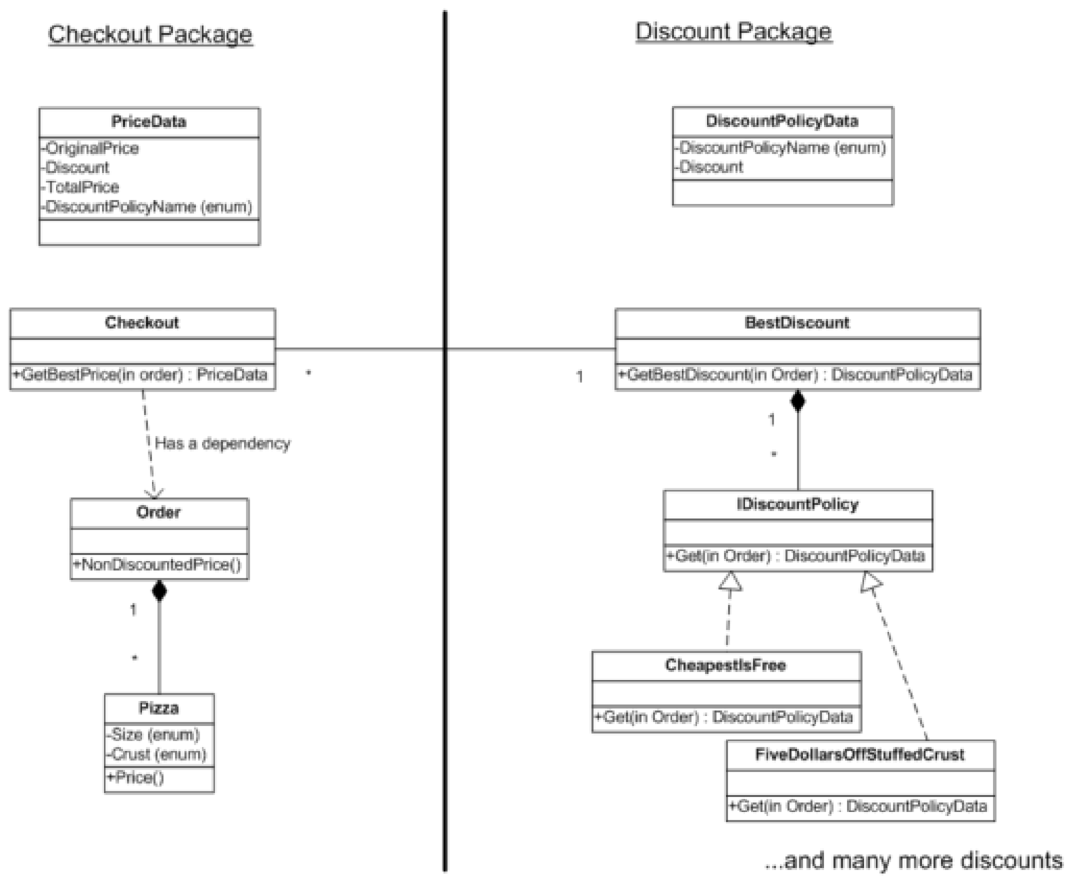
Delegates and Lambdas

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
BestDiscount
+GetBestDiscount(in Order) : DiscountPolicyData
Has a dependency
Order
+NonDiscountedPrice()
Pizza
-Size (enum)
-Crust (enum)
+Price()
IDiscountPolicy
+Get(in Order) : DiscountPolicyData
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 PizzaProjectSolution.sln solution in the **’01 Pizza/Begin’** folder 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 asan 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:

```
publicstaticclassDiscountPolicies
{
publicstaticDiscountPolicyDataCheapestIsFree(Order order)
    {
System.Diagnostics.Debug.WriteLine("CheapestIsFree");
var pizzas = order.Pizzas;
if (pizzas.Count< 2)
        {
returnnewDiscountPolicyData(DiscountPolicyName.None, 0M);
        }

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

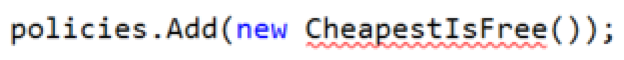
Repeat for all discount policies.

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; privateset;}
= 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:

```
usingstaticPizzaProject.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)
{
DiscountPolicyDatathisOrdersPolicyData = 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.

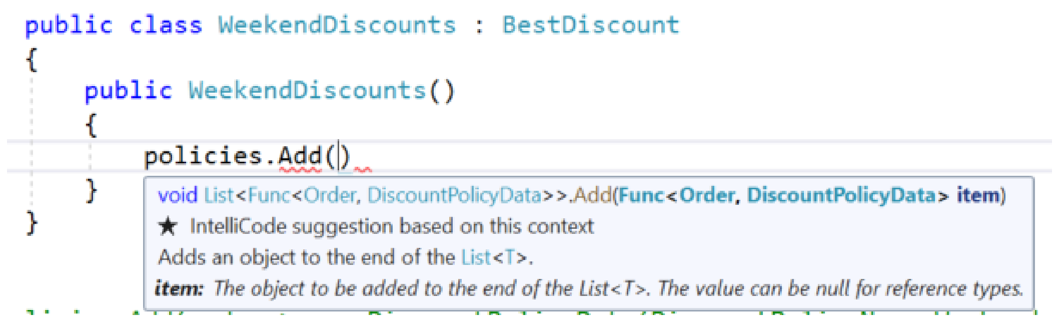
12 Add this test to your **CheckoutTests**.

```
[Fact]
publicvoidWeekend_Ten_Percent_Off()
{
    checkout = newCheckout(newWeekendDiscounts());
order.Pizzas.Add(new Pizza(Size.Small_10, Crust.Regular_2));

PriceDatapriceData = 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 **DiscountPolicyNameenum**:

	`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.
	
	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.

15	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.