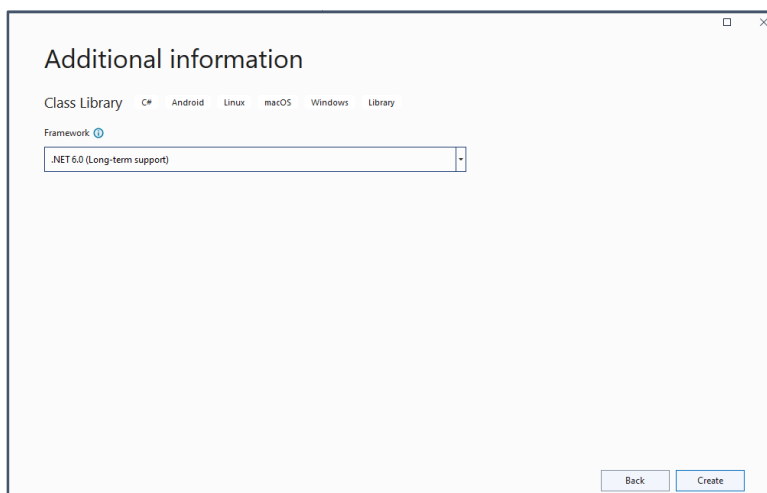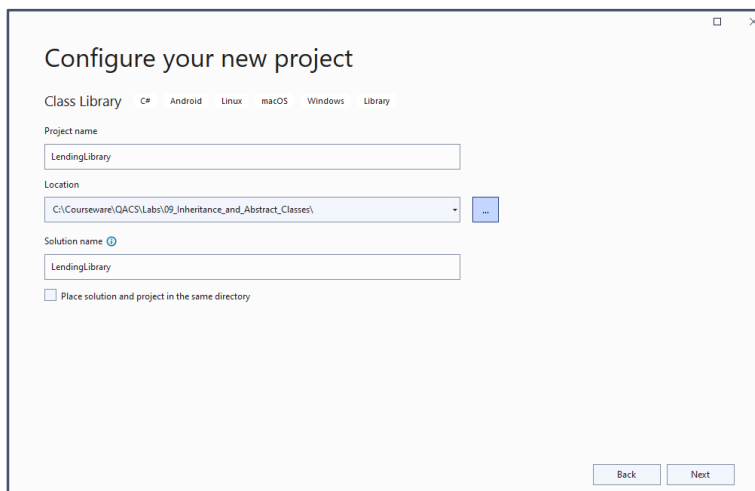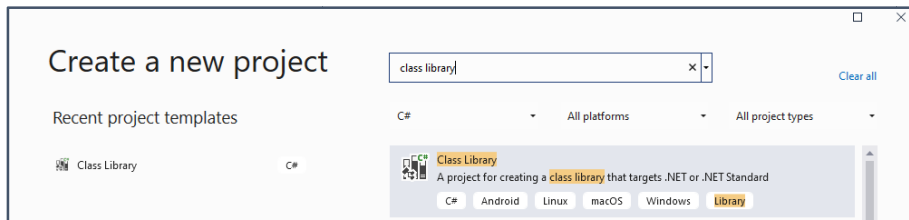# Inheritance

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).

```csharp
[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.

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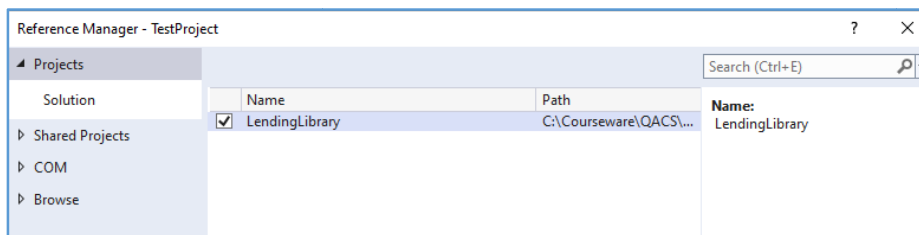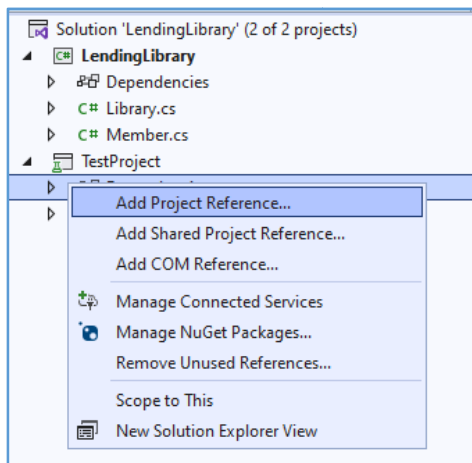
```
namespace LendingLibrary
{
    public class Member
    {
        public string Name { get; }
        public int MembershipNumber { get; }
        public int Age { get; }

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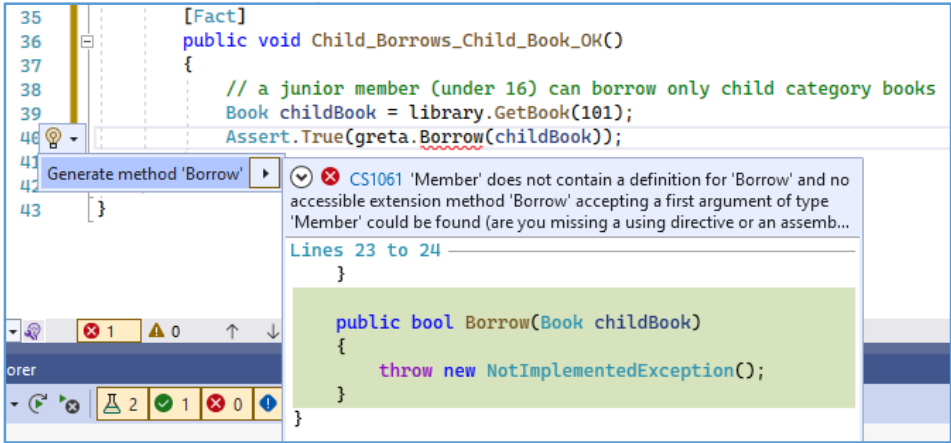
Add a using statement 'using LendingLibrary;' to **UnitTest1.cs**

Run the **Create** test and confirm it passes.
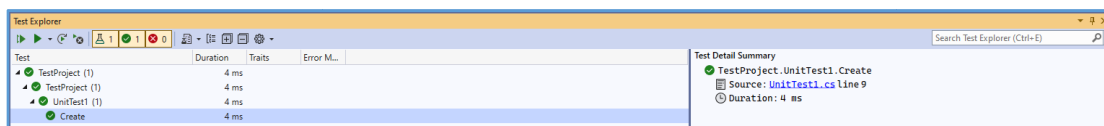
4	We are going to need the **library** instance and **greta** and **donald** member objects in further tests, so to avoid duplication of code, move the declarations of these to the class level and initialise them in the constructor.

	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.
	
	``csharp
	public UnitTest1()
	{
	    library = new Library();
	    greta = library.Add(name: "Greta Thunberg", age: 15);
	    greta.Street = "Queen Street";
	    greta.City = "Stockholm";
	    greta.OutstandingFines = 25M;
	    donald = donald = library.Add(name: "Donald Trump", age: 73);
	    donald.Street = "Trump Tower";
	    donald.City = "New York";
	    donald.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**
	
	``csharp
	namespace LendingLibrary
	{
	    public enum BookCategory
	    {
	        Children,
	        Adult
	    }
	}
	``

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

```csharp
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:

```
35              [Fact]
36              public void Child_Borrows_Child_Book_OK()
37              {
38                  // a junior member (under 16) can borrow only child category books
39                  Book childBook = library.GetBook(101);
40                  Assert.True(greta.Borrow(childBook));
41
42
43              }
```

```
Generate method 'Borrow'    ▶    ⊗ 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 23 to 24
                                     }

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

Populate the method like this:

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

Run the Test and confirm it passes.

8	Add in the test **Child_Borrows_Adult_Book_Fails.**

```csharp
[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));
}
```

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

	Now we need to modify Borrow:
	
	``csharp
	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**.
	
	``csharp
	[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));
	}
	``
	
	Run this Test. It fails because the **Borrow**() method only returns true for children's books.
	
	Modify the **Borrow** code again:
	
	``csharp
	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:

```csharp
[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:

```csharp
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;
}
```

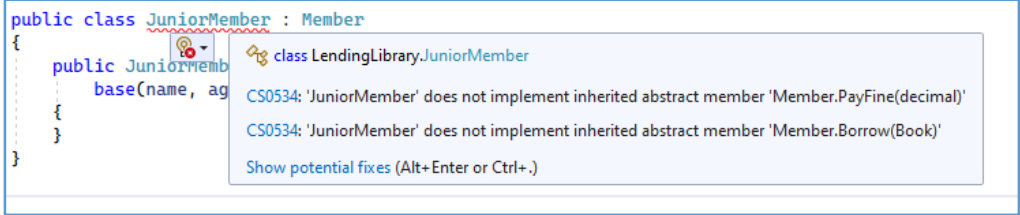Confirm all tests pass.

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 tqo folders in LendingLibrary:
	     a. 01 Without Inheritance
	     b. 02 With Inheritance
	 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.

12	In the 02 With Inheritance folder, add two new classes: **JuniorMember** and **AdultMember**. Adjust their namespaces to be just **LendingLibrary**.
| 13 | Get both of these subclasses to derive from the **Member** base class. <br><br> Implement a constructor that passes all parameters to the base class constructor: <br><br> ```csharp
namespace LendingLibrary
{
    public class JuniorMember : Member
    {
        public JuniorMember(string name, int age, int membershipNumber) :
            base(name, age, membershipNumber)
        {
        }
    }
}
``` <br><br> Do the same for **AdultMember**. |
| 14 | We need to modify **Library.Add** to create either a *Junior* or an *Adult* member. Change Library.Add() to this: |

```
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. 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 greta is a JuniorMember and Donald is an AdultMamber:

```
public class UnitTest1
{
    Library library;
    Member greta;
    Member donald;
```

There are two ways of fixing this:

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.

We'll do both… |
| 21 | Make these changes: |

```
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;
}
```

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

```csharp
[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:

```csharp
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:

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

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

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

AdultMember:

```csharp
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.