021     }
3022 }
3023
3024 class Derived : Base
3025 {
3026     public new void Display()
3027     {
3028         Console.WriteLine("Derived Display");
3029     }
3030 }
3031
3032 Usage:
3033
3034 Base b = new Derived();
3035 b.Display(); // Calls Base.Display (not Derived)
3036
3037 Note: Method hiding does not replace base method behavior unless explicitly casted.
3038
3039
3040
3041
3042
3043
3044 95. What is a sealed class in C#?
3045
3046 Answer:
3047 Note: A sealed class cannot be inherited.
```

3048 Note: Use sealed keyword to prevent derivation.

3049

3050 Example:

3051

```
3052 sealed class FinalClass
```

```
3053 {
```

```
3054     public void Show()
```

```
3055     {
```

```
3056         Console.WriteLine("Final Class Method");
```

```
3057     }
```

```
3058 }
```

3059

3060 Note: Attempting to inherit from a sealed class will cause compile-time error.

3061

3062 Note: Sealed classes are used for:

3063

3064 Security (prevent overriding critical behavior),

3065

3066 Performance (JIT can optimize calls).

3067

3068

3069

3070

3071 96. What is an abstract class and how is it different from an interface?

3072

3073 Answer:

3074 Note: Abstract class:

3075

3076 Can have implemented and unimplemented methods.

3077

3078 Can have fields, properties, constructors.

3079

3080 Note: Interface:

3081

3082 Can only have method signatures (until C# 8.0, now can have default methods too).

3083

3084 Cannot have fields.

3085

3086 Example of Abstract Class:

3087

```
3088 abstract class Animal
```

```
3089 {
```

```
3090     public abstract void Sound();
```

```
3091
```

```
3092     public void Sleep()
```

```
3093     {
```

```
3094         Console.WriteLine("Sleeping...");
```

```
3095     }
```

```
3096 }
```

3097

```
3098 class Dog : Animal
```

```
3099 {
```

```
3100     public override void Sound()
```

```
3101     {
```

```
3102         Console.WriteLine("Bark");
```

```
3103     }
```

```
3104 }
```

3105

3106 Note: Use abstract class when:

3107

3108 You want common functionality + force derived classes to implement some behavior.

3109

3110

3111

3112

3113

3114

3115 97. What is a default interface method (C# 8.0 onwards)?

3116

```

3117 Answer:
3118 Note: In C# 8.0+, interfaces can have default method implementations!
3119
3120 Example:
3121
3122 interface ILogger
3123 {
3124     void Log(string message)
3125     {
3126         Console.WriteLine("Logging: " + message);
3127     }
3128 }
3129
3130 Note: Now classes implementing ILogger are not forced to override Log() unless they
3131 want to.
3132
3133 Note: Default interface methods make interfaces more flexible without breaking old
3134 implementations.
3135
3136
3137

```

3138 98. What is a Tuple in C#?

```

3139 Answer:
3140 Note: A Tuple is a lightweight object for grouping multiple values together.
3141
3142 Example:
3143
3144 var employee = Tuple.Create(101, "Karthik", "Developer");
3145 Console.WriteLine($"{employee.Item1} - {employee.Item2} - {employee.Item3}");
3146
3147 Note: From C# 7 onwards: You can use ValueTuple syntax:
3148
3149 (var id, var name, var role) = (101, "Karthik", "Developer");
3150 Console.WriteLine($"{id} - {name} - {role}");
3151
3152 Note: Tuples are great for returning multiple values from a method easily.
3153
3154
3155
3156
3157
3158
3159

```

3160 99. What is the difference between Value Type and Reference Type in C#?

3161 Answer:

Feature	Value Type	Reference Type
Stored in	Stack	Heap
Example types	int, double, struct	class, interface, array, string
Assignment	Copies value	Copies reference
Nullability	Cannot be null (except nullable types)	Can be null

3171 Example of Value Type:

```

3172
3173 int x = 5;
3174 int y = x;
3175 y = 10;
3176 Console.WriteLine(x); // Outputs 5
3177

```

3178 Example of Reference Type:

```

3179
3180 class Person
3181 {

```

```
3184         public string Name;
3185     }
3186
3187     Person p1 = new Person();
3188     p1.Name = "Karthik";
3189     Person p2 = p1;
3190     p2.Name = "Rajesh";
3191     Console.WriteLine(p1.Name); // Outputs "Rajesh"
3192
3193     Note: Value types store data,
3194     Note: Reference types store memory address (reference).
3195
3196
3197
3198
3199
3200
3201
3202     100. What is Nullable type in C#?
3203
3204     Answer:
3205     Note: Nullable types allow value types (int, double, etc.) to represent null values.
3206
3207     Example:
3208
3209     int? age = null;
3210
3211     if (age.HasValue)
3212         Console.WriteLine(age.Value);
3213     else
3214         Console.WriteLine("Age is not set");
3215
3216     Note: Nullable types are crucial for:
3217
3218     Database operations,
3219
3220     Optional fields,
3221
3222     Handling missing values safely.
3223
3224     Shortcut syntax:
3225
3226     Nullable<int> age1 = 30; // same as int? age1 = 30;
3227
3228     Note: Avoids null reference exceptions by checking .HasValue before accessing .Value.
```