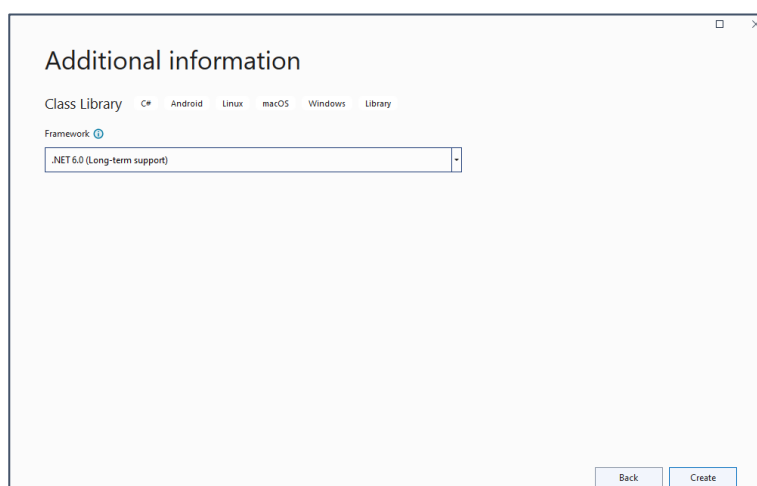
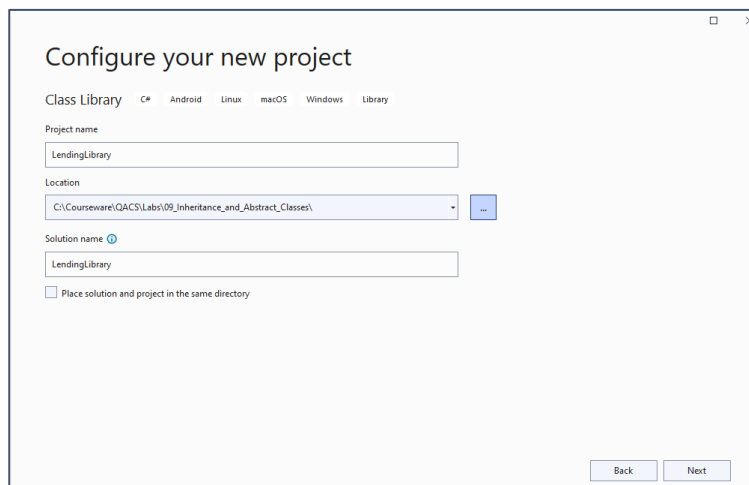
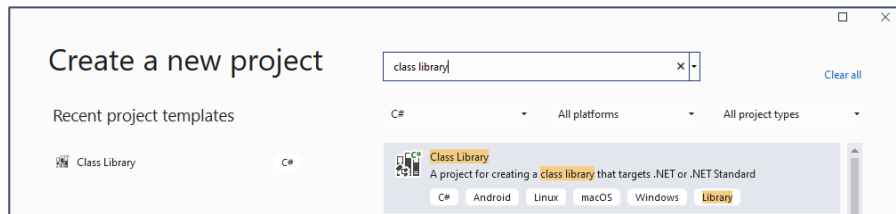


Inheritance

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

- 1 Create a new Visual Studio project of type Class Library called **LendingLibrary**.



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 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 greta = library.Add(name: "Greta Thunberg", age: 15);
    Member donald = donald = library.Add(name: "Donald Trump", age: 73);

    Assert.Equal(2, library.NumberOfMembers);
    Assert.Equal(1, greta.MembershipNumber);
    Assert.Equal(2, donald.MembershipNumber);
}
```

- 3 Create Library and Member classes in the LendingLibrary project, then copy in the code listed just after the 'Create' test.

```
namespace LendingLibrary
{
    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;
        }
    }
}
```

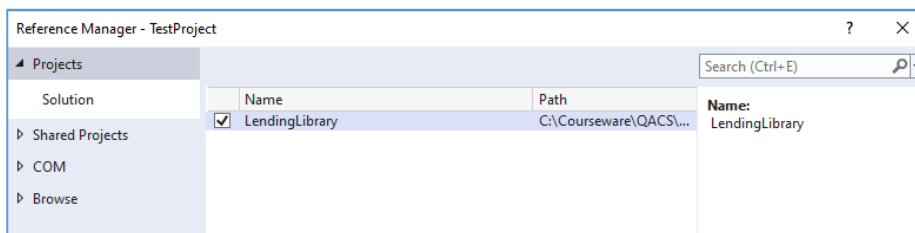
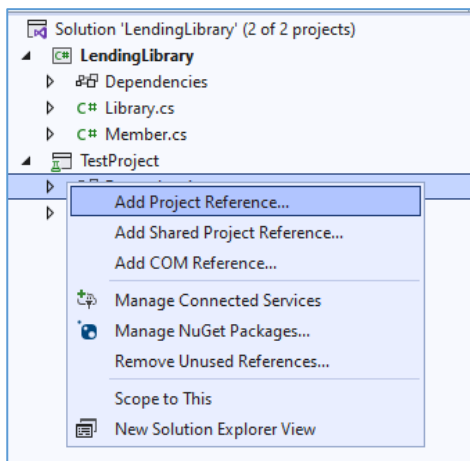
```

namespace LendingLibrary
{
    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;
        }
    }
}

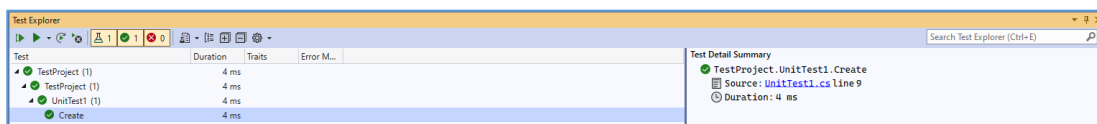
```

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



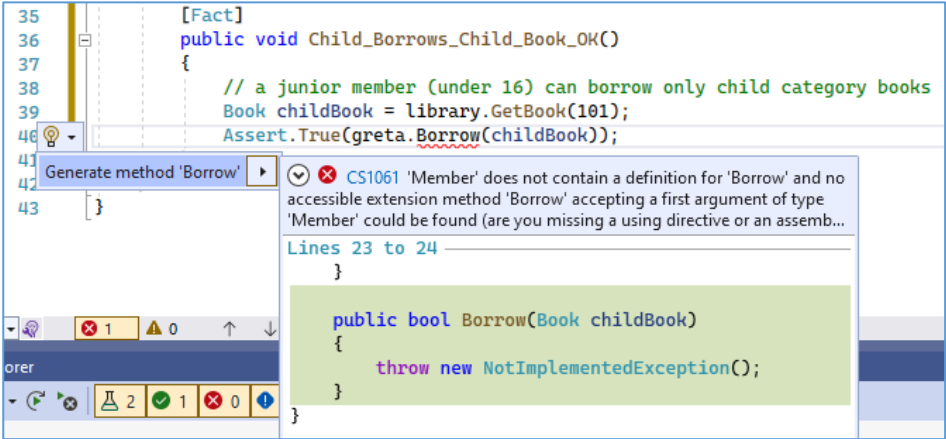
Add a using statement `'using LendingLibrary;'` to **UnitTest1.cs**

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



- 4 We are going to need the **library** instance and **greta** and **donald** 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.

| | |
|---|--|
| | <p>Note: If you get stuck, we've shown it in the Assets folder.</p> |
| 5 | <p>Later on, we're going to need some additional properties of Member, so add these in and get Visual Studio to create the properties.</p> <pre> public UnitTest1() { library = new Library(); greta = library.Add(name: "Greta Thunberg", age: 15); greta.Street = "Queen Street"; greta.City = "Stockholm"; greta.OutstandingFines = 25M; donald = library.Add(name: "Donald Trump", age: 73); donald.Street = "Trump Tower"; donald.City = "New York"; donald.OutstandingFines = 2500M; } </pre> |
| 6 | <p>In this library we have different rules for borrowing a book, dependent on whether the member is over 16 or not.</p> <p>We are going to need this enum. Add it to your LendingLibrary, in a file called BookCategory.cs</p> <pre> namespace LendingLibrary { public enum BookCategory { Children, Adult } } </pre> |
| | <p>We will need a Book class. Add the class and the properties and get Visual Studio to generate the constructor:</p> <pre> 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; } } </pre> <p>We will also need some Book code in the Library class:</p> |

| | |
|---|---|
| | <pre> 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)); } </pre> |
| 7 | <p>In the TestProject, add in the Child_Borrows_Child_Book_OK test from the Assets folder.</p> <p>Get Visual Studio to resolve the Borrow() method:</p>  <pre> [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(greta.Borrow(childBook)); } </pre> <p>Populate the method like this:</p> <pre> public bool Borrow(Book book) { return true; } </pre> <p>Run the Test and confirm it passes.</p> |
| 8 | <p>Add in the test Child_Borrows_Adult_Book_Fails.</p> <pre> [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(greta.Borrow(adultBook)); } </pre> <p>Run this Test. It fails because it is currently hard-coded to return true.</p> |

| | |
|----|---|
| | <p>Now we need to modify Borrow:</p> <pre>public bool Borrow(Book book) { return book.Category == BookCategory.Children; }</pre> <p>Re-run the test. It should now pass.</p> |
| 9 | <p>Add the test Adult_Can_Borrow_Any_Book.</p> <pre>[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(donald.Borrow(adultBook)); Assert.True(donald.Borrow(childBook)); }</pre> <p>Run this Test. It fails because the Borrow() method only returns true for children's books.</p> <p>Modify the Borrow code again:</p> <pre>public bool Borrow(Book book) { if (Age >= 16) { return true; } else { return (book.Category == BookCategory.Children); } }</pre> <p>Run the Tests. All tests should now pass.</p> |
| 10 | <p>In this library, fines are handled differently for juniors and adults. Juniors must provide a CashFund; Adults must provide BankTransfer details.</p> <p>Add in the two 'Fines' tests:</p> |

```
[Fact]
public void Child_Pays_Fine_From_Cash_Fund()
{
    greta.CashFund = 20M;
    greta.PayFine(7M);
    Assert.Equal(13M, greta.CashFund);
}

[Fact]
public void Adult_Pays_Fine_By_Bank_Transfer()
{
    donald.SetupBankTransferLimit(20M);
    donald.PayFine(7M);
    Assert.Equal(13M, donald.BankTransferAvailable);
}
```

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 |
|--------------------------------|

| | |
|----|---|
| | <ol style="list-style-type: none"> 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: <ol style="list-style-type: none"> a. 01 Without Inheritance b. 02 With Inheritance 6) And move Member1.cs to 01 Without Inheritance and Member2.cs to 02 With Inheritance. <p>Your project now only has one uncommented Member class in Member2.cs.</p> <p>All tests will pass as no code has been changed.</p> |
| 12 | <p>In the 02 With Inheritance folder, add two new classes: JuniorMember and AdultMember.</p> <p>Adjust their namespaces to be just LendingLibrary.</p> |
| 13 | <p>Get both of these subclasses to derive from the Member base class.</p> <p>Implement a constructor that passes all parameters to the base class constructor:</p> <pre>namespace LendingLibrary { public class JuniorMember : Member { public JuniorMember(string name, int age, int membershipNumber) : base(name, age, membershipNumber) { } } }</pre> <p>Do the same for AdultMember.</p> |
| 14 | <p>We need to modify Library.Add to create either a <i>Junior</i> or an <i>Adult</i> member. Change Library.Add() to this:</p> |

| | |
|----|---|
| | <pre> 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; } </pre> <p>All tests should pass.</p> |
| 15 | <p>Actually, we want to disallow creating 'Member' objects – clients should be forced to create either Junior or Adult members.</p> <p>Make the Member class abstract.</p> |
| 16 | <p>In Member2.cs, copy Borrow(), CashFund, PayFine(), BankTransferAvailable and SetUpBankTransferLimit() into notepad, deleting them from Member.</p> |
| 17 | <p>In Member, insert these two abstract methods:</p> <pre> public abstract bool Borrow(Book book); public abstract void PayFine(decimal fine); </pre> <p>These abstract methods replace the concrete versions we just deleted.</p> |
| 18 | <p>Go to JuniorMember. You will see a red squiggly.</p> <pre> public class JuniorMember : Member { public JuniorMember(base(name, ag </pre>  <p>Using Ctrl+dot, implement the Abstract class. This will put in the signatures of the methods defined in Member:</p> |

| | |
|----|--|
| | <pre> public override bool Borrow(Book book) { throw new NotImplementedException(); } public override void PayFine(decimal fine) { throw new NotImplementedException(); } </pre> |
| 19 | <p>Inside each of these members, copy the relevant code from that which you stored in notepad that is specific just to <i>Junior</i> members</p> <p>Repeat for AdultMember.</p> <ul style="list-style-type: none"> 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 | <p>If you now go back to the tests, you can see that it doesn't know that greta is a JuniorMember and Donald is an AdultMamber:</p> <pre> public class UnitTest1 { Library library; Member greta; Member donald; } </pre> <p>There are two ways of fixing this:</p> <ol style="list-style-type: none"> 1) Make greta a Junior and donald 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. <p>We'll do both...</p> |
| 21 | Make these changes: |

| | |
|--------|---|
| | <pre> Library library; JuniorMember greta; AdultMember donald; public UnitTest1() { library = new Library(); greta =(JuniorMember) library.Add(name: "Greta Thunberg", age: 15); greta.Street = "Queen Street"; greta.City = "Stockholm"; greta.OutstandingFines = 25M; donald = donald = (AdultMember)library.Add(name: "Donald Trump", age: 73); donald.Street = "Trump Tower"; donald.City = "New York"; donald.OutstandingFines = 2500M; } </pre> |
| 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, <i>StudentMember</i> , you will have to modify the client code. |
| 23 | Use Ctrl+z (Undo) to remove the changes you made in step 21. |
| 2 4 | <p>A better way of looking at it is:</p> <p><i>'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></p> <p>i.e., the same intent but they have different implementations?</p> <p>How about SetFineLimit() and GetFineCredit() ?</p> |
| 25 | <p>Make these changes:</p> <p>TestProject</p> |

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

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

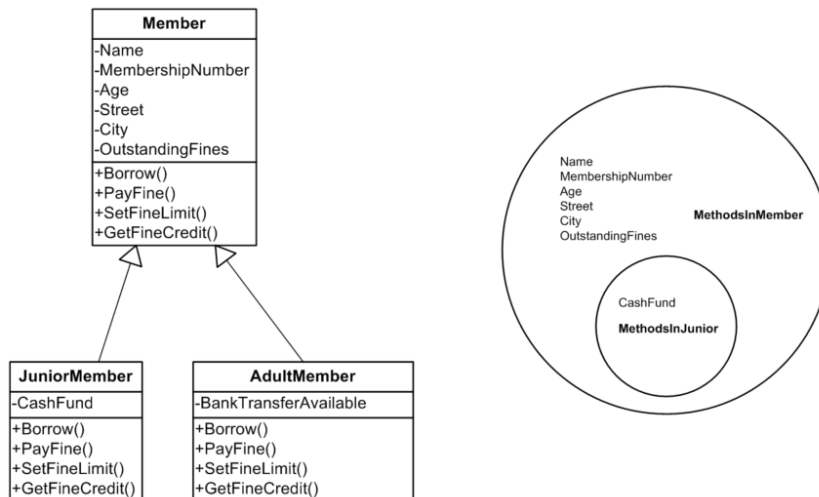
What happens when Greta turns 16?

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:

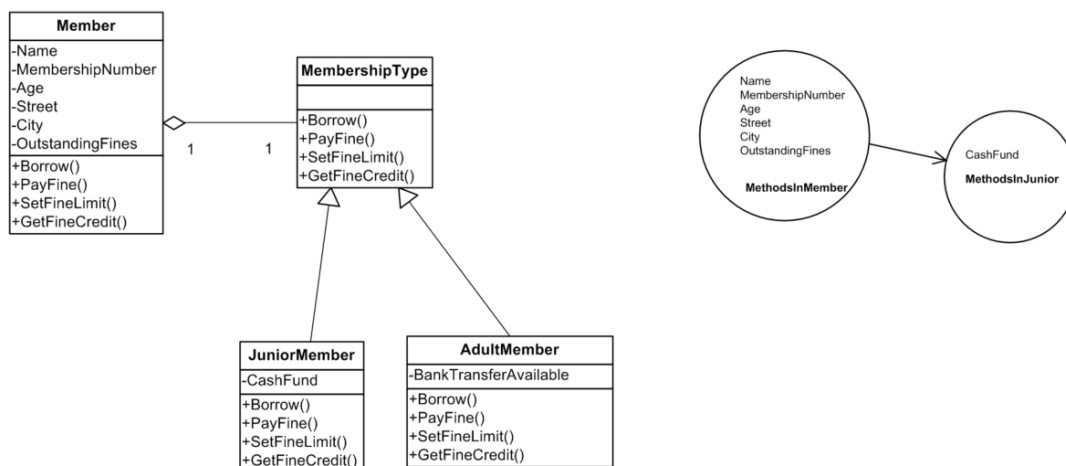
1. You have to transfer all the data (name, street, city, etc.) - what if some of that data is private?
2. Greta will feature in various collections. She would need to be removed from these and the new Greta would need to be inserted silently (probably breaking the rules we have established).

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.

