C# The Programming Language Part 2

Exercise Workbook

Learn. To Change.

Object Oriented Programming Principles

A review of the OOP concepts presented in the Foundations course

Q1

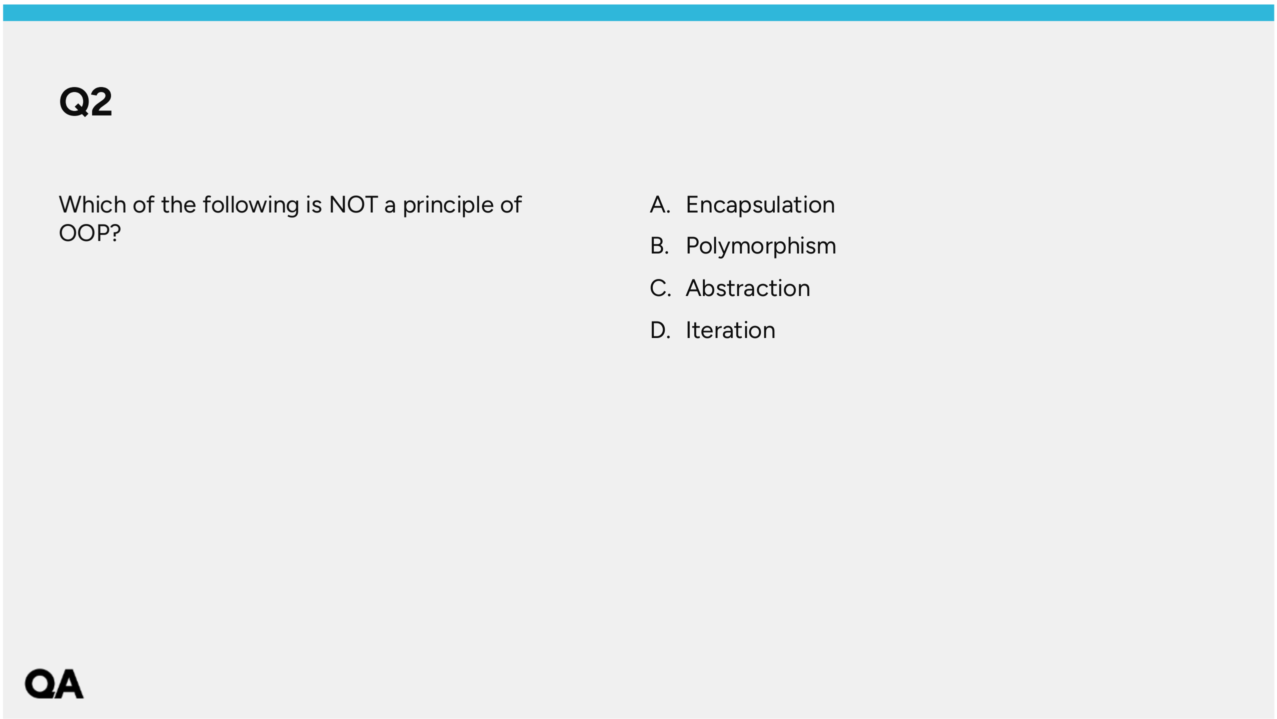
Which of the following best defines Object-Oriented Programming?

A. A programming paradigm based on functions and procedures.

B. A programming style that focuses on using classes and objects to design applications.

C. A style of programming focused entirely on data manipulation using loops and conditionals.

D. A method of coding without any use of predefined libraries.



Q2

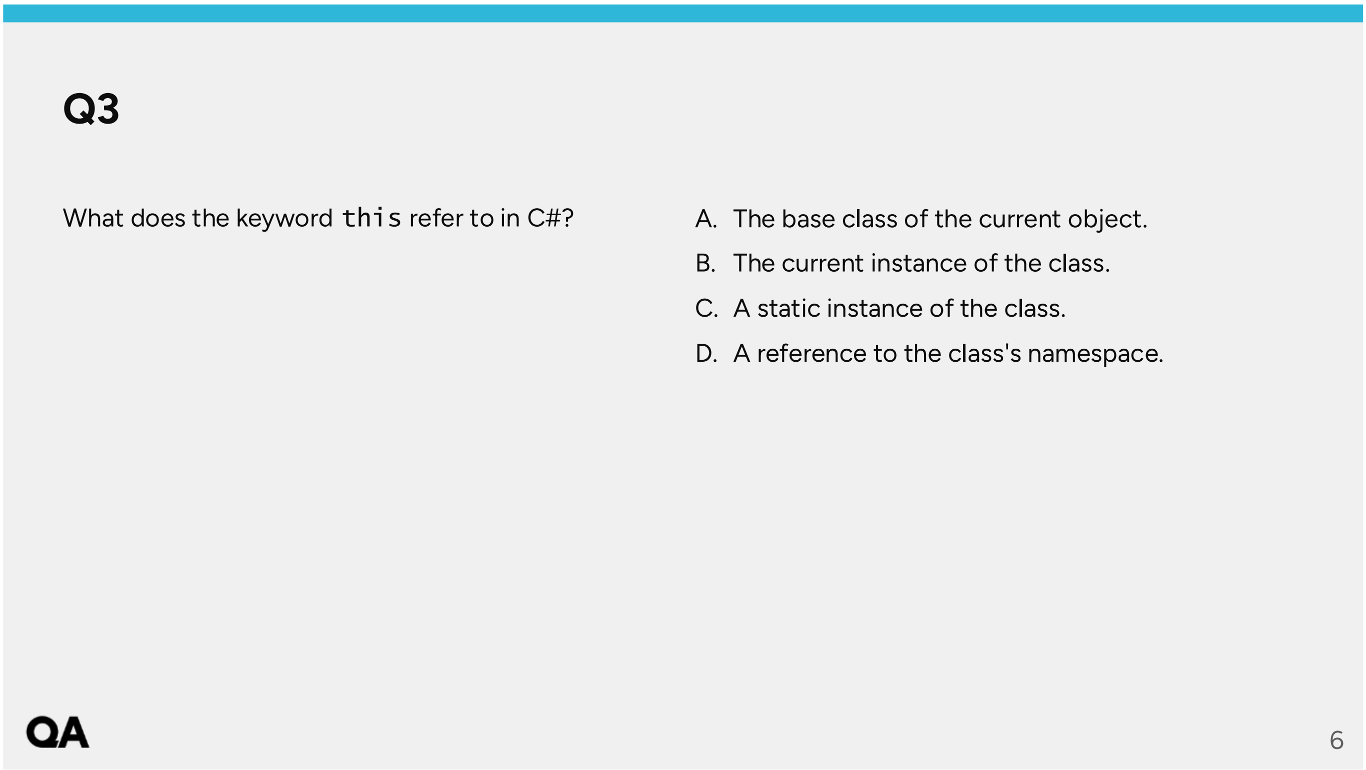
Which of the following is NOT a principle of OOP?

A. Encapsulation

B. Polymorphism

C. Abstraction

D. Iteration

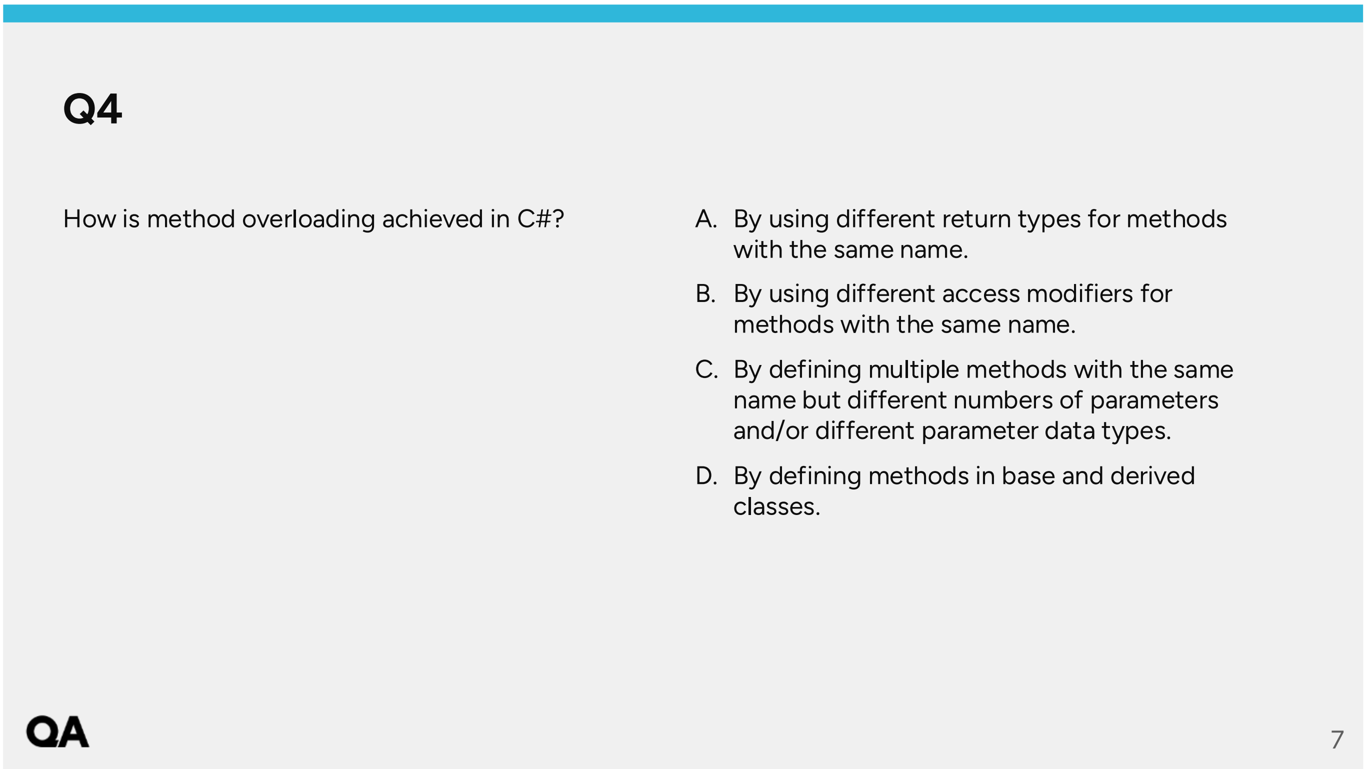


Q3

What does the keyword `this` refer to in C#?

A. The base class of the current object.
B. The current instance of the class.
C. A static instance of the class.
D. A reference to the class's namespace.

6

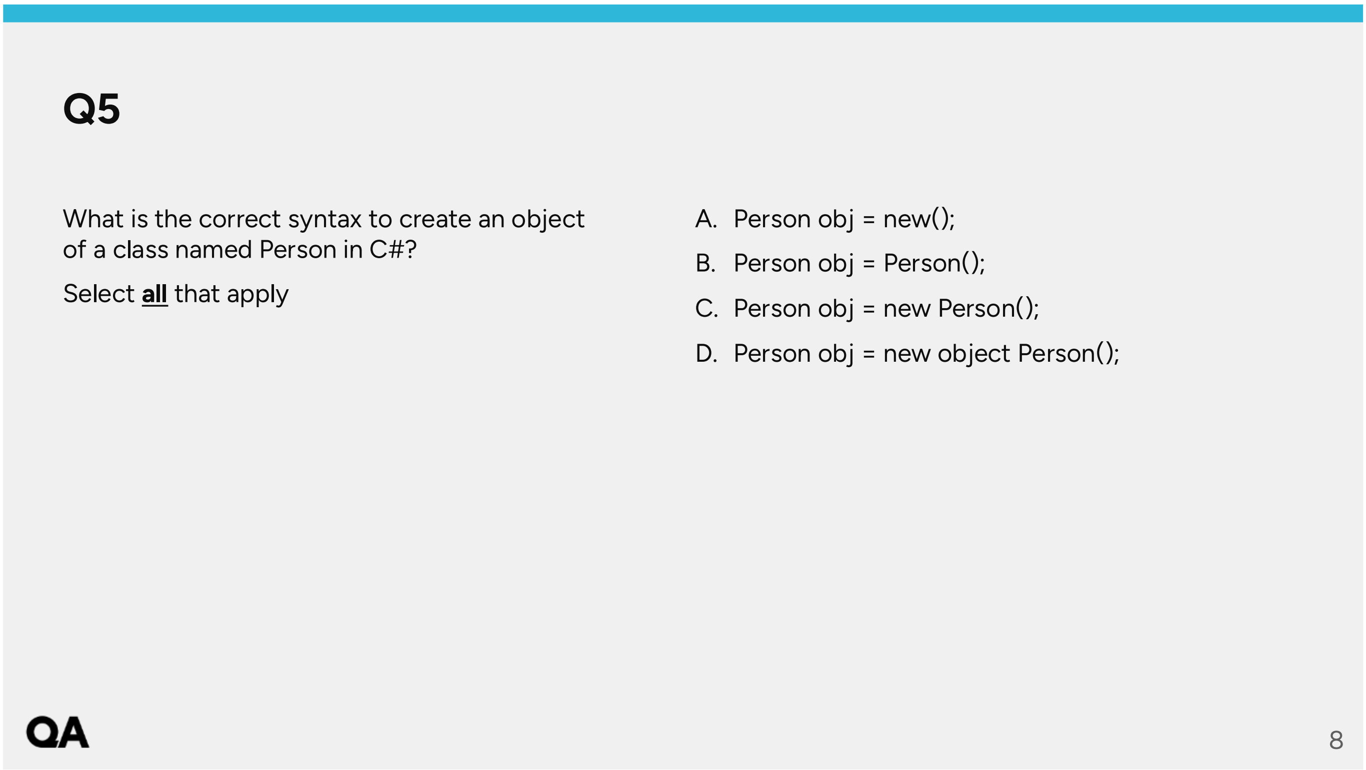


Q4

How is method overloading achieved in C#?

A. By using different return types for methods with the same name.
B. By using different access modifiers for methods with the same name.
C. By defining multiple methods with the same name but different numbers of parameters and/or different parameter data types.
D. By defining methods in base and derived classes.

7



Q5

What is the correct syntax to create an object of a class named Person in C#?

Select **all** that apply

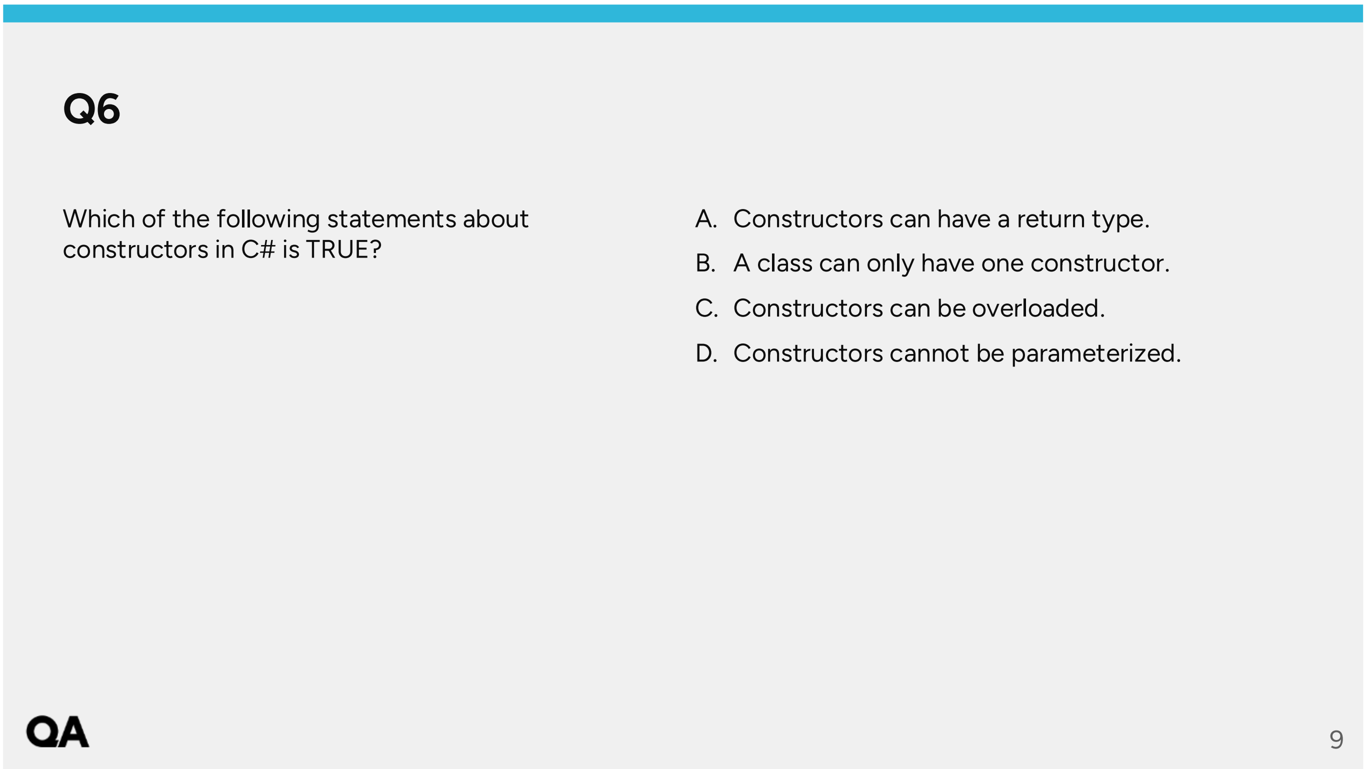
A. Person obj = new();

B. Person obj = Person();

C. Person obj = new Person();

D. Person obj = new object Person();

8



Q6

Which of the following statements about constructors in C# is TRUE?

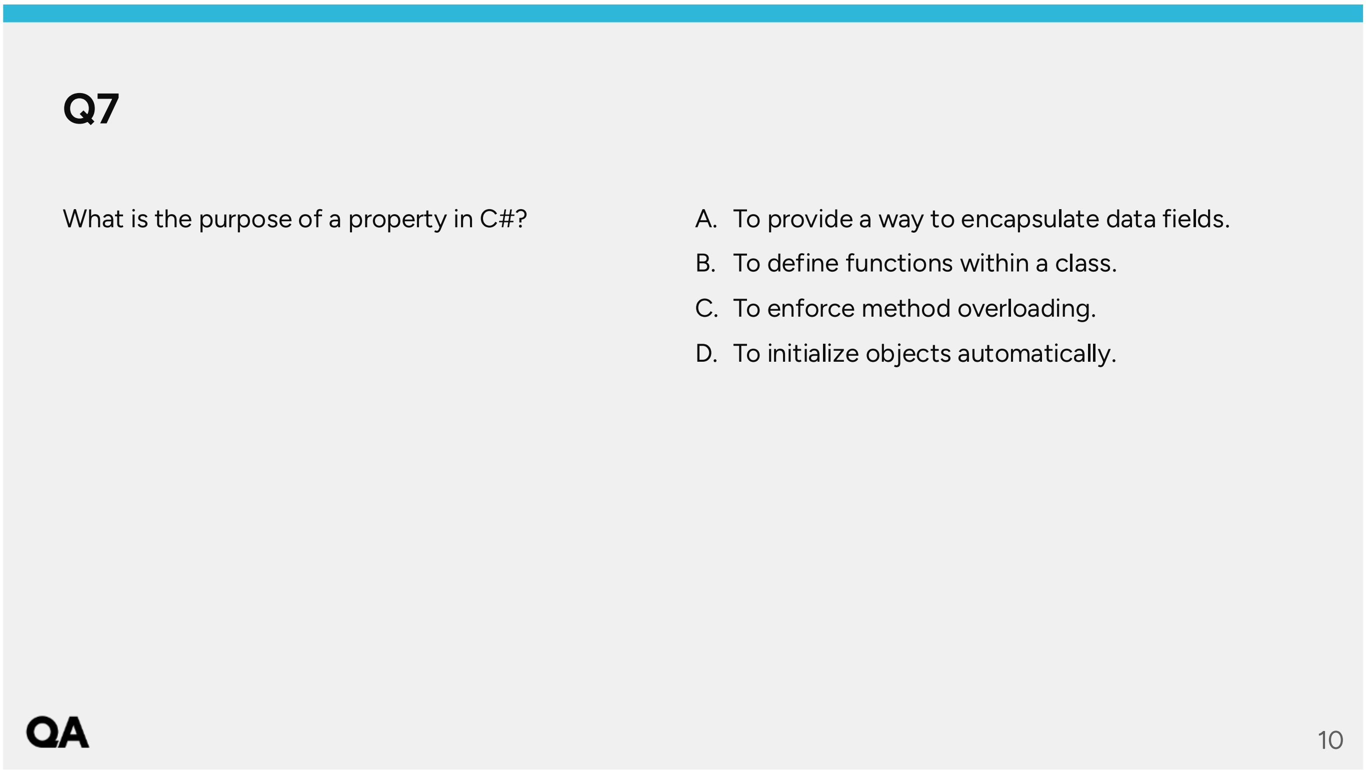
A. Constructors can have a return type.

B. A class can only have one constructor.

C. Constructors can be overloaded.

D. Constructors cannot be parameterized.

9



Q7

What is the purpose of a property in C#?

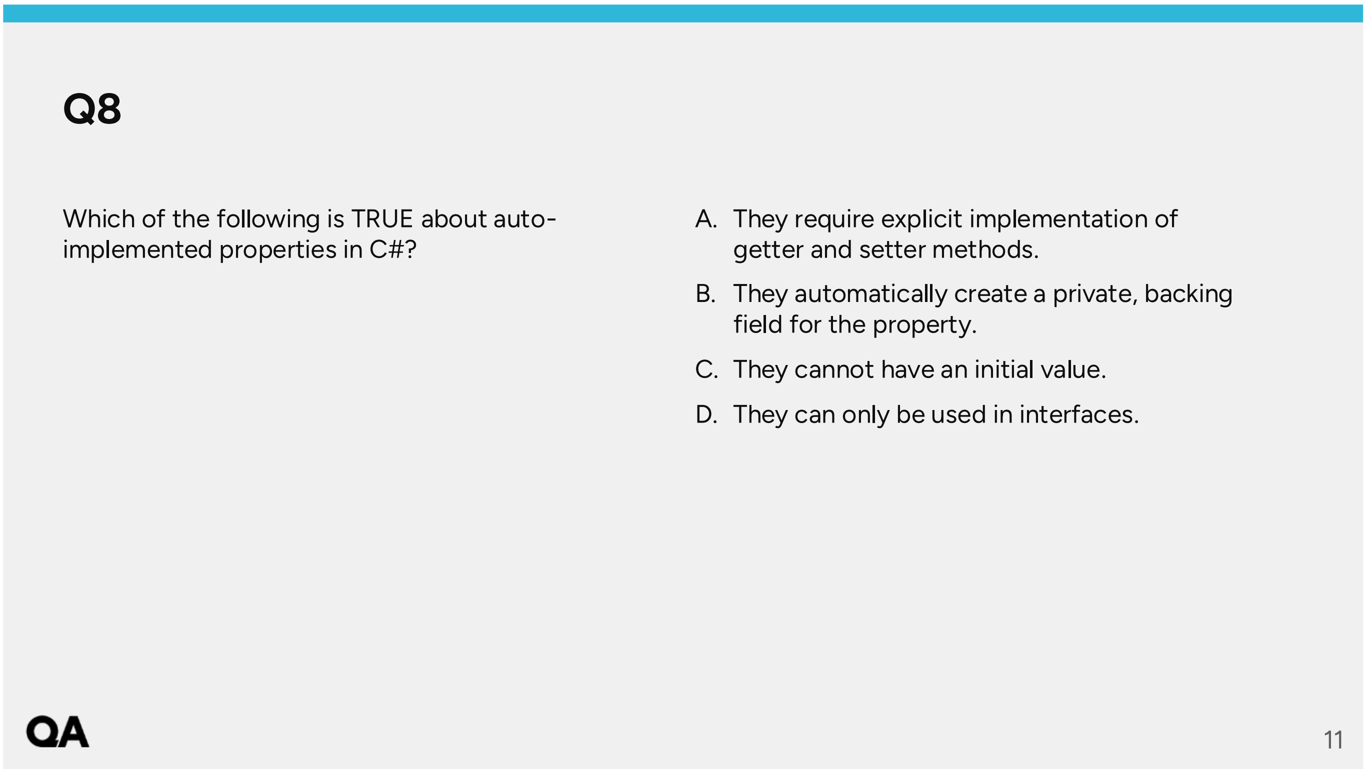
A. To provide a way to encapsulate data fields.

B. To define functions within a class.

C. To enforce method overloading.

D. To initialize objects automatically.

10



Q8

Which of the following is TRUE about auto-implemented properties in C#?

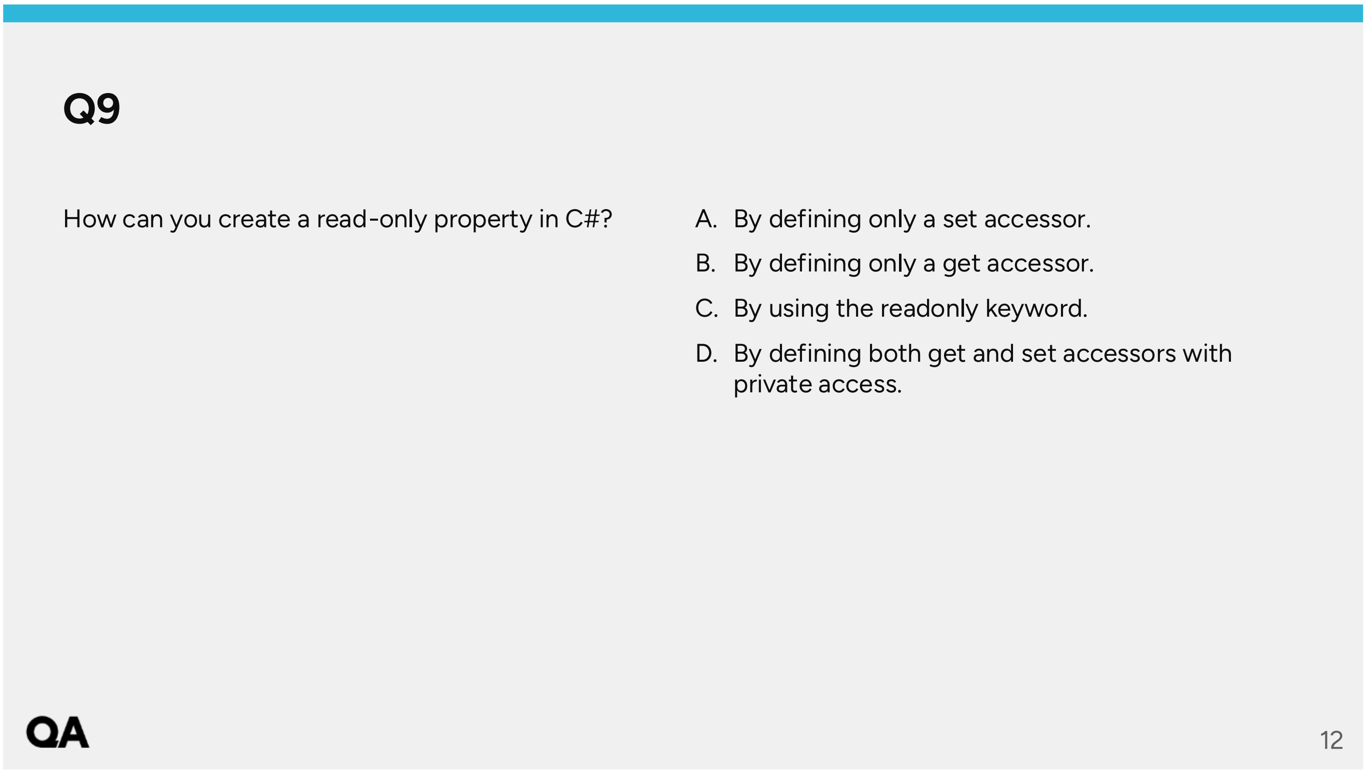
A. They require explicit implementation of getter and setter methods.

B. They automatically create a private, backing field for the property.

C. They cannot have an initial value.

D. They can only be used in interfaces.

11



Q9

How can you create a read-only property in C#?

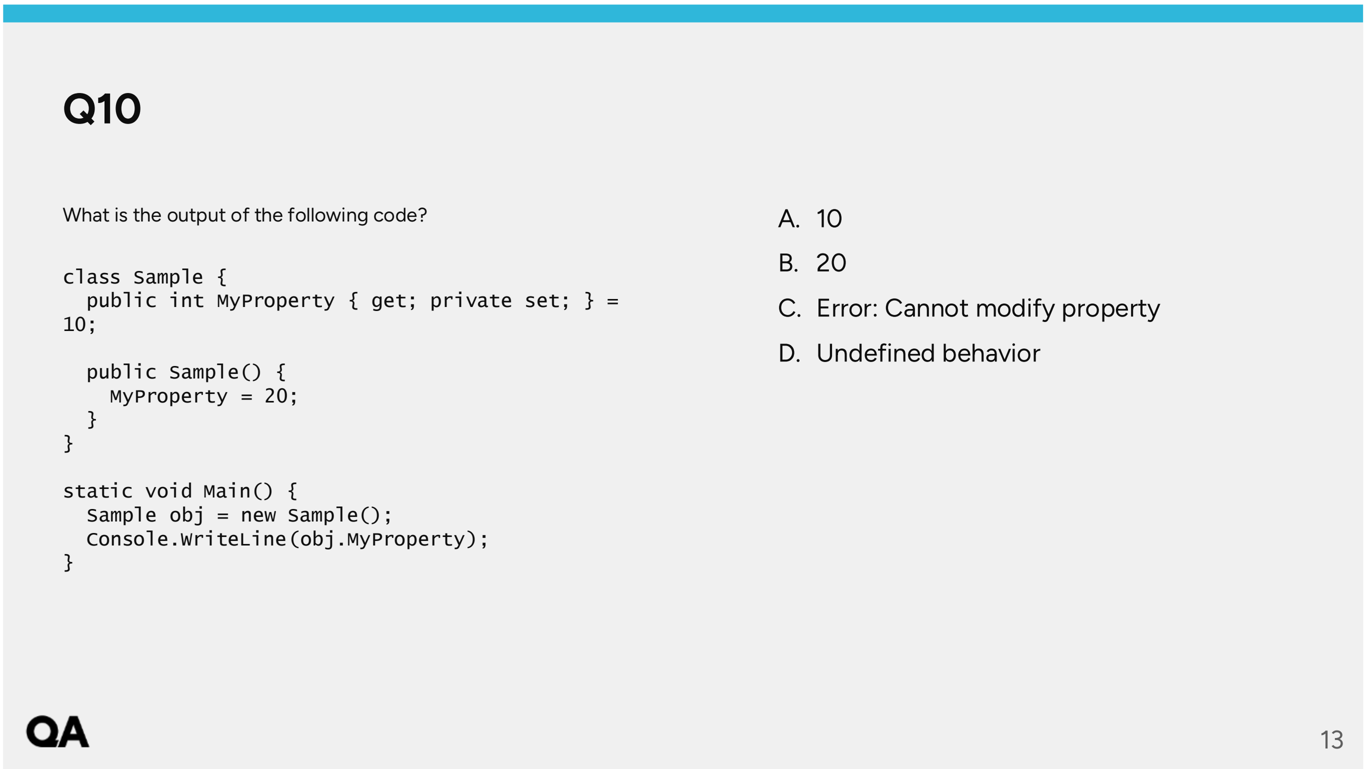
A. By defining only a set accessor.

B. By defining only a get accessor.

C. By using the readonly keyword.

D. By defining both get and set accessors with private access.

12



Q10

What is the output of the following code?

```
class Sample {
  public int MyProperty { get; private set; } =
10;

  public Sample() {
    MyProperty = 20;
  }
}

static void Main() {
  Sample obj = new Sample();
  Console.WriteLine(obj.MyProperty);
}
```

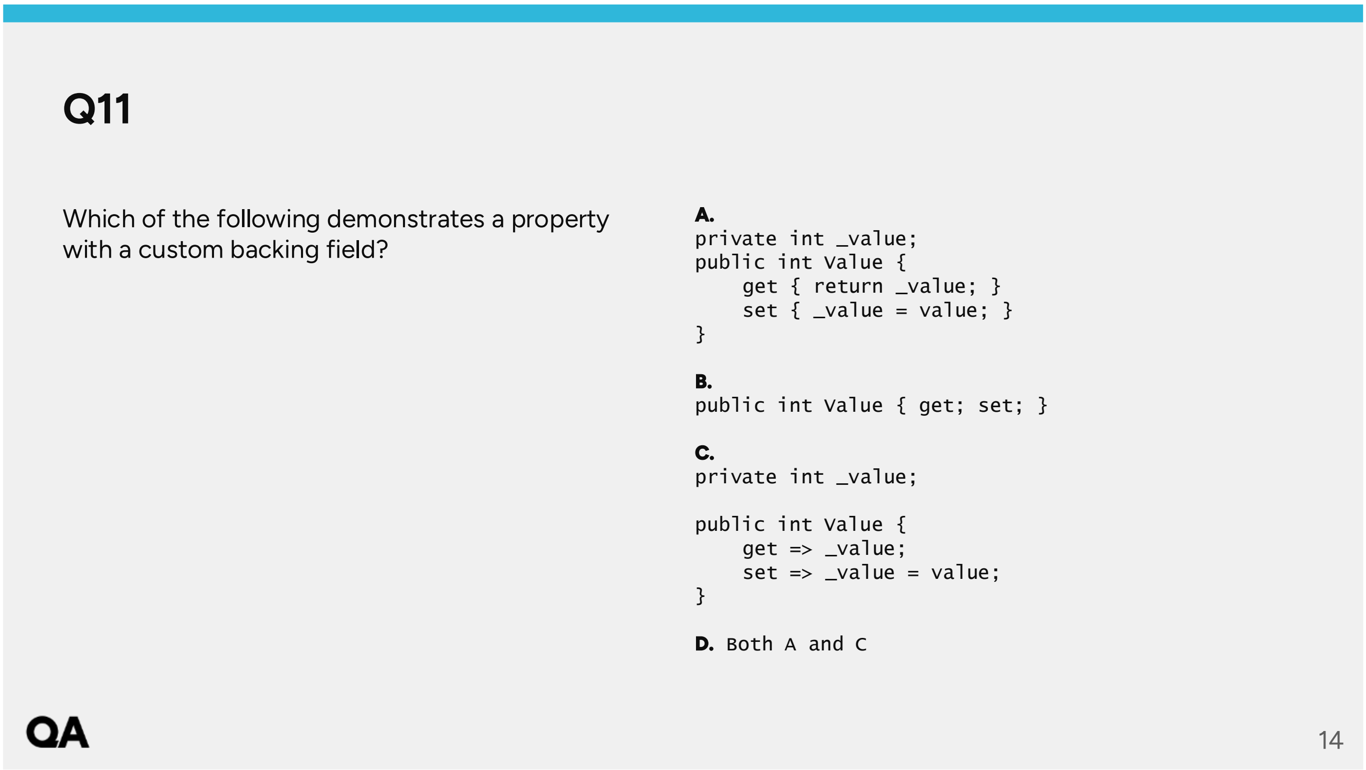
A. 10

B. 20

C. Error: Cannot modify property

D. Undefined behavior

13



Q11

Which of the following demonstrates a property with a custom backing field?

A.

```
private int _value;
public int Value {
    get { return _value; }
    set { _value = value; }
}
```

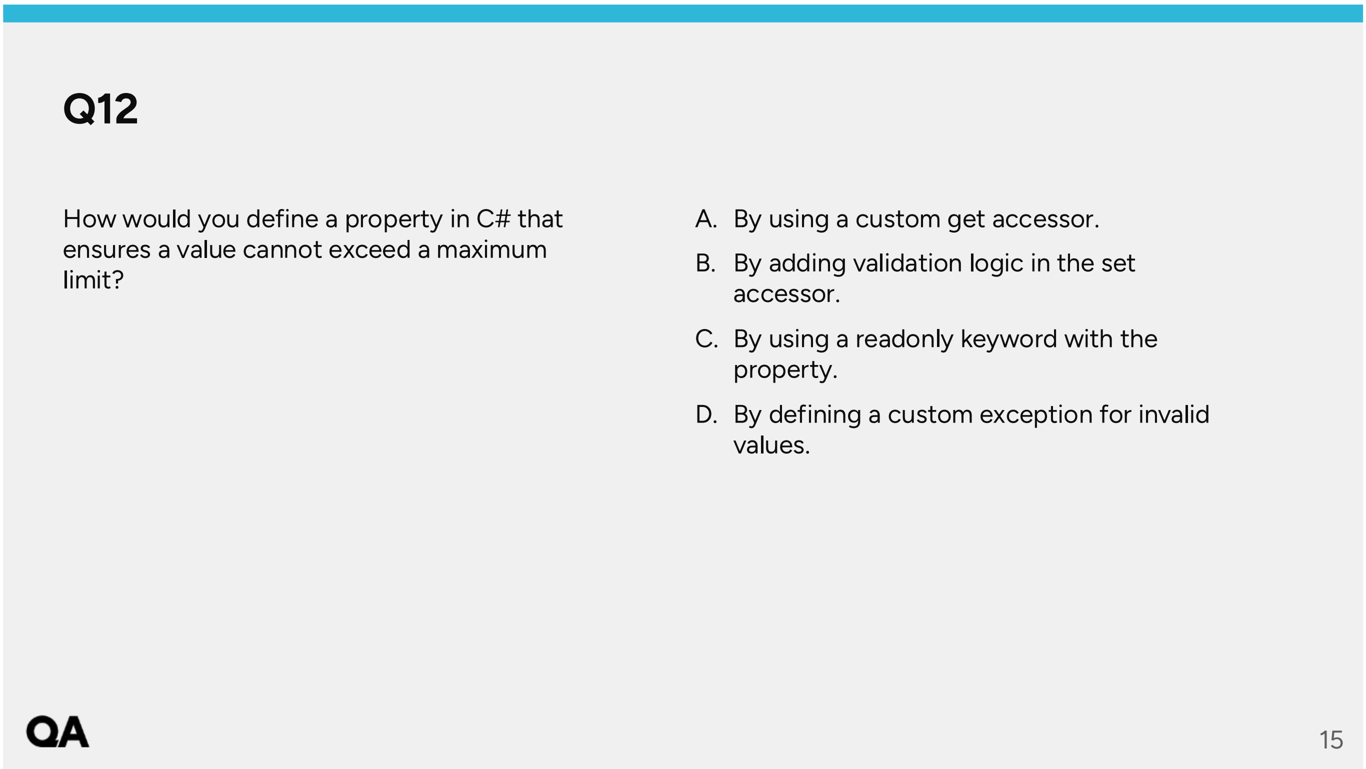
B.

```
public int Value { get; set; }
```

C.

```
private int _value;

public int Value {
    get => _value;
    set => _value = value;
}
```

D. `Both A and C`



Q12

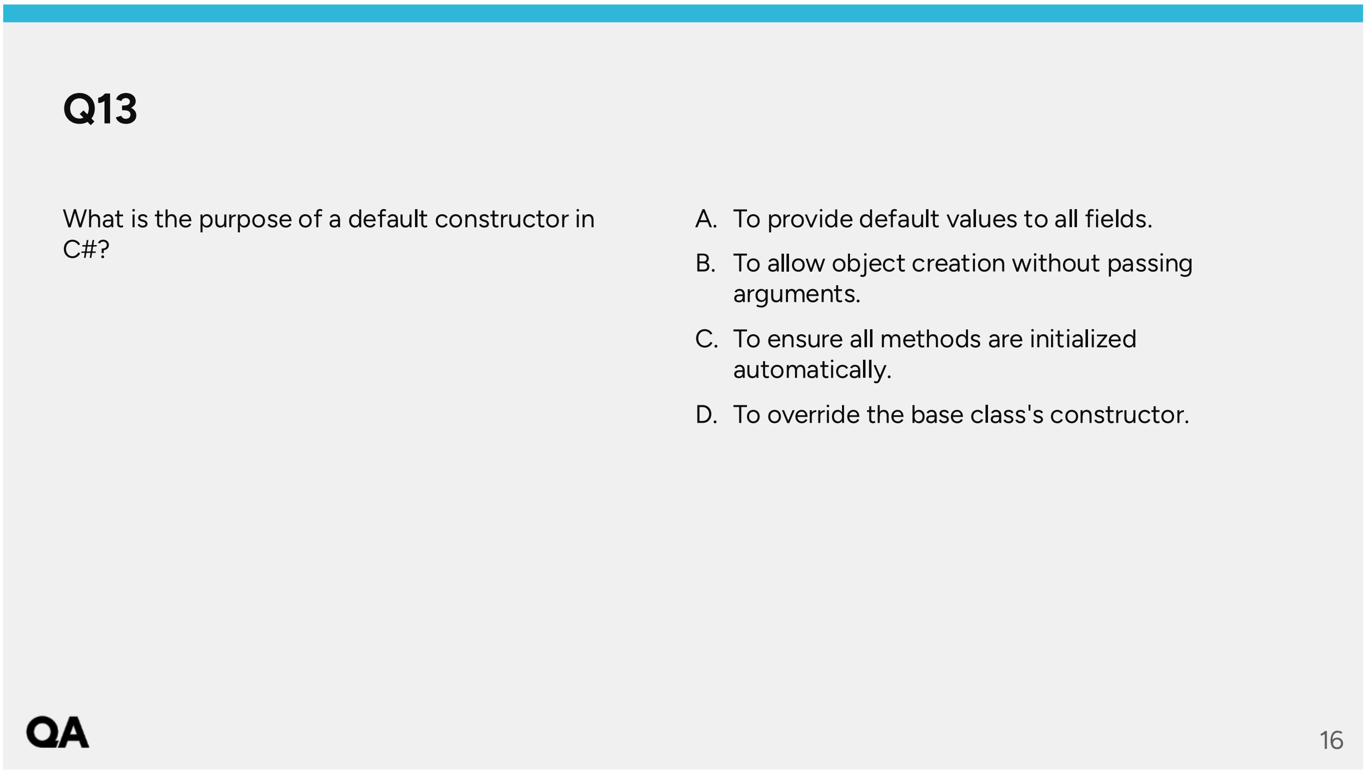
How would you define a property in C# that ensures a value cannot exceed a maximum limit?

A. By using a custom get accessor.

B. By adding validation logic in the set accessor.

C. By using a readonly keyword with the property.

D. By defining a custom exception for invalid values.



Q13

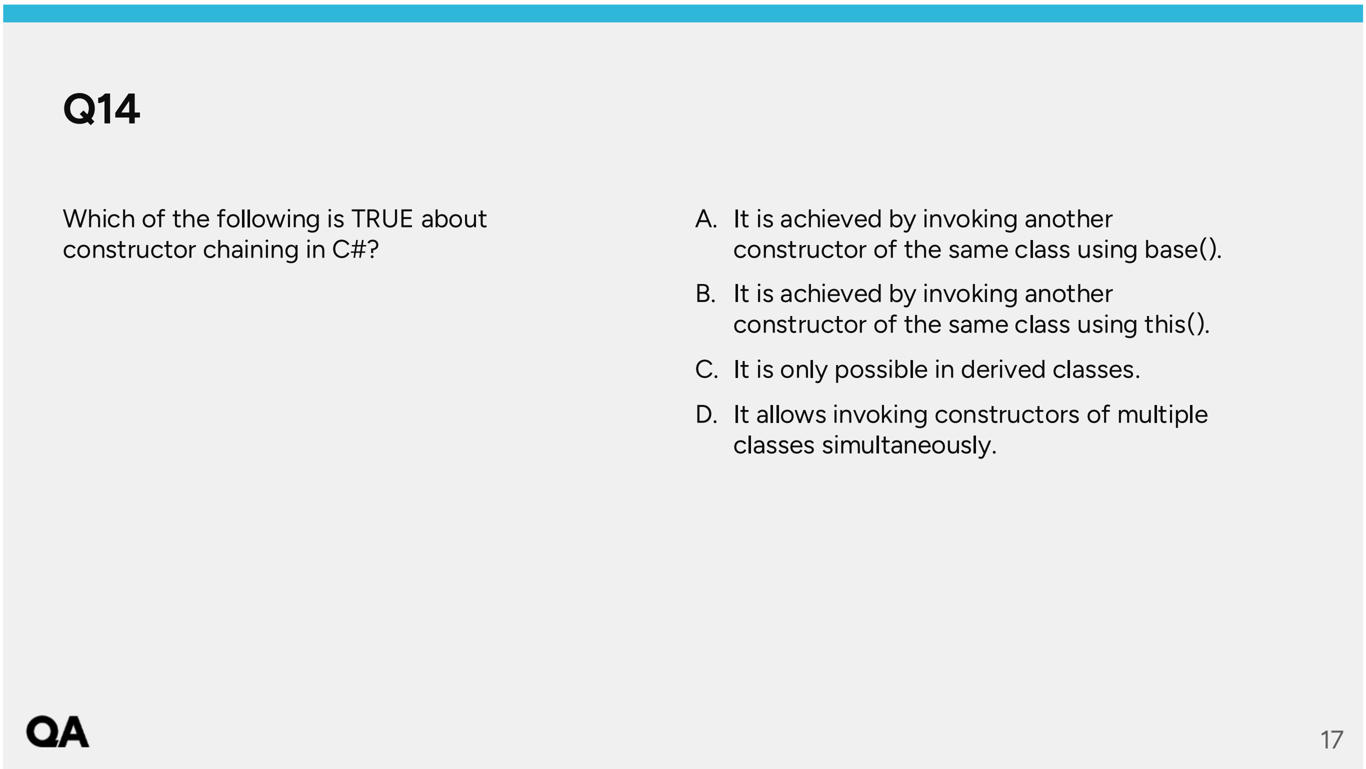
What is the purpose of a default constructor in C#?

A. To provide default values to all fields.

B. To allow object creation without passing arguments.

C. To ensure all methods are initialized automatically.

D. To override the base class's constructor.



Q14

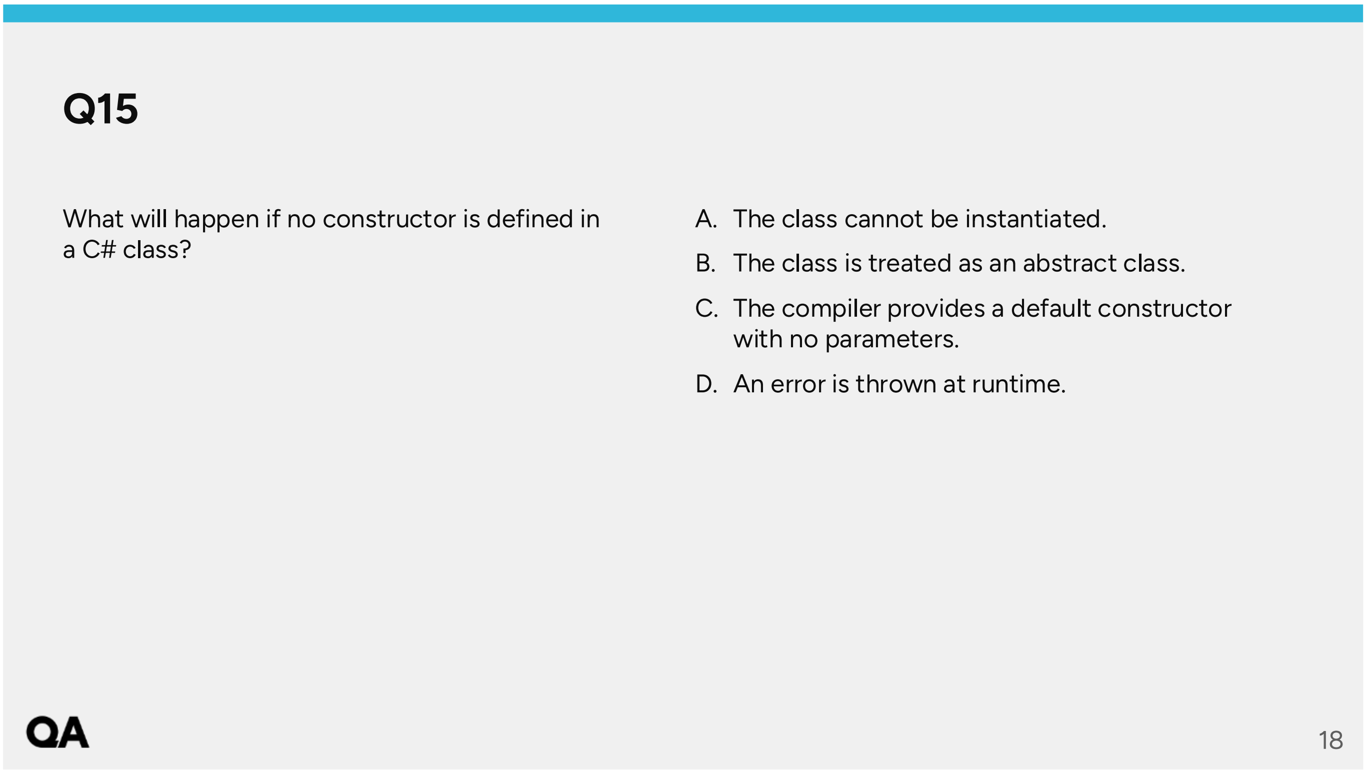
Which of the following is TRUE about constructor chaining in C#?

A. It is achieved by invoking another constructor of the same class using base().

B. It is achieved by invoking another constructor of the same class using this().

C. It is only possible in derived classes.

D. It allows invoking constructors of multiple classes simultaneously.



Q15

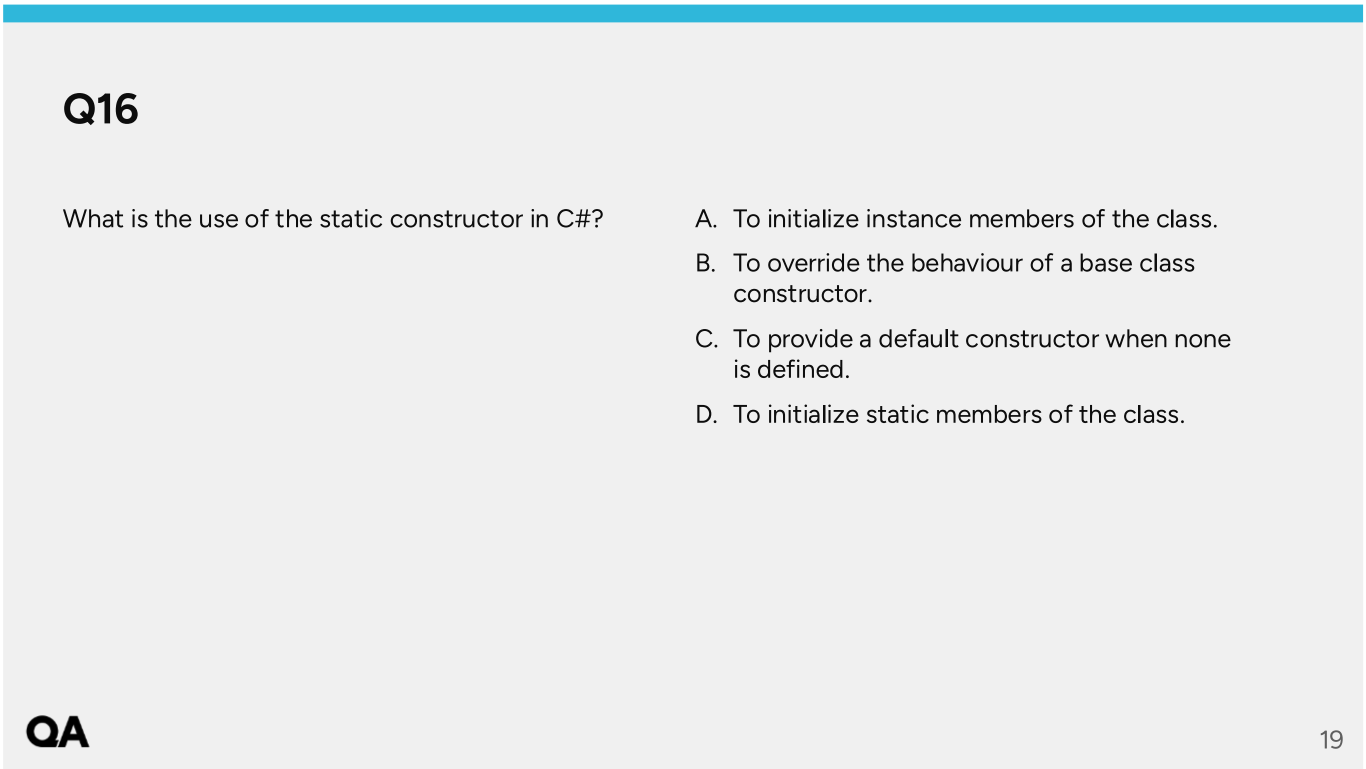
What will happen if no constructor is defined in a C# class?

A. The class cannot be instantiated.

B. The class is treated as an abstract class.

C. The compiler provides a default constructor with no parameters.

D. An error is thrown at runtime.



Q16

What is the use of the static constructor in C#?

A. To initialize instance members of the class.

B. To override the behaviour of a base class constructor.

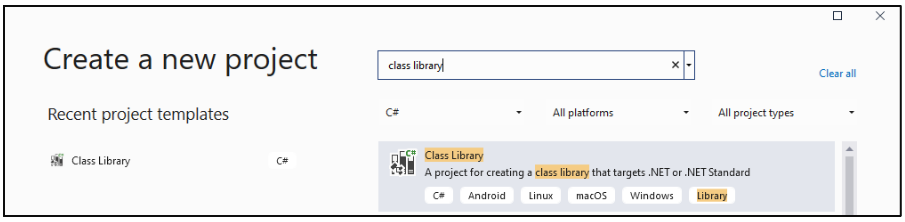
C. To provide a default constructor when none is defined.

D. To initialize static members of the class.

Inheritance

The objective of this exercise is to consolidate your understanding of inheritance by building an inheritance hierarchy.

1 Locate the **'Labs\03_Inheritance_and_Abstract_Classes\Begin'** folder and, inside it, create a new Visual Studio project of type Class Library called **LendingLibrary**.



Create a new project

class library

Recent project templates

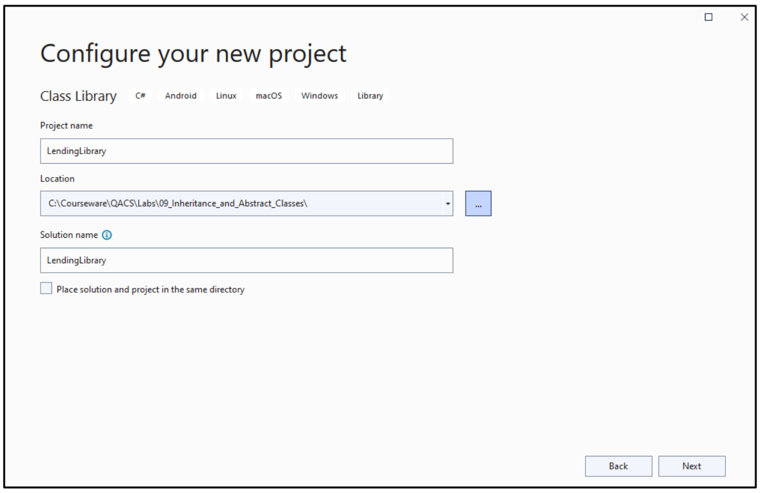
Class Library C#

C# · All platforms · All project types

Class Library

A project for creating a class library that targets .NET or .NET Standard

C# Android Linux macOS Windows Library



Configure your new project

Class Library C# Android Linux macOS Windows Library

Project name

LendingLibrary

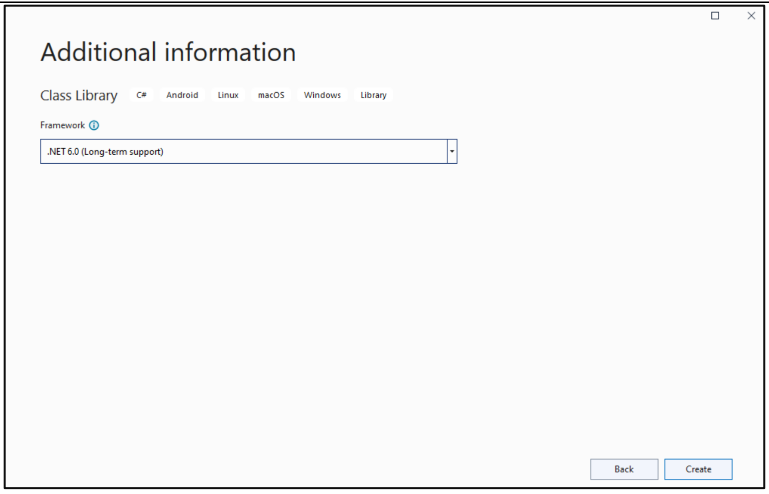
Location

C:\Courseware\QACS\Labs\09_Inheritance_and_Abstract_Classes\

Solution name

LendingLibrary

□ Place solution and project in the same directory



Additional information

Class Library C# Android Linux macOS Windows Library

Framework: .NET 6.0 (Long-term support)

Delete **Class1**

Add to this solution an XUnit Test project called **TestProject**

2 In the Test project, replace the starter test with the ‘Create’ test as found in the **‘Labs\03_Inheritance_and_Abstract_Classes\Assets’** folder (copy just the first part of the file, up to the end of the test).

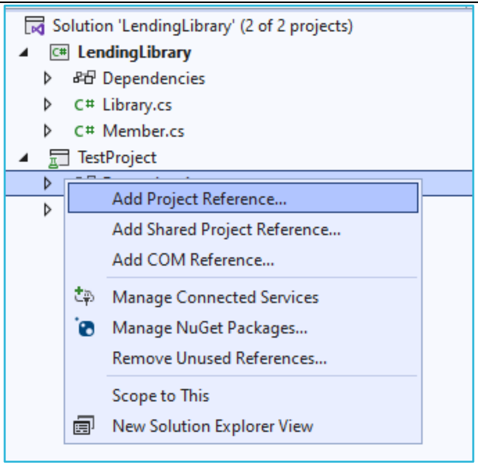
```
[Fact]
public void Create()
{
  Library library = new Library();
  Member ittzy = library.Add(name: "Ittzy Child", age: 15);
  Member irma = irma = library.Add(name: "Irma Adult", age: 73);

  Assert.Equal(2, library.NumberOfMembers);
  Assert.Equal(1, ittzy.MembershipNumber);
  Assert.Equal(2, irma.MembershipNumber);
}
```

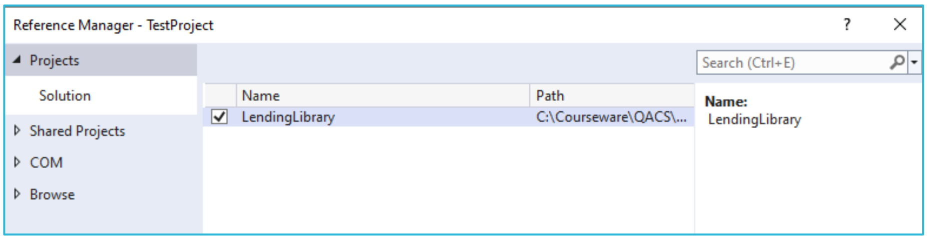
3 Create **Library** and **Member** classes in the **LendingLibrary** project, then copy in the code listed just after the ‘Create’ test.

```
public class Library
{
  Dictionary<int, Member> members = new Dictionary<int, Member>();
  public int NumberOfMembers => members.Keys.Count;

  int GetNextFreeMembershipNumber()
  {
    return (members.Keys.Count == 0) ? 1 : members.Keys.Max() + 1;
  }

  public Member Add(string name, int age)
  {
    Member member =
        new Member(name, age, GetNextFreeMembershipNumber());
    members.Add(member.MembershipNumber, member);
    return member;
  }
}
```

```
public class Member
{
  public string Name { get; }
  public int MembershipNumber { get; }
  public int Age { get; }

  public Member(string name, int age, int membershipNumber)
  {
    this.Name = name;
    this.Age = age;
    this.MembershipNumber = membershipNumber;
  }
}
```

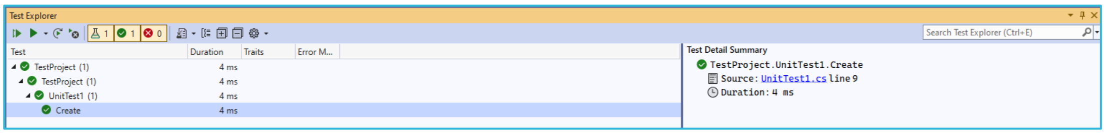
In the test project, add a project reference to the **LendingLibrary** project.


Solution 'LendingLibrary' (2 of 2 projects)
LendingLibrary
Dependencies
Library.cs
Member.cs
TestProject
Add Project Reference...
Add Shared Project Reference...
Add COM Reference...
Manage Connected Services
Manage NuGet Packages...
Remove Unused References...
Scope to This
New Solution Explorer View



Add a using statement ‘using LendingLibrary;’ to **UnitTest1.cs**

Run the **Create** test and confirm it passes.



```
TestProject.UnitTest1.Create
Source: UnitTest1.cs line 9
Duration: 4 ms
```

4 We are going to need the **library** instance and **iittzy** and **irma** member objects in further tests, so to avoid duplication of code, move the declarations of these to the class level and initialise them in the constructor.

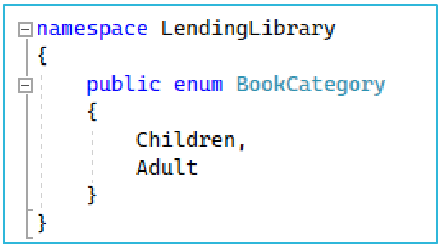
Note: If you get stuck, we’ve shown it in the Assets folder.

5 Later on, we’re going to need some additional properties of **Member**, so add these in and get Visual Studio to create the properties.

```
public UnitTest1()
{
  library = new Library();
  ittzy = library.Add(name: "Ittzy Child", age: 15);
  ittzy.Street = "1 the High Street";
  ittzy.City = "Buddlington";
  ittzy.OutstandingFines = 25M;
  irma = irma = library.Add(name: "irma Adult", age: 73);
  irma.Street = "2 the High Street";
  irma.City = "Cruddlington";
  irma.OutstandingFines = 2500M;
}
```

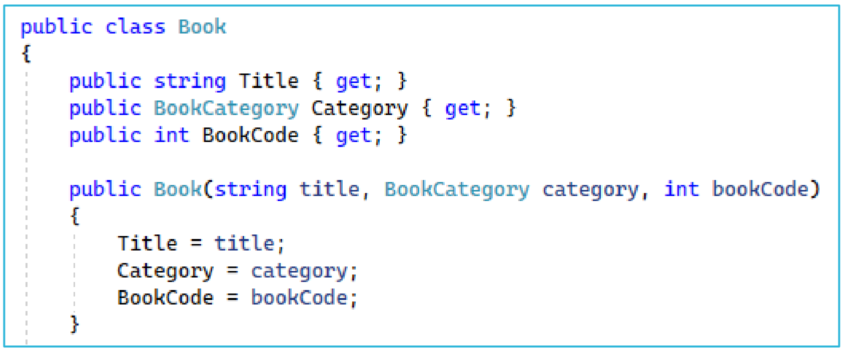
6 In this library we have different rules for borrowing a book, dependent on whether the member is over 16 or not.

We are going to need this enum. Add it to your **LendingLibrary**, in a file called **BookCategory.cs**



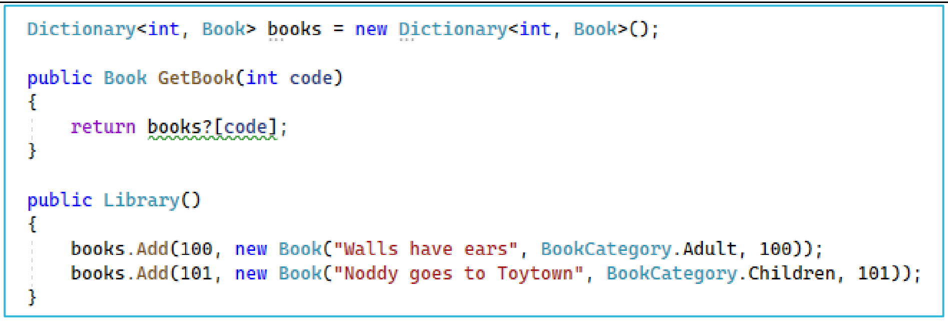
```
namespace LendingLibrary
{
    public enum BookCategory
    {
        Children,
        Adult
    }
}
```

We will need a **Book** class. Add the class and the properties and get Visual Studio to generate the constructor:



```
public class Book
{
    public string Title { get; }
    public BookCategory Category { get; }
    public int BookCode { get; }

    public Book(string title, BookCategory category, int bookCode)
    {
        Title = title;
        Category = category;
        BookCode = bookCode;
    }
```

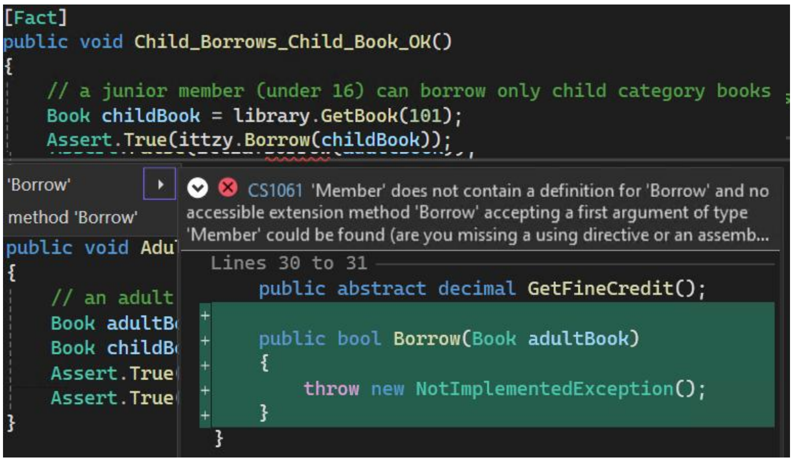
We will also need some Book code in the **Library** class:



```
Dictionary<int, Book> books = new Dictionary<int, Book>();

public Book GetBook(int code)
{
    return books?[code];
}

public Library()
{
    books.Add(100, new Book("Walls have ears", BookCategory.Adult, 100));
    books.Add(101, new Book("Noddy goes to Toytown", BookCategory.Children, 101));
}
```

7 In the TestProject, add in the **Child_Borrows_Child_Book_OK** test from the **Assets** folder.

Get Visual Studio to resolve the **Borrow()** method:



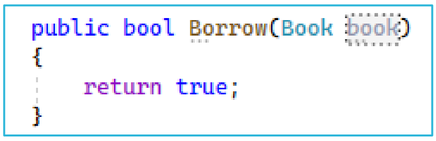
```
[Fact]
public void Child_Borrows_Child_Book_OK()
{
    // a junior member (under 16) can borrow only child category books
    Book childBook = library.GetBook(101);
    Assert.True(ittzy.Borrow(childBook));
```

```
public void Adu
{
    // an adult
    Book adultB
    Book childB
    Assert.True
    Assert.True
}
```

CS1061 'Member' does not contain a definition for 'Borrow' and no accessible extension method 'Borrow' accepting a first argument of type 'Member' could be found (are you missing a using directive or an assemb...

```
Lines 30 to 31
    public abstract decimal GetFineCredit();

+   public bool Borrow(Book adultBook)
+   {
+       throw new NotImplementedException();
+   }
}
```

Populate the method like this:



```
public bool Borrow(Book book)
{
    return true;
}
```

Run the Test and confirm it passes.

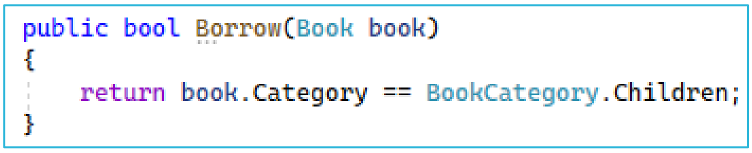
8 Add in the test **Child_Borrows_Adult_Book_Fails**.

```
[Fact]
```

```
public void Child_Borrows_Adult_Book_Fails()
{
  // a junior member (under 16) can borrow only child category
books
  Book adultBook = library.GetBook(100);
  Assert.False(ittzy.Borrow(adultBook));
}
```

Run this Test. It fails because it is currently hard-coded to return true.

Now we need to modify Borrow:



```
public bool Borrow(Book book)
{
    return book.Category == BookCategory.Children;
}
```

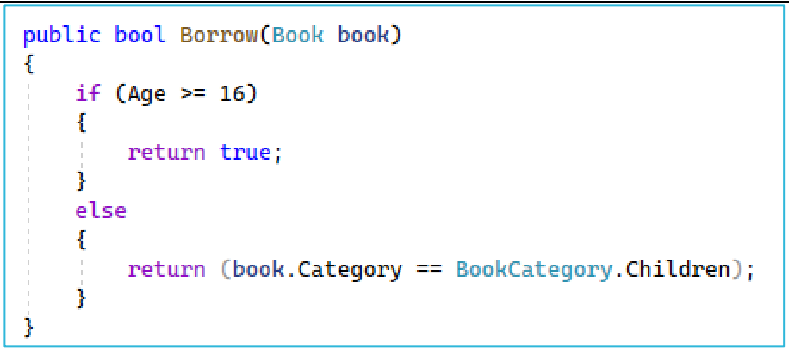
Re-run the test. It should now pass.

9 Add the test **Adult_Can_Borrow_Any_Book.**

```
[Fact]
public void Adult_Can_Borrow_Any_Book()
{
  // an adult member (over 16) can borrow any book
  Book adultBook = library.GetBook(100);
  Book childBook = library.GetBook(101);
  Assert.True(irma.Borrow(adultBook));
  Assert.True(irma.Borrow(childBook));
}
```

Run this Test. It fails because the **Borrow**() method only returns true for children's books.

Modify the **Borrow** code again:



```
public bool Borrow(Book book)
{
    if (Age >= 16)
    {
        return true;
    }
    else
    {
        return (book.Category == BookCategory.Children);
    }
}
```

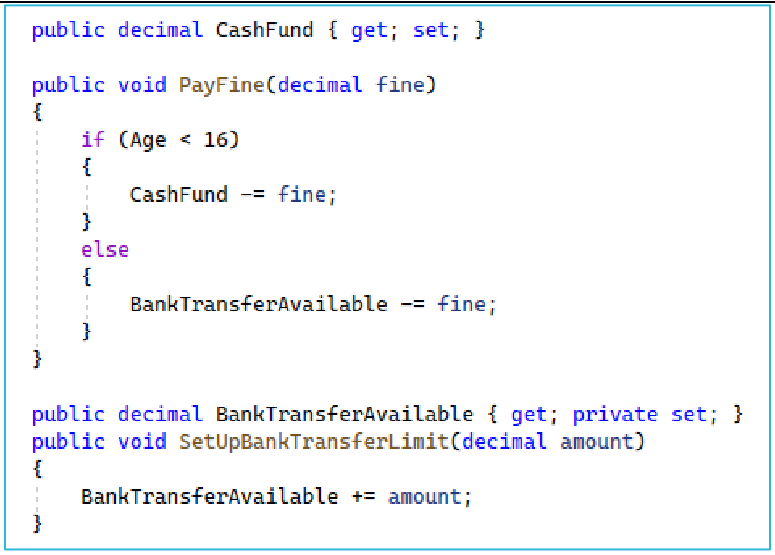
Run the Tests. All tests should now pass.

10 In this library, fines are handled differently for juniors and adults. Juniors must provide a CashFund; Adults must provide BankTransfer details.

Add in the two ‘Fines’ tests:

```
[Fact]
public void Child_Pays_Fine_From_Cash_Fund()
{
  ittzy.SetFineLimit(20M);
  ittzy.PayFine(7M);
  Assert.Equal(13M, ittzy.GetFineCredit());
}

[Fact]
public void Adult_Pays_Fine_By_Bank_Transfer()
{
  irma.SetFineLimit(20M);
  irma.PayFine(7M);
  Assert.Equal(13M, irma.GetFineCredit());
}
```

Add this to your **Member** class:



```
public decimal CashFund { get; set; }

public void PayFine(decimal fine)
{
    if (Age < 16)
    {
        CashFund -= fine;
    }
    else
    {
        BankTransferAvailable -= fine;
    }
}

public decimal BankTransferAvailable { get; private set; }
public void SetUpBankTransferLimit(decimal amount)
{
    BankTransferAvailable += amount;
}
```

Confirm all tests pass.

Where we are so far

OK – it works.

But can you see how we are constantly changing working code as we discover more about the junior and adult rules that apply to this library.

In a real project, this means we are *constantly* breaking working code.

Now we will switch to **Inheritance** to see if this helps the situation.

11 We are going to refactor the **Member** class. In order to compare versions, we will do this:

1) Copy+Paste **Member.cs**
2) Rename the original to **Member1.cs**. When it asks you if you want Visual Studio to perform a rename, answer **No**.
3) Rename the copy to **Member2.cs**.
4) In Member1.cs, press **Ctrl+A Ctrl+K Ctrl+C** to comment out the entire class.
5) Create two folders in LendingLibrary:
    a. 01 Without Inheritance
    b. 02 With Inheritance

<table>
<tr><td></td><td>

6) And move Member1.cs to **01 Without Inheritance** and Member2.cs to **02 With Inheritance**.

Your project now only has one uncommented Member class in Member2.cs.

All tests will pass as no code has been changed.

</td></tr>
<tr><td>12</td><td>

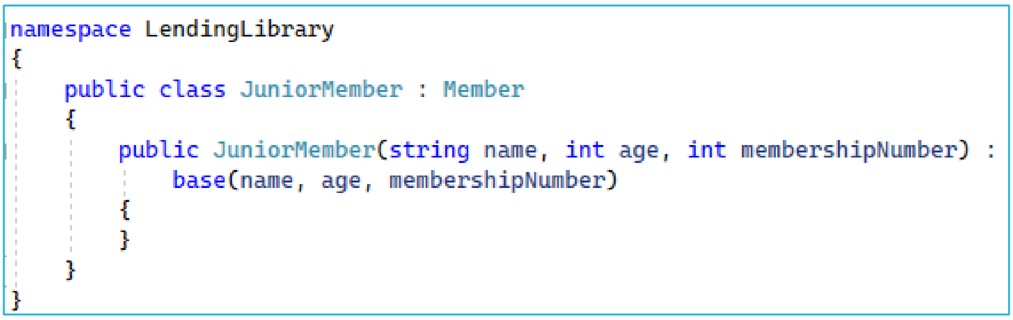
In the 02 With Inheritance folder, add two new classes: **JuniorMember** and **AdultMember**.

Adjust their namespaces to be just **LendingLibrary**.

</td></tr>
<tr><td>13</td><td>

Get both of these subclasses to derive from the **Member** base class.

Implement a constructor that passes all parameters to the base class constructor:



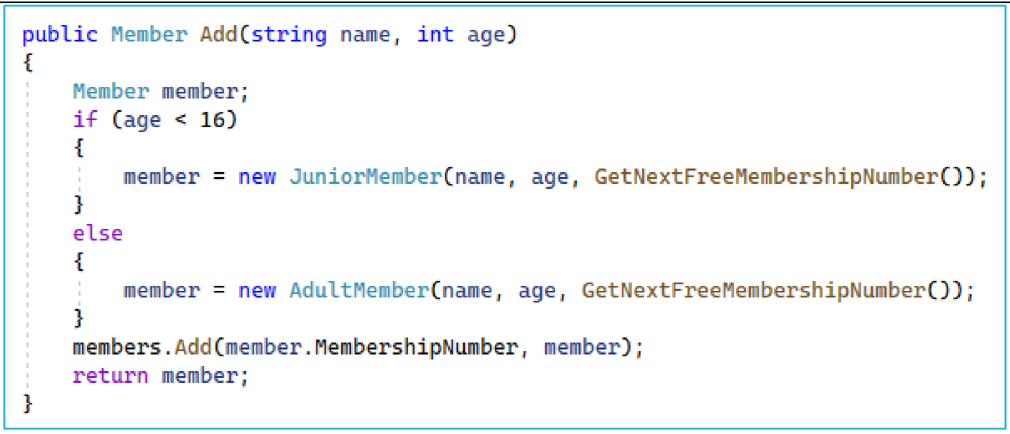
```
namespace LendingLibrary
{
    public class JuniorMember : Member
    {
        public JuniorMember(string name, int age, int membershipNumber) :
            base(name, age, membershipNumber)
        {
        }
    }
}
```

Do the same for **AdultMember**.

</td></tr>
<tr><td>14</td><td>

We need to modify **Library.Add** to create either a *Junior* or an *Adult* member. Change Library.Add() to this:

</td></tr>
</table>



```
public Member Add(string name, int age)
{
    Member member;
    if (age < 16)
    {
        member = new JuniorMember(name, age, GetNextFreeMembershipNumber());
    }
    else
    {
        member = new AdultMember(name, age, GetNextFreeMembershipNumber());
    }
    members.Add(member.MembershipNumber, member);
    return member;
}
```

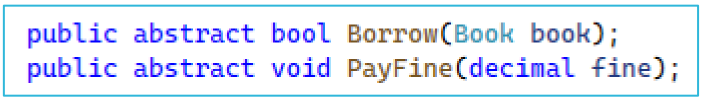
All tests should pass.

15 Actually, we want to disallow creating ‘Member’ objects – clients should be forced to create either Junior or Adult members.

Make the Member class **abstract**.

16 In Member2.cs, copy Borrow(), CashFund, PayFine(), BankTransferAvailable and SetUpBankTransferLimit() into notepad, deleting them from Member.

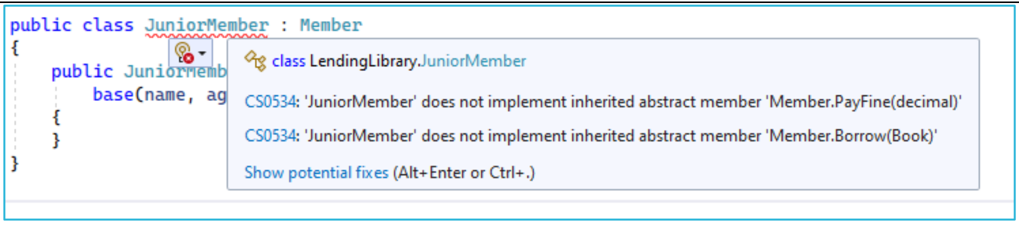
17 In Member, insert these two abstract methods:



```
public abstract bool Borrow(Book book);
public abstract void PayFine(decimal fine);
```

These abstract methods replace the concrete versions we just deleted.

18 Go to **JuniorMember**. You will see a red squiggly.



```
public class JuniorMember : Member
{
    public JuniorMemb
        base(name, ag
    {
    }
}
```

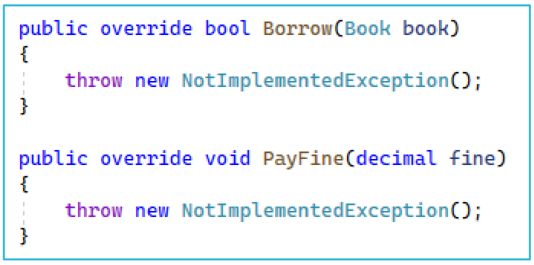
class LendingLibrary.JuniorMember

CS0534: 'JuniorMember' does not implement inherited abstract member 'Member.PayFine(decimal)'

CS0534: 'JuniorMember' does not implement inherited abstract member 'Member.Borrow(Book)'

Show potential fixes (Alt+Enter or Ctrl+.)

Using **Ctrl+dot**, implement the Abstract class. This will put in the signatures of the methods defined in Member:



```
public override bool Borrow(Book book)
{
    throw new NotImplementedException();
}

public override void PayFine(decimal fine)
{
    throw new NotImplementedException();
}
```

19 Inside each of these members, copy the relevant code from that which you stored in notepad that is specific just to *Junior* members

Repeat for AdultMember.

- You will no longer need the 'if' statement because you are removing the code that doesn't apply
- You will need to paste in the **CashFund** property for the JuniorMember, and the **BankTransferAvailable** and associated **SetUpBankTransferLimit()** method for the AdultMember.

20 If you now go back to the tests, you can see that it doesn't know that iittzy is a JuniorMember and Irma is an AdultMamber:

```
public class UnitTest1
{
  Library library;
  Member ittzy;
  Member irma;
```

There are two ways of fixing this:

1) Make iittzy a Junior and irma an Adult.
2) Find a form of word that works for both such that the client software is unaware as to whether they are Junior or Adult.

We’ll do both…

21 Make these changes:

```
Library library;
Member Iittzy;
Member irma;

public UnitTest1()
{
  library = new Library();
  Iittzy = (JuniorMember) library.Add(
      name: "Iittzy Child", age: 15);
  Iittzy.Street = "1 the High Street";
  Iittzy.City = "Buddlington";
  Iittzy.OutstandingFines = 25M;
  irma = (AdultMember) library.Add(
      name: "irma Adult", age: 73);
  irma.Street = "2 the High Street";
  irma.City = "Cruddlington";
  irma.OutstandingFines = 2500M;
}
```

22 The tests will pass, however, this is not a great solution because the client code is now acutely aware of the subclasses, meaning if a new subclass is invented, for example, *StudentMember*, you will have to modify the client code.

23 Use Ctrl+z (Undo) to remove the changes you made in step 21.

24 A better way of looking at it is:

'Is there some form of words that could operate at the Member level (i.e., for every type of Member) that makes sense to both Junior and Adult members and could be interpreted correctly by both of them?'

i.e., the same intent but they have different implementations?

How about **SetFineLimit**() and **GetFineCredit**() ?

25 Make these changes:

TestProject

```
 [Fact]
public void Child_Pays_Fine_From_Cash_Fund()
{
  Iittzy.SetFineLimit(20M);
  Iittzy.PayFine(7M);
  Assert.Equal(13M, Iittzy.GetFineCredit());
}

[Fact]
public void Adult_Pays_Fine_By_Bank_Transfer()
{
  irma.SetFineLimit(20M);
  irma.PayFine(7M);
  Assert.Equal(13M, irma.GetFineCredit());
}
```

Member:

```
public abstract void SetFineLimit(decimal amount);
public abstract decimal GetFineCredit();
```

When adding methods to JuniorMember and AdultMember, you should use **ctrl-dot** on the red squiggly to create the method signatures for you, then delete the **NotImplementedException** and replace it with the relevant code for the specific class.

JuniorMember:

```
private decimal CashFund { get; set; }// now private

public override void SetFineLimit(decimal amount)
{
  CashFund = amount;
}

public override decimal GetFineCredit()
{
  return CashFund;
}
```

AdultMember:

```
private decimal BankTransferAvailable { get; set; } // now private

public override void SetFineLimit(decimal amount)
{
  SetUpBankTransferLimit(amount);
}

public override decimal GetFineCredit()
{
  return BankTransferAvailable;
}
```

All tests should pass.

State Pattern

If you don't have time:

Then the moral of this section is 'Always ask the question – can a subtype morph into another subtype?'. For example, can a dog become a cat, keeping the original mammally bits? If the answer is 'No', as is the case with dogs and cats, then inheritance (as we've done so far in this lab) is fine.

But often one type can morph into another. In our case, a JuniorMember can become an AdultMember when they turn 16. Therefore, the statement should be:

A LibraryMember has a MembershipType (and a Junior MembershipType is a type of MembershipType)

rather than:

A LibraryMember is a Junior Member

If you have time, then carry on:

Note: If this section is difficult to appreciate, don't worry. It's something you should, in time, be aware of.

Unfortunately, even though the solution we've just developed looks really elegant as there are almost no 'if' statements in there and all the concerns are separated, there is still a problem.

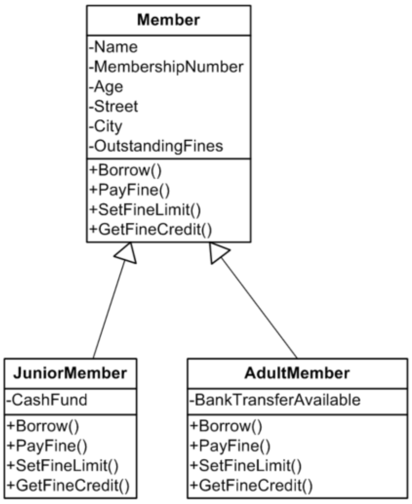
What happens when Iittzy turns 16?

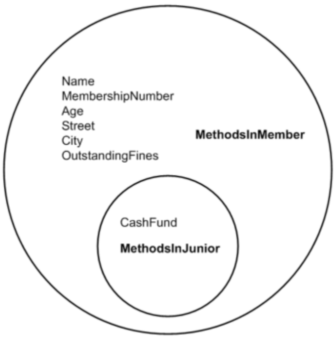
Easy, you say – you just create her as an AdultMember

The rules of inheritance are that you cannot morph one type into another, so you have to destroy the original JuniorMember and create a brand new AdultMember. This presents some problems:

1. You have to transfer all the data (name, street, city, etc.) - what if some of that data is private?
2. Iittzy will feature in various collections. She would need to be removed from these and the new Iittzy would need to be inserted silently (probably breaking the rules we have established).

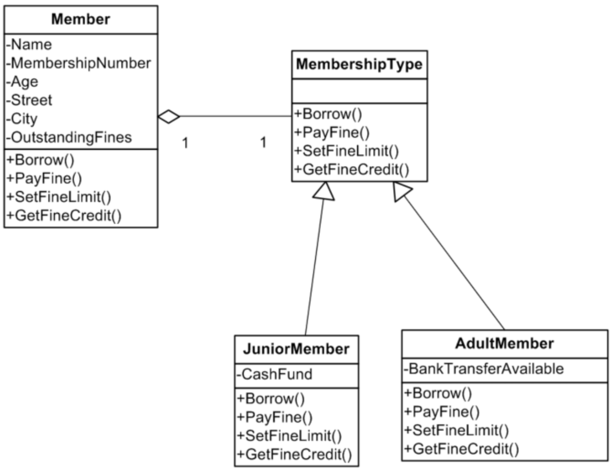
Our current class diagram and a representative object is shown below:

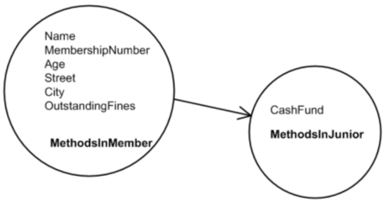




It's like we have one object built from two recipes – the Member recipe and the JuniorMember recipe. And when we destroy this object, we not only throw away the CashFund and MethodsInJunior (which is fine), we also throw away all the data and methods in the Member bit of the object.

What we need is the following. We need two objects such that we only throw away the fields and methods specific to being a Junior. For example, a Member has a MembershipType rather than a Member is a Junior from birth to death.





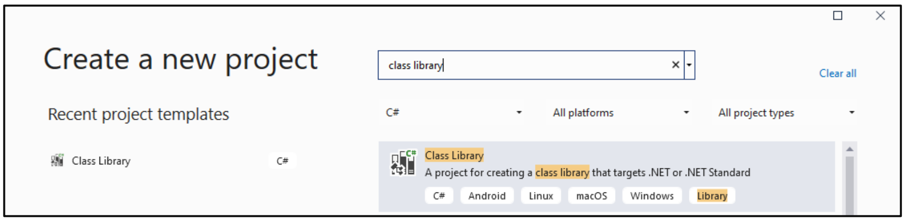
If you have plenty of time and feel very confident, then go ahead and have a go at changing your solution to be the State Pattern.

However, if it’s been a long lab already, it’s best to open the solution (**End2**), go to the **View** menu and select **Task List**. We have already made the required changes, and we have also annotated the few changes that were needed.

Interfaces

The objective of this exercise is to consolidate your understanding of interfaces.

1 Locate the **'..\Labs\04_Interfaces\Begin'** folder and, inside it, create a new Visual Studio project of type Class Library called **InterfaceProject**.



Create a new project

class library

Recent project templates

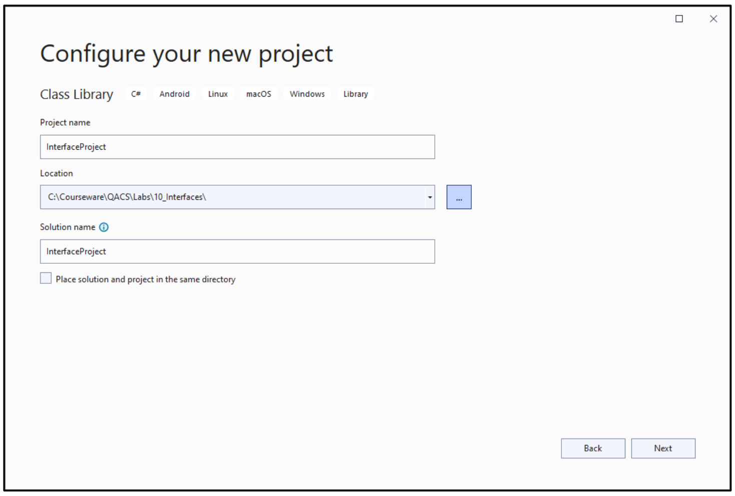
Class Library C#

C# · All platforms · All project types

Class Library

A project for creating a class library that targets .NET or .NET Standard

C# Android Linux macOS Windows Library



Configure your new project

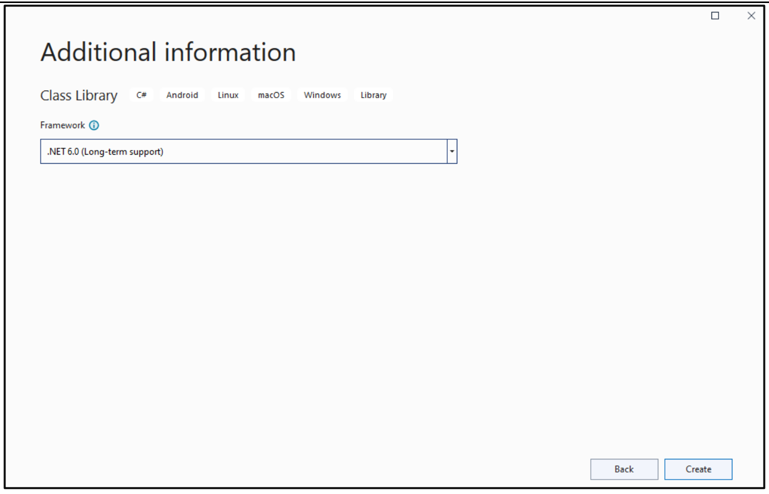
Class Library C# Android Linux macOS Windows Library

Project name: InterfaceProject

Location: C:\Courseware\QACS\Labs\10_Interfaces\

Solution name: InterfaceProject

□ Place solution and project in the same directory



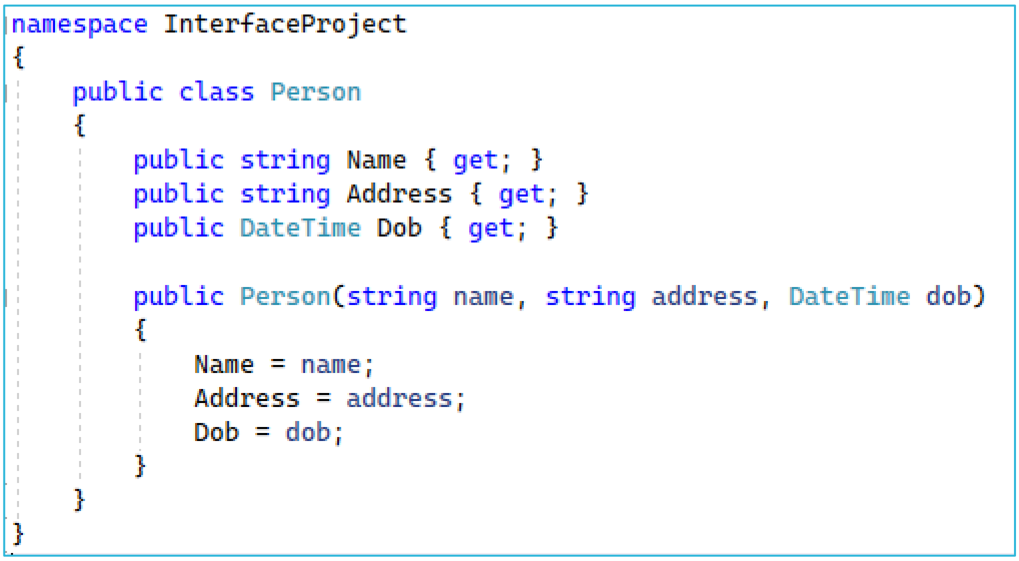
Framework

.NET 6.0 (Long-term support)

Delete **Class1**.

Add to this solution an XUnit Test project called **TestProject**.

2 We have provided a **Person** class in the Assets folder. Drag this file onto your **InterfaceProject**.



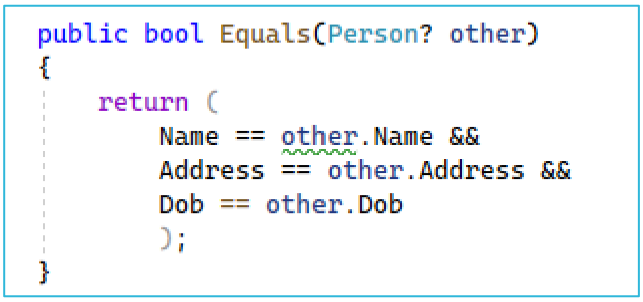
```
namespace InterfaceProject
{
    public class Person
    {
        public string Name { get; }
        public string Address { get; }
        public DateTime Dob { get; }

        public Person(string name, string address, DateTime dob)
        {
            Name = name;
            Address = address;
            Dob = dob;
        }
    }
}
```

<table>
<tr><td>3</td><td>Replace the provided Test1() with the two tests in <b>IEquatable.txt</b> from the <b>'..\Labs\04_Interfaces\Assets'</b> folder and resolve the red squiggles.</td></tr>
<tr><td>4</td><td>Run the tests. The first test fails, the second test passes by accident.

The problem is that the compiler doesn't know how to figure out if two Persons are the same person. For example, if your bank account has a balance of £100 and so does mine – does that mean they are the same account?</td></tr>
<tr><td>5</td><td>Make Person implement the interface <b>IEquatable<Person&gt;</b>

This will make you implement the <b>Equals</b> method.

As far as we are concerned, if the Name, Address, and DateOfBirth are the same, it is the same person. Put in this code and ensure both tests now pass.

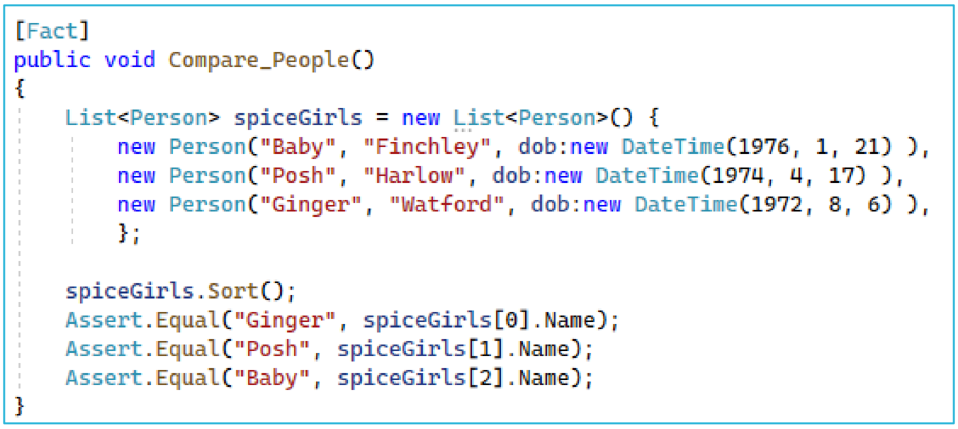


```
public bool Equals(Person? other)
{
    return (
        Name == other.Name &&
        Address == other.Address &&
        Dob == other.Dob
        );
}
```

</td></tr>
<tr><td>6</td><td>You will now compare some people, the Spice Girls, by putting them in order based on their DateOfBirth.

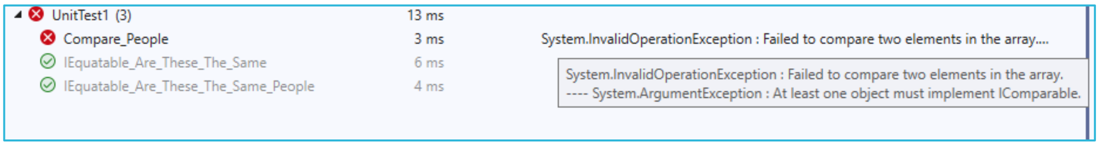
We'll put these in DateOfBirth order.</td></tr>
</table>

Copy in the test in the Assets folder called **Compare_People** and resolve the red squiggles.



```
[Fact]
public void Compare_People()
{
    List<Person> spiceGirls = new List<Person>() {
        new Person("Baby", "Finchley", dob:new DateTime(1976, 1, 21) ),
        new Person("Posh", "Harlow", dob:new DateTime(1974, 4, 17) ),
        new Person("Ginger", "Watford", dob:new DateTime(1972, 8, 6) ),
        };

    spiceGirls.Sort();
    Assert.Equal("Ginger", spiceGirls[0].Name);
    Assert.Equal("Posh", spiceGirls[1].Name);
    Assert.Equal("Baby", spiceGirls[2].Name);
}
```

Run the test. It will fail. Look at the error message:



```
◢ ☒ UnitTest1 (3)                          13 ms
    ☒ Compare_People                        3 ms    System.InvalidOperationException : Failed to compare two elements in the array....
    ✓ IEquatable_Are_These_The_Same         6 ms      System.InvalidOperationException : Failed to compare two elements in the array.
    ✓ IEquatable_Are_These_The_Same_People  4 ms      ---- System.ArgumentException : At least one object must implement IComparable.
```

It fails because it does not know how to sort Persons into order because they are not comparable.

7 Implement the interface **IComparable<Person>**

Notice the **CompareTo** method returns an int.

We want to rank Person by **DateOfBirth**, so:

CompareTo should return **+1** if DoB is greater than the compared to object's' DoB.

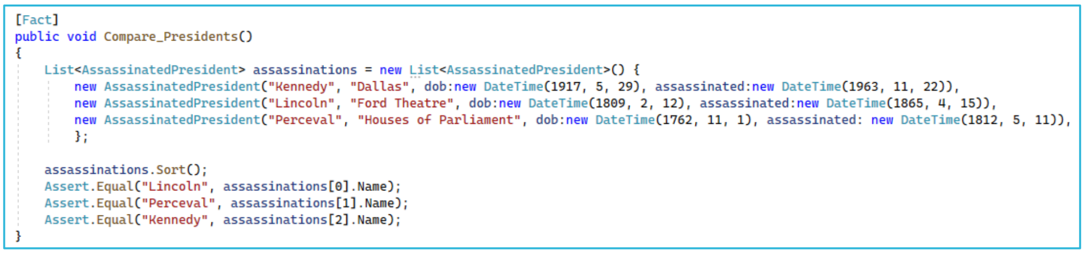
-1 if smaller

Otherwise return **0**.

Enter the code and run the test to ensure it now passes.

8 In the Assets folder is a class **AssassinatedPresident**. Drag this onto your **InterfaceProject**.

Add the test **Compare_Presidents**.



```
[Fact]
public void Compare_Presidents()
{
    List<AssassinatedPresident> assassinations = new List<AssassinatedPresident>() {
        new AssassinatedPresident("Kennedy", "Dallas", dob:new DateTime(1917, 5, 29), assassinated:new DateTime(1963, 11, 22)),
        new AssassinatedPresident("Lincoln", "Ford Theatre", dob:new DateTime(1809, 2, 12), assassinated:new DateTime(1865, 4, 15)),
        new AssassinatedPresident("Perceval", "Houses of Parliament", dob:new DateTime(1762, 11, 1), assassinated: new DateTime(1812, 5, 11)),
        };

    assassinations.Sort();
    Assert.Equal("Lincoln", assassinations[0].Name);
    Assert.Equal("Perceval", assassinations[1].Name);
    Assert.Equal("Kennedy", assassinations[2].Name);
}
```

Now implement **IComparable** for **AssassinatedPresident** where we want the sort order to be by the *month* in which they were *assassinated*.

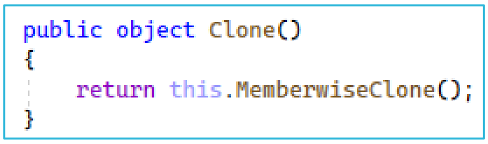
Run the Test and confirm it passes.

9 You will now experiment with cloning objects.

Copy in the **Clone_A_Spice_Girl** test.

The interface you need is **ICloneable**. Make Person implement the interface **ICloneable**.

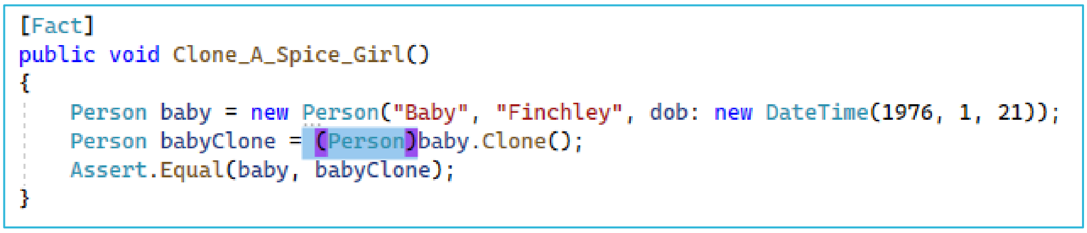
Conveniently, there is a protected method on the Object class which does this for us:



```
public object Clone()
{
    return this.MemberwiseClone();
}
```

But you'll get a red squiggly in the test.

This is because **Clone** method returns an object, so we have to cast the result to **Person** in the test:



```
[Fact]
public void Clone_A_Spice_Girl()
{
    Person baby = new Person("Baby", "Finchley", dob: new DateTime(1976, 1, 21));
    Person babyClone = (Person)baby.Clone();
    Assert.Equal(baby, babyClone);
}
```

Re-run the test. It should now pass.

10 This works, but it’s not ideal.

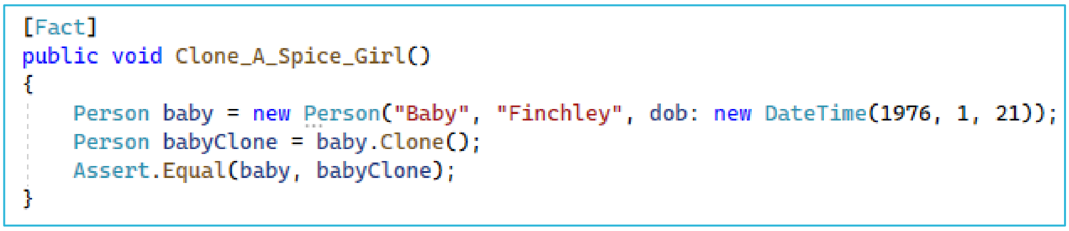
It’s a pity that the framework does not offer a generic version of **ICloneable.**

Have a go at creating your own generic **ICloneable<T>** interface.

Hint: Do an F12 on the generic interfaces **IEquatable** and **IComparable** to see how they are created.

Implement your generic version of **ICloneable<T>** in Person instead of the non-generic **ICloneable**. You can perform the explicit cast in the implemented **Clone** method.

Your test should no longer require the explicit cast and should be as follows:

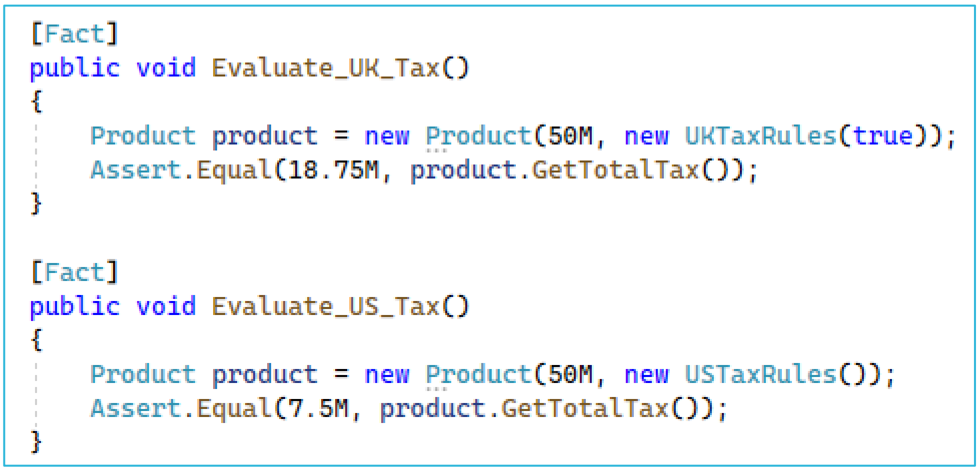


```
[Fact]
public void Clone_A_Spice_Girl()
{
    Person baby = new Person("Baby", "Finchley", dob: new DateTime(1976, 1, 21));
    Person babyClone = baby.Clone();
    Assert.Equal(baby, babyClone);
}
```

Run the Tests. All tests should now pass.

11 You will now explore another interface **ITaxRules** and see how it is implemented in different classes.

Into your **TestProject**, copy the **Tax Tests** from the **Assets** folder:



```
[Fact]
public void Evaluate_UK_Tax()
{
    Product product = new Product(50M, new UKTaxRules(true));
    Assert.Equal(18.75M, product.GetTotalTax());
}

[Fact]
public void Evaluate_US_Tax()
{
    Product product = new Product(50M, new USTaxRules());
    Assert.Equal(7.5M, product.GetTotalTax());
}
```

We have provided all the code for you.

Copy the **Product** folder from the **Assets** folder and drop it onto your **InterfaceProject**.

Have a look at the **Product** class. See that it is passed in an **ITaxRules** in its constructor, but it doesn't know or care what sort of tax rules they, are as long as they implement **ITaxRules.**

Review the code and confirm all tests pass.

If you have time: Explicit interfaces

NOTE, This part of the lab makes use of concepts discussed on slides 13 to 15 of the appendix

Because one class can implement many interfaces, it's possible that there could be a clash of method name. In the example coming up, we create an **AmphibiousVehicle** (this is the DUKW – pronounced Duck), and it was actually the US Army coding for such a vehicle.

D=1942

U=Utility

K=all-wheel drive

W=2 powered rear axles

This remarkable vehicle is truly a water vehicle and truly a land vehicle. In other technologies we might describe this by using multiple inheritance. .NET outlaws multiple inheritance, so we would achieve this by implementing multiple interfaces.

1	Drag the folder **DUKW** onto your **InterfaceProject** and have a look at the three files. In particular, notice that both interfaces have a **Brake** method. A land vehicle brakes by squeezing the disc pads. A water vehicle brakes by putting the propeller into reverse. Therefore, we need two methods.
2	Open the **AmphibiousVehicle** class and implement the interface **ILandVehicle**. Build the project to confirm there are no errors. You will see it gives you one method and both interfaces are satisfied.

However, it gives you no chance to have two methods.

Delete the **Brake** method.

3 Now implement the **IWaterVehicle** interface *explicitly* by placing the cursor on **IWaterVehicle** in **AmphibiousVehicle** and using **Ctrl+dot** -> **Implement all members explicitly**.

Delete the **throw NotImplementedException**.

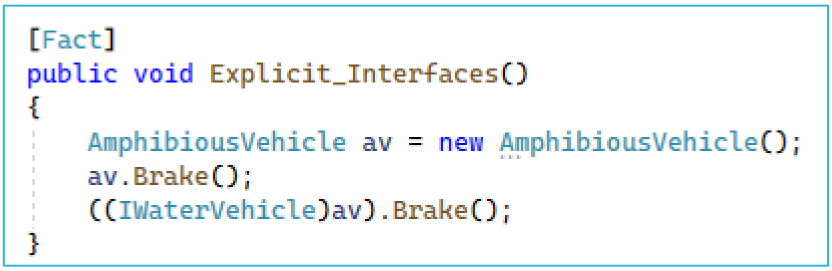
Now place the cursor on **ILandVehicle** and use **Ctrl+dot** -> **Implement interface**.

You get the Brake method.

Delete the **throw NotImplementedException**.

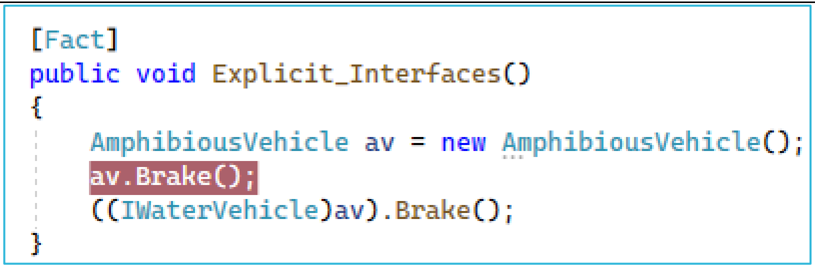
4 You will now use a test to call the implicitly implemented brake method and the explicitly implemented break method.

Add this test to **TestProject**:



```
[Fact]
public void Explicit_Interfaces()
{
    AmphibiousVehicle av = new AmphibiousVehicle();
    av.Brake();
    ((IWaterVehicle)av).Brake();
}
```

Add a breakpoint:



```
[Fact]
public void Explicit_Interfaces()
{
    AmphibiousVehicle av = new AmphibiousVehicle();
    av.Brake();
    ((IWaterVehicle)av).Brake();
}
```

Right-click in the test and choose **Debug Tests.** Step into the code using and **Step Into (F11)** to confirm that both brake methods are called.

Database Access using the Entity Framework

An introduction to using the Entity Framework and a database first approach to gain a basic understanding of how to Create, Read, Update and Delete data in a database table..

Note: This lab makes use of the Northwind database hosted in a local instance of SQL Server. If this database does not exist on the machine you are working on you can find a script that will create it in the Assets folder.

1	Locate the **'Labs\05_Database_Access_using_the_Entity_Framework\Begin\'** folder and, within it, create a new console application called **EF1**.
2	In the Solution Explorer window, right click on the EF1 project's Dependencies tab and select the "Manage NuGet Packages…" option. In the Browse tab of the window search for "Microsoft.EntityFrameworkCore" and then select and install the following: Microsoft.EntityFrameworkCore Microsoft.EntityFrameworkCore.SqlServer Microsoft.EntityFrameworkCore.Design Microsoft.EntityFrameworkCore.Tools Selecting "I Accept" in the pop-up dialog each time.
3	Select Visual Studio's Tools menu > NuGet Package Manager >Package Manager Console. Enter the following as one continuous line of code: Scaffold-DbContext -provider Microsoft.EntityFrameworkCore.SqlServer -connection "Data Source=(local);Initial Catalog=Northwind;trusted_connection=true;Encrypt=False" -OutputDir "Models" -Project "EF1" After a few moments the build should succeed, and a folder called Models will be added to the project. The folder will be filled with a number of source code files each one of

which defines a class with properties that map to a corresponding data base table and its columns.

4 There will be an additional file in the Models folder called NorthwindContext.cs which hosts:

- a class that inherits DbContext
- A set of DbSet properties that map to the imported classes
- An override of the OnConfiguring method that sets up the connection string to the Northwind database.
- A set of calls to the modelBuilder’s generic Entity method that use the fluent API to describe how the application's entity types are mapped to the underlying database.

5 Locate Program.cs and delete all the code it contains, replacing it with the following:

```
using (NorthwindContext con = new NorthwindContext())
{
    var customers = con.Customers;
    foreach (var customer in customers)
    {
        Console.WriteLine($"Company Name: {customer.CompanyName}, Contact 
Name: {customer.ContactName}, Phone: {customer.Phone}");
    }
}
```

The code creates an instance of the NorthwindContext object which is the “magical” link to the database. It can be used to refer to collections of objects that map to rows within the database tables. In this example by simply referring to the context object’s (con) Customers property we gain access to a collection of Customer objects whose data will be retrieved from the associated rows in the Customers table in the database. The code then loops around the collection and prints information about each customer to the console.

If you run the code you should see the following output (we’ve only shown the first 2 and last lines).

```
Company Name: Alfreds Futterkiste, Contact Name: Maria Anders, Phone: 030-0074321
Company Name: Ana Trujillo Emparedados y helados, Contact Name: Ana Trujillo, Phone: (5) 555-4729
...
Company Name: Wolski  Zajazd, Contact Name: Zbyszek Piestrzeniewicz, Phone: (26) 642-7012
```

6 We can add a “Where” clause to limit the number of Customers being returned. In the following example the resulting collection will only contain customers whose company names start with the letter “A”:

```
using (NorthwindContext con = new NorthwindContext())
{
    var customers =
        con.Customers .Where(c => c.CompanyName.StartsWith("A"));
    foreach (var customer in customers)
    {
        Console.WriteLine($"Company Name: {customer.CompanyName}, Contact 
Name: {customer.ContactName}, Phone: {customer.Phone}");
    }
}
```

Running the code should give you the following:

```
Company Name: Alfreds Futterkiste, Contact Name: Maria Anders, Phone: 030-0074321
Company Name: Ana Trujillo Emparedados y helados, Contact Name: Ana Trujillo, Phone: (5) 555-4729
Company Name: Antonio Moreno Taquería, Contact Name: Antonio Moreno, Phone: (5) 555-3932
Company Name: Around the Horn, Contact Name: Thomas Hardy, Phone: (171) 555-7788
```

7 To insert a new row into the database’s `customers` table you simply have to create a new `Customer` object and invoke the appropriate `DbSet` property’s (in this case `Customers`) `Add` method. To ensure the new row is inserted into the database table you also need to call the context object’s `SaveChanges` method. Add the following code to the bottom of the source code.

```
using (NorthwindContext con = new NorthwindContext())
{    Customer cust = new Customer() {
        CustomerId = "AARDV",
    CompanyName = "Aardarks R Us",
    ContactName = "Mr Aardvark",
    Phone = "0123 456789"
    };
    con.Customers.Add(cust);
    con.SaveChanges();
}
```

8 Before attempting to run the code add a `Console.WriteLine()` statement that prints a line of asterisks to the foot of the code (just beneath the line that closes the `using` clause)

9	Just beneath this add a third `using` clause that creates a new `NorthwindContext` object.
10	Within the `using` clause add code that retrieves all customers whose names starts with "**Aa**" and writes the resulting set of rows to the console. (You could do this by copying and adapting the code in the original `using` clause).
11	If you run the program you should see the information about Aardvarks R Us printed beneath the asterisks proving the row must have been successfully added to the database table.

If You Have Time: Remove a row from the database

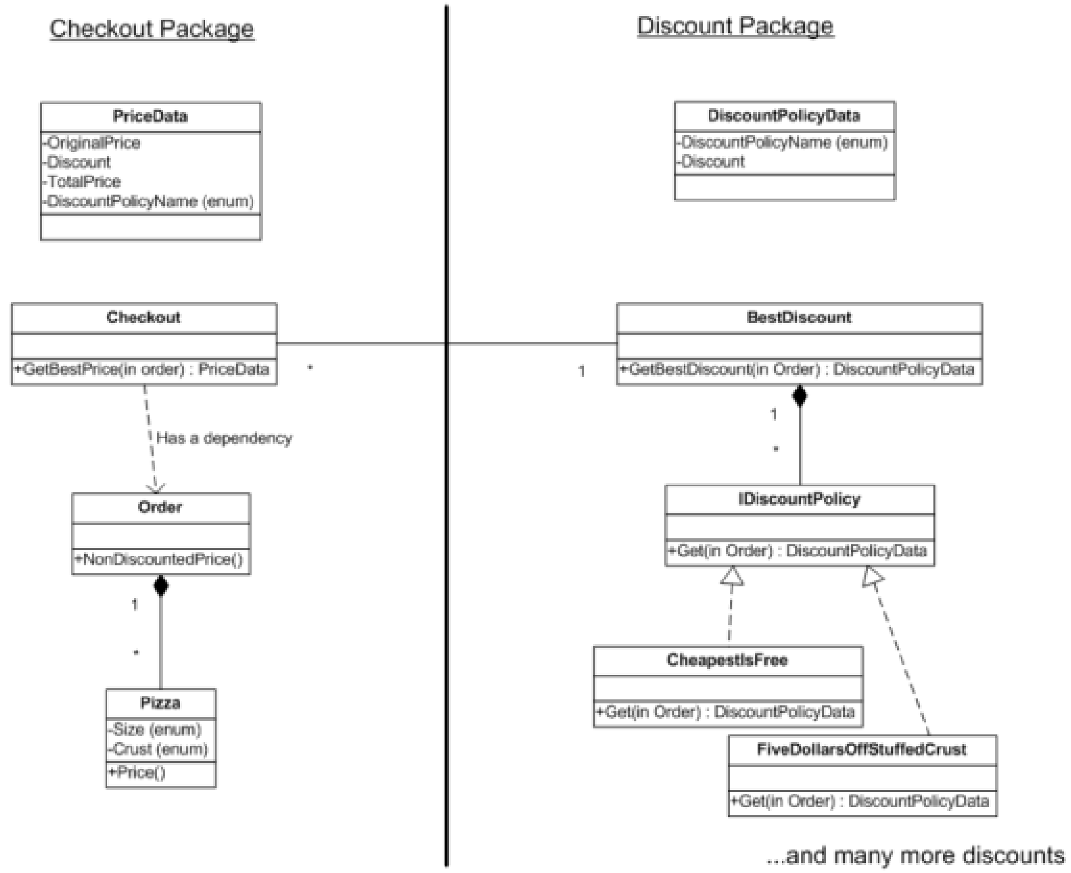
12	Given `DbSet` objects support a `Remove` method that operates in a very similar way to the `Add` method you've already worked with. Try to add code to the foot of the program so it deletes the newly added Aardvarks R Us row from the database table. **Note:** Before running your code, you will need to remove the existing Aardvarks R Us entry in the database's customers table. You can do this by starting up SQL Server Management Studio (SSMS), drilling into the Northwind database via the Object Explorer, Right-Clicking on the customers table and selecting "Edit Top 200 Rows", Rick-Clicking the Aardvarks R Us row and selecting Delete from the menu that pops up.
13	Investigate how to go about updating an existing table row's data. For example change Aardvarks R Us's contact name from "Mr Aardvark" to "Ms Abigail Aardvark". **Note:** There is no update method. Instead, you need to retrieve the appropriate `Customer` object from the Customers `DbSet` (you can use the `SingleOrDefault` method to do this using a lambda expression such as: `c => c.CustomerId == "AARDV"` All you then need to do is alter the `Customer` object's property settings and call the context object's `SaveChanges` method

Delegates and Lambdas

The objective of this exercise is to consolidate your understanding of delegates and lambdas.

Scenario

We have the following classes and interfaces being used by a Pizza Ordering application.


Checkout Package
Discount Package
PriceData
-OriginalPrice
-Discount
-TotalPrice
-DiscountPolicyName (enum)
DiscountPolicyData
-DiscountPolicyName (enum)
-Discount
Checkout
+GetBestPrice(in order) : PriceData
*
1
BestDiscount
+GetBestDiscount(in Order) : DiscountPolicyData
Has a dependency
1
*
Order
+NonDiscountedPrice()
IDiscountPolicy
+Get(in Order) : DiscountPolicyData
1
*
Pizza
-Size (enum)
-Crust (enum)
+Price()
CheapestIsFree
+Get(in Order) : DiscountPolicyData
FiveDollarsOffStuffedCrust
+Get(in Order) : DiscountPolicyData
...and many more discounts

An Order may contain one or many Pizzas.

Checkout needs to know the best discount to apply. We have a BestDiscount class to do this for us, which has a list of all available discounts. It passes in the order to each discount to see which one gets the best discount. It needs to return both the discount price and the name of the discount with said price. It does this by packaging the result into a DiscountPolicyData object and returning that to the Checkout.

The Checkout needs to know the price and name of the applied best discount, as well as what that discount was, the original price, hence the total to pay.

This exercise will take around 30 minutes.

1	Open the **'..\Labs\06_Delegates_and_Lambdas\Pizza'** project and compile it.
2	To familiarise yourself with the problem, have a look at these classes: **Pizza** Notice that it has a Price property and it calculates the price from the size and the crust. To make it easier for you to follow the discounts, we have suffixed the size and crust with the price (so Small is actually Small_10 because its $10). We wouldn't do this in real life because it would be hard to maintain if the prices change, but for our purposes, it makes the tests easier to understand. **Order** Order is a List of pizzas. Notice how it sums the prices of each pizza to get the non-discounted total. **Checkout** The GetBestPrice is passed the order (which is a List of pizzas). It passes this off to the bestDiscount object to work it out. **BestDiscount** Has a list of IDiscountPolicies. To get the best discount, it asks each discount policy what its discount would be for this order. Just as an aside here, notice how we have an abstract class BestDiscount that is subclassed by various strategies (Weekday,

	Weekend, etc.) where we can apply different rules in different circumstances. **DiscountPolicyData** In BestDiscount, it compares these to see which gives the best discount. We have to implement IComparable. Finally, the policies themselves e.g., **CheapestIsFree**. There is no discount here if the order consists only of one pizza. If it's more than one, it loops round and remembers the cheapest.
3	Now look at the tests in **CheckoutTests**. We've written a few tests that return the discount, and it's here where the Small_10, Thin_4 etc. will help you see that it really has selected the best discount.
4	Run all tests and confirm they all pass.

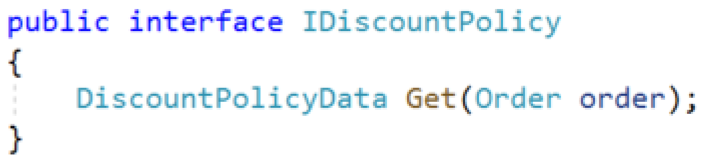
As you can see, we can use Interfaces to achieve the decoupling we need – the BestDiscount class has no knowledge of the individual discount policies. It does have knowledge of IDiscountPolicy and all discount policies implement that.

But it is a bit clumsy:

For each discount policy, we must create a new class that implements IDiscountPolicy. That class has just one method which must be called 'Get'.

In this instance, we could have a neater syntax that doesn't care about the name of the discount policy method and just cares about the signature.

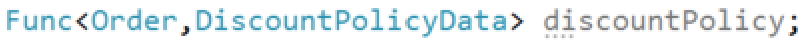
Look at the signature of the 'Get' method:



```
public interface IDiscountPolicy
{
    DiscountPolicyData Get(Order order);
}
```

We would describe this as a method that takes one **Order** as a parameter and returns something of type **DiscountPolicyData**.

We can use the **Func<T, TReturns>** delegate for this:



```
Func<Order,DiscountPolicyData> discountPolicy;
```

We refer to it as a generic delegate because you can provide any type. Note that, for Func<>, the last type is always the return type.

5	Delete the file **IDiscountPolicy.cs**
6	In the Discounts folder, create a **public static class DiscountPolicies**. This will contain all our policies. Change the namespace to just **PizzaProject**

	For each of the policies in the Policies folder, copy the **Get()** method and paste it into the class DiscountPolicies, replacing 'Get' with the name of the original class. Make the method static.
	For example, the method **CheapestIsFree.Get** becomes:
	`public static class DiscountPolicies`
	`{`
	`  public static DiscountPolicyData CheapestIsFree(Order order)`
	`  {`
	`    System.Diagnostics.Debug.WriteLine("CheapestIsFree");`
	`    var pizzas = order.Pizzas;`
	`    if (pizzas.Count < 2)`
	`    {`
	`      return new DiscountPolicyData(DiscountPolicyName.None, 0M);`
	`    }`

```
    // Loop round all pizzas in the order and get the one with the minimum price
    decimal minPrice = decimal.MaxValue;
    foreach (Pizza pizza in pizzas)
    {
      if (pizza.Price < minPrice)
      {
        minPrice = pizza.Price;
      }
    }
    return new DiscountPolicyData(DiscountPolicyName.Cheapest_Is_Free, minPrice);
  }
}
```

Repeat for all discount policies.

7 Delete the folder **Policies** and the three files within it

8 If you compile now, it will fail because **BestDiscount** still relies on the now deleted **IDiscountPolicy**. Change this to:

```
protected List<Func<Order,DiscountPolicyData>> policies
                     { get; private set;}
                  = new List<Func<Order, DiscountPolicyData>>();
```

9 Also, in BestDiscount, we now don’t have classes like CheapestIsFree.

```
policies.Add(new CheapestIsFree());
```

But we do have methods that satisfy the signature of Func<Order,DiscountPolicyData>>

Change it to:

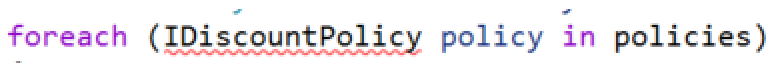
```
policies.Add(DiscountPolicies.CheapestIsFree);
```

10 Repeat for the other two policies.

To make the code even simpler, put in a using statement:

```
using static PizzaProject.DiscountPolicies;
```

11 We are then left with one problem:



```
foreach (IDiscountPolicy policy in policies)
```

Resolve this as follows:

```
foreach (Func<Order,DiscountPolicyData> policy in policies)
{
    DiscountPolicyData thisOrdersPolicyData = policy(order);
```

Run the tests. They should all pass.

Have a think about what you've just done.

You had an interface with one method and wherever you define such a method it must be packaged into a class and you will probably never re-use either the class or the method. You have replaced this with a pointer to a method of the required signature.

Now we have delegates in play, we can use lambda expressions.

12 Add this test to your **CheckoutTests**.

```
[Fact]
public void Weekend_Ten_Percent_Off()
{
    checkout = new Checkout(new WeekendDiscounts());
    order.Pizzas.Add(new Pizza(Size.Small_10, Crust.Regular_2));
    PriceData priceData = checkout.GetBestPrice(order);


    Assert.Equal(10.8M, priceData.TotalPrice);

    Assert.Equal(DiscountPolicyName.Weekend_10_Percent_Off,
priceData.DiscountPolicyName);

}
```

We want a new discount policy to operate on weekends where you get 10% off.

13 Add this to the **DiscountPolicyName enum**:

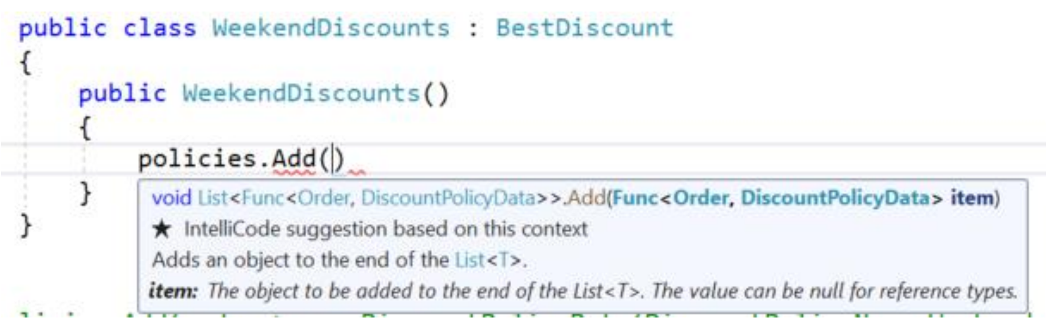
```
    Weekend_10_Percent_Off
```

14 In the BestDiscount file, there is a class WeekendDiscounts.

Add a constructor. Within the constructor type the code:

policies.Add(

Have a look at the IntelliSense:



```
public class WeekendDiscounts : BestDiscount
{
    public WeekendDiscounts()
    {
        policies.Add(|)
    }
}
```

void List<Func<Order, DiscountPolicyData>>.Add(Func<Order, DiscountPolicyData> item)
★ IntelliCode suggestion based on this context
Adds an object to the end of the List<T>.
item: The object to be added to the end of the List<T>. The value can be null for reference types.

You will see it is expecting a parameter of type Func that takes an Order and returns a DiscountPolicyData.

<table>
<tr><td></td><td>That means we can either point to a method of this signature (which is what we have done in the WeekdayDiscounts constructor) or we could put in a lambda expression.</td></tr>
<tr><td>15</td><td>

Put in this lambda:

```
policies.Add(order=> new
  DiscountPolicyData(DiscountPolicyName.Weekend_10_Percent_Off,
                                order.NonDiscountedPrice * 0.1M) );
```

Run the **Weekend_Ten_Percent_Off** test and confirm it passes.

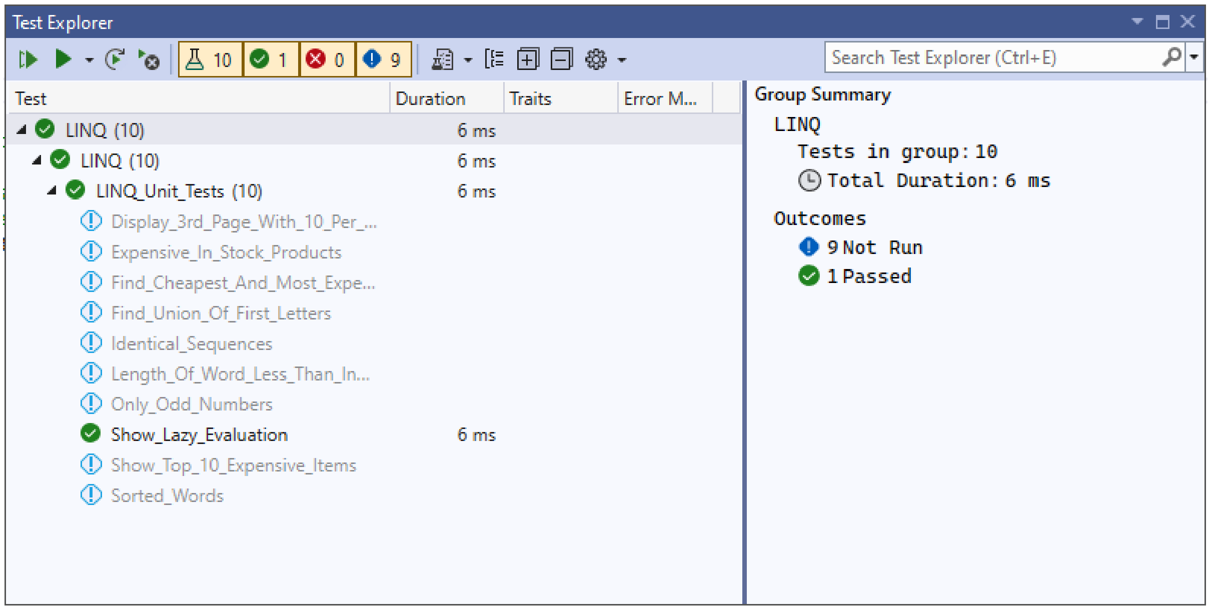
</td></tr>
</table>

Language Integrated Query (LINQ)

The objective of this exercise is to give you a wide view of the capabilities of Language Integrated Query (LINQ).

Writing LINQ Queries

1. Open **'..\Labs\07_LINQ\Begin\LINQ_Solution.sln'**
2. Open the **LINQ_Unit_Tests.cs** file and read the instructions at the top of the file.
3. Each test shows an imperative way of solving a problem. At the end of each test are clues as to how you might get started solving the same problem with LINQ. Have a go solving the problem with LINQ.
4. After writing the LINQ equivalent code, run each unit test by right-clicking within the unit test and choosing Run Tests. Ensure each test still passes.
5. As you progress, do a visual comparison of how much briefer the declarative LINQ solutions are compared to the provided non-LINQ solutions.

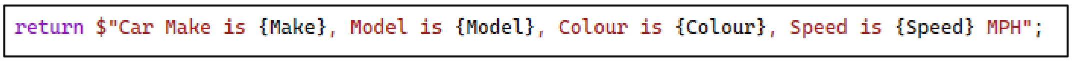


Test	Duration	Traits	Error M...
✓ LINQ (10)	6 ms		
✓ LINQ (10)	6 ms		
✓ LINQ_Unit_Tests (10)	6 ms		
! Display_3rd_Page_With_10_Per_...			
! Expensive_In_Stock_Products			
! Find_Cheapest_And_Most_Expe...			
! Find_Union_Of_First_Letters			
! Identical_Sequences			
! Length_Of_Word_Less_Than_In...			
! Only_Odd_Numbers			
✓ Show_Lazy_Evaluation	6 ms		
! Show_Top_10_Expensive_Items			
! Sorted_Words			

```
Group Summary
 LINQ
   Tests in group: 10
   Total Duration: 6 ms

 Outcomes
   ! 9 Not Run
   ✓ 1 Passed
```

Exception Handling

The objective of this exercise is to consolidate your understanding of exception handling and creating and throwing custom exceptions.

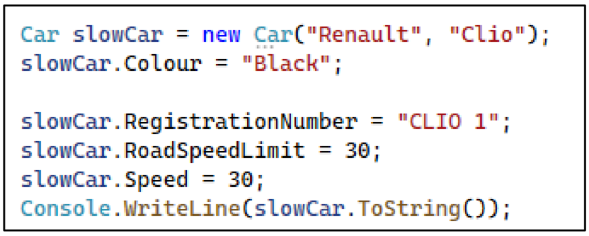
1	Open the **CarLibrary** solution in: **'..\Labs\08_Exception_Handling\Begin'**
2	Comment out the existing code in **Program.cs**
3	Open **Car.cs** and override the **ToString** method to display detailed car information:  `return $"Car Make is {Make}, Model is {Model}, Colour is {Colour}, Speed is {Speed} MPH";`
4	Add a new auto-implemented property called **RoadSpeedLimit**.
5	You will change the logic of the Speed setter to account for the road's speed limit and whether or not the value that you are setting is a legal driving speed for the current road. - If it is, set the value - If it is not, you will raise a custom exception
6	In a separate file in the class library project, create an exception class called **SpeedingException**. Don't forget to use inheritance.

7 Within the new custom exception class, create an auto-implemented property called **ExcessSpeed** and set this value within the constructor.

8 In **Car.cs**, if the car is not travelling at a legal speed, throw a new instance of **SpeedingException,** ensuring you pass in the excess speed value.

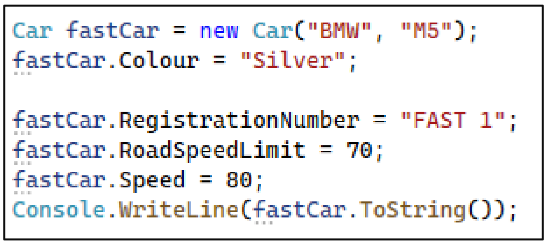
9 In **Program.cs**, create two new car instances: **slowCar** and **fastCar**

10 Set the following values for **slowCar**:



```
Car slowCar = new Car("Renault", "Clio");
slowCar.Colour = "Black";

slowCar.RegistrationNumber = "CLIO 1";
slowCar.RoadSpeedLimit = 30;
slowCar.Speed = 30;
Console.WriteLine(slowCar.ToString());
```

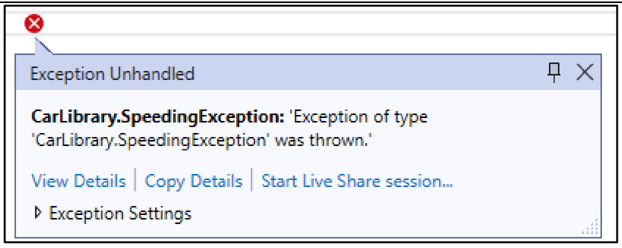
11 Set the following values for **fastCar**:



```
Car fastCar = new Car("BMW", "M5");
fastCar.Colour = "Silver";

fastCar.RegistrationNumber = "FAST 1";
fastCar.RoadSpeedLimit = 70;
fastCar.Speed = 80;
Console.WriteLine(fastCar.ToString());
```

12 Compile and run your application.

You should see an unhandled exception:



Exception Unhandled

CarLibrary.SpeedingException: 'Exception of type 'CarLibrary.SpeedingException' was thrown.'

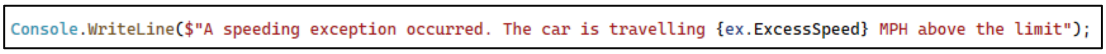
13 Uncomment the existing code in **Program.cs**.

Set the **RoadSpeedLimit** to **50** for Car **c2.**

Wrap the code in this file with **try...catch...finally** blocks to handle the exceptions that are thrown.

Add a catch block for **Exception** as well as for **SpeedingException**.

Utilise the properties of the exception class to display useful messages to the console:

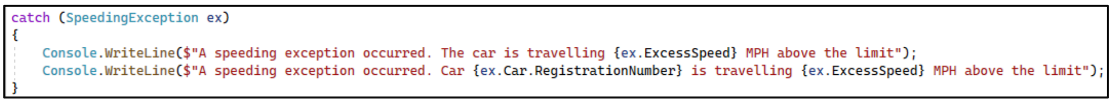


```
Console.WriteLine($"A speeding exception occurred. The car is travelling {ex.ExcessSpeed} MPH above the limit");
```

14 Compile and run your application.

Observe the exceptions that are thrown and caught.

15 Add a property to store the *Car instance* within the **SpeedingException** and use this to access information that can be output to the console to help identify the Car that is speeding:



```
catch (SpeedingException ex)
{
    Console.WriteLine($"A speeding exception occurred. The car is travelling {ex.ExcessSpeed} MPH above the limit");
    Console.WriteLine($"A speeding exception occurred. Car {ex.Car.RegistrationNumber} is travelling {ex.ExcessSpeed} MPH above the limit");
}
```

1.1. If you have time

16	Create a list of valid colours within **Car.cs** and create a custom **InvalidColourException** that is thrown if the colour is not in the list.
17	Observe how this exception is caught by the generic Exception event handler.
18	Add a custom catch block to handle this specific type of exception.
19	A suggested solution is provided in the **End** folder for your reference.

90 Working with Data and Files

Working with JSON data

1	Create a console app called JSON_Films.
2	Use NuGet to add the Newtonsoft.Json package
3	Add a class to the project that is suitable for holding film information (film_id, title, synopsis, director, release_date).
4	Using the file called Films.json in the Assets folder deserialize a JSON file containing films into a List<Film> object.
5	Display the film information on a console screen.
6	Add another film to the list.
7	Serialize the list back out to the JSON file.
8	If you have time, duplicate the code but use dynamic / anonymous types rather than purpose-built classes.

Reading and Writing CSV files

9	Create a console app called MoviesCSVFile..
10	Use NuGet to add the CSVHelper package
11	Add a class to the project that is suitable for holding film information (Title, ReleaseYear, Genres, Revenue and StreamedOn).
11	Write code to read the CSV file "streaming_movies.csv" into a list using a custom class suitable for the data within each row. You will find the file in the Assets folder.
12	Filter the list to get just movies on the Netflix platform, and sort it by movie title.

13	Write the filtered list of movies to a new CSV file, but only including the columns: Title, ReleaseYear, and Revenue.

Streams

14	Open the Visual Studio solution called Streams.sln located in the Begin folder.
15	Review the code that already exists. You will notice there are functions called CompressIntoByteArray, DecompressByteArrayIntoString, EncryptString and DecryptString. Three of these functions contain ready written (and functioning code) but the DecompressByteArrayIntoString currently does nothing with the passed in byte array.
16	Using the CompressToByteArray function as a guide try to add appropriate code to the DecompressByteArrayIntoString to make the decompression process work.

Before moving on don't forget to commit and push your work to GitHub.

Learn. To Change.

QA.com