Aia Al-Qasab

T5616SN

OBJECT-ORIENTED PROGRAMMING

ASSIGNMENT 4

VISUAL STUDIO CONSOLE APPLICATION

INHERITANCE

September 2017



VISUAL STUDIO CONSOLE APPLICATION

INHERITANCE

Inheritance is hardly the be-all, end-all of OOP, and interfaces are often a better choice, you absolutely should not get the idea that you should avoid using inheritance. Inheritance is a powerful tool that saves you a lot of work if you use it correctly. “Correctly” simply means that you should not use inheritance if it is not clear that the “is a” relationship holds:

• An instance of a child class A that inherits from a parent class B must be usable in every piece of code that would take an instance of its parent type as a parameter.

In other words, if you have a function whose header is

UseIt (bThing as B)

In fact, the way inheritance is used in the .NET Framework is a perfect example of why inheritance should not be completely replaced by interfaces in object-oriented programming. What happened in the .NET Framework may well occur in your own projects:

• When you build frameworks that other programmers will depend on, well-designed, thoroughly debugged base classes can be used over via inheritance as the base on which they build their classes.

The Inherits keyword must be the first nonblank, no comment line after the name of the child class. (IntelliSense will automatically show you what classes you can inherit from, by the way.) Note that a derived class cannot use the Public access modifier if its parent class is marked Friend or Private. The reason is that the access modifier you use for the derived class cannot be less restrictive than the one used for its parent class. But it can be more restrictive, so a class marked Friend class may inherit from one marked Public, for example. The next step in building a child class is to make sure that you give it the correct kind of constructor. Because the child class must have at least the same amount of functionality as its parent class, constructors for child classes usually call the constructor of the parent class to correctly initialize the instance fields of their parent class. This is done using the special keyword MyBase, which accesses the parent class .

VB .NET comes with a Protected access modifier that automatically gives child classes access to the item specified with this modifier. This is true whether the item is a member function or an instance field. You may be tempted to use this modifier to make all instance fields of the parent class Protected, to give derived classes a quick and dirty way to gain access to the parent class’s instance fields. Do not give in to this temptation. Design principles dictate that the Protected access modifier should be used only for member functions, not for instance fields. Doing anything else violates encapsulation and prevents you from doing validation in the place it belongs—the parent class. You need to rely on the parent class to validate the data. As in real life, “trust but verify” is the default behaviour for good parenting.

Unlike many OOP languages, the syntax used in VB .NET makes it clear that you want to override a method in the parent class by a method from the child class. The clarity comes from the two required keywords:

• Overridable, which is used in the parent class to indicate that a method can be overridden.

• Overrides, which is used in the child class to indicate that you are overriding a method.

VB .NET’s Not Inheritable keyword prevents inheritance from a class (these kinds of classes are sometimes called sealed or final classes). The main reason to mark an entire class as Not Inheritable is if the class has such vital behaviour that you cannot risk changes to it. Many framework classes such as String are marked as Not Inheritable for this reason. However, you do not have to mark a whole class as Not Inheritable if all you want to do is place a single member of the class off limits to overriding it: you can mark a member you do not want overridden as Not Overridable.

The Class View and the Object Browser are useful when your inheritance chain grows in complexity.

**1.Experiment with a simple class hierarchy. Your base class properties are as follows:**

namespace OOPAia4A

{

class Program

{

static void Main(string[] args)

{

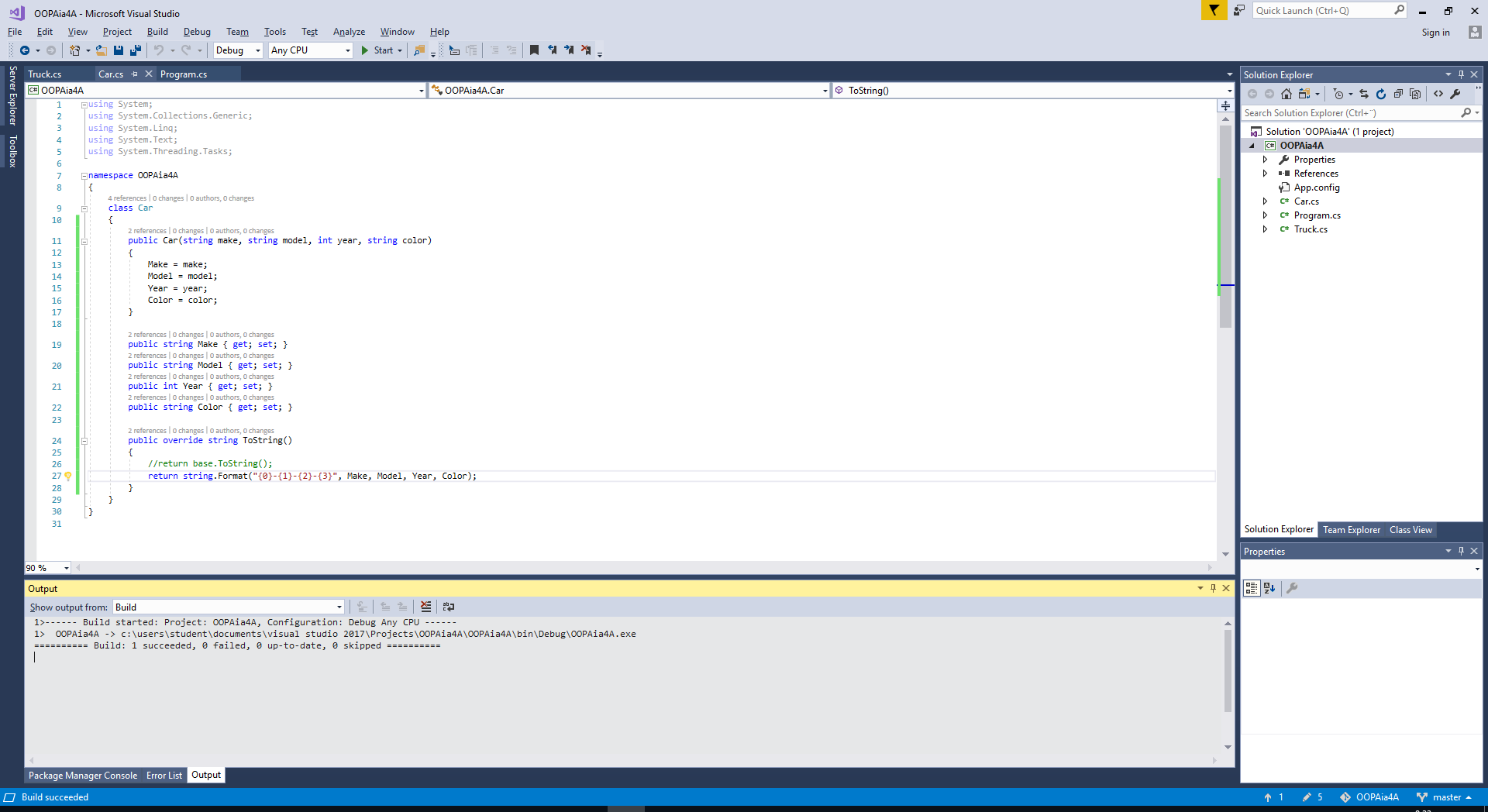
//Testing the base class:

//Car myCar1 = new Car("VW", "Jetta 1.6", 2007, "Blue");

//Overridden version of ToString():

//Console.WriteLine(myCar1);

}



class Car

{

public Car(string make, string model, int year, string color)

{

Make = make;

Model = model;

Year = year;

Color = color;

}

public string Make { get; set; }

public string Model { get; set; }

public int Year { get; set; }

public string Color { get; set; }

public override string ToString()

{

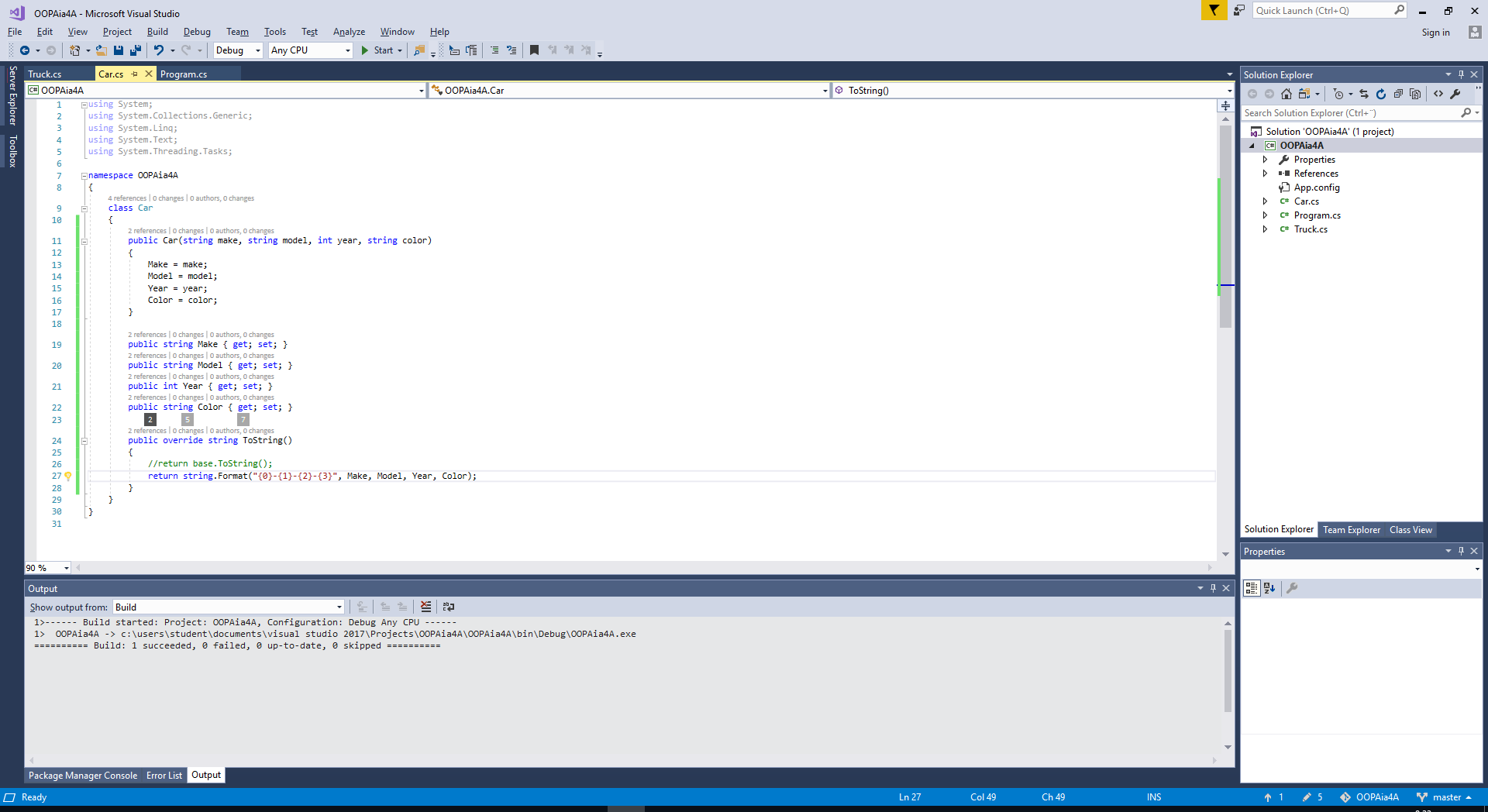
//return base.ToString();

return string.Format("{0}-{1}-{2}-{3}", Make, Model, Year, Color);

}

}

}



namespace OOPAia4A

{

class Truck:Car

{

public Truck(string make, string model, int year, string color, int towingcapacity) :

base(make, model, year, color)

{

TowingCapacity = towingcapacity;

}

public int TowingCapacity { get; set; }

public override string ToString()

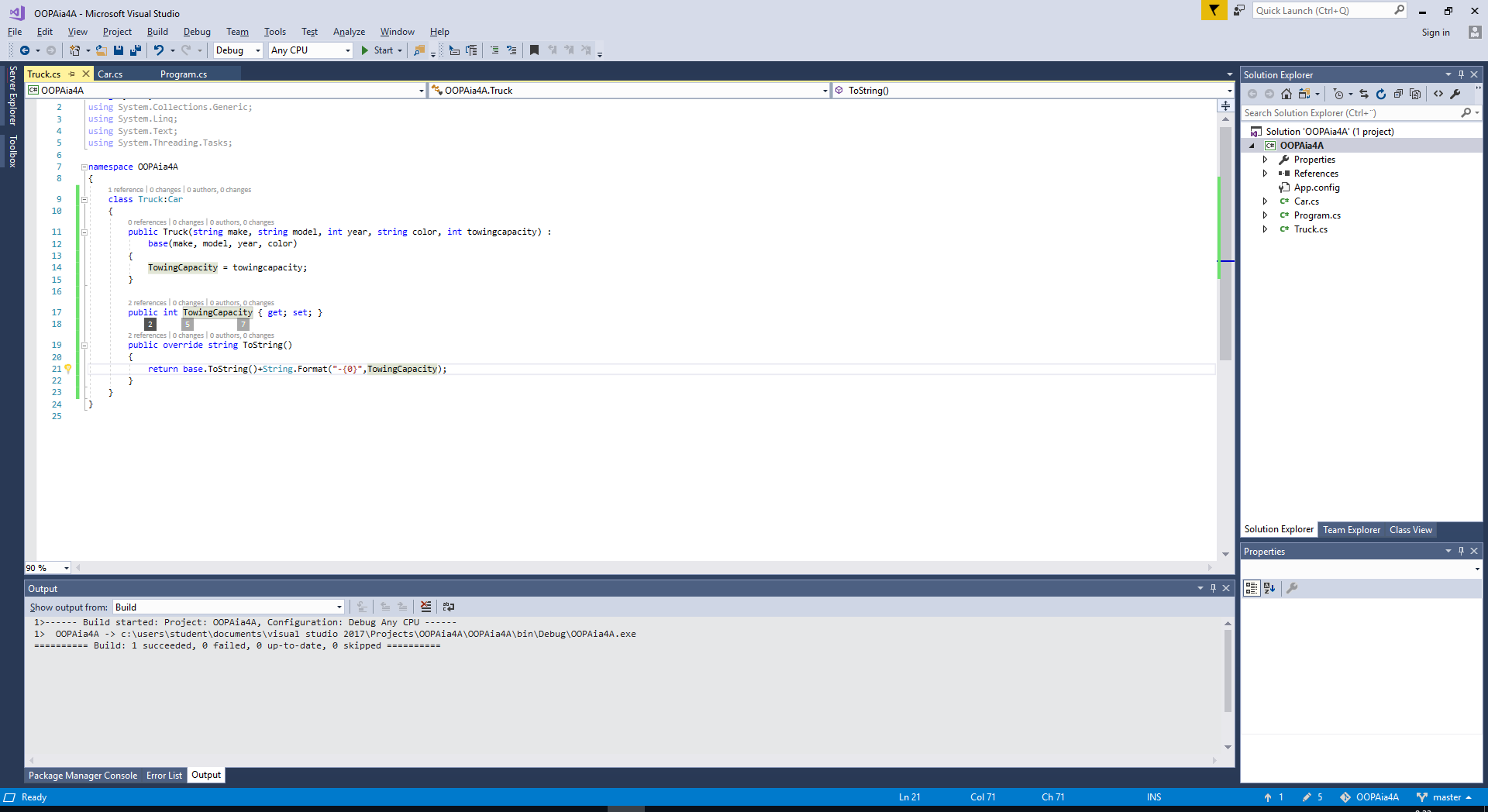
{

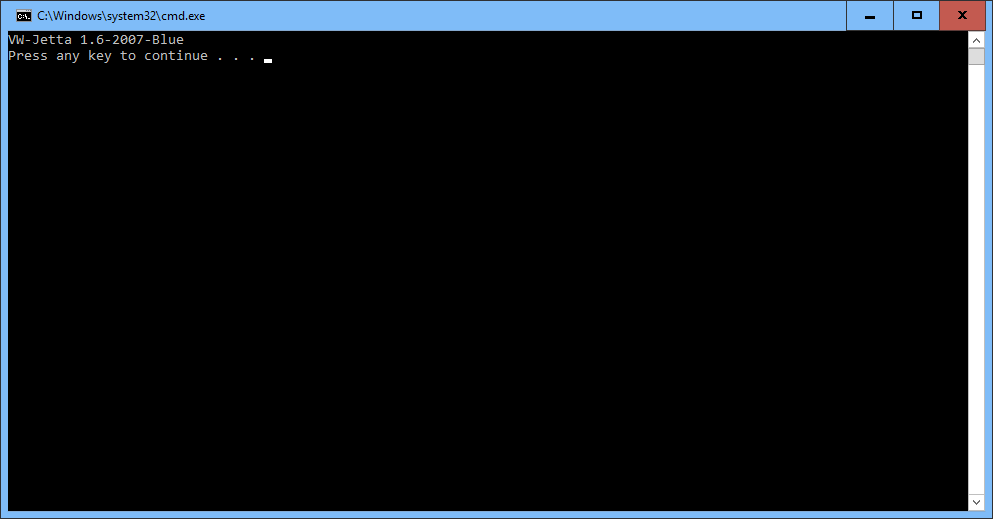
return base.ToString()+String.Format("-{0}",TowingCapacity);

}

}

}





namespace OOPAia4A

{

class Program

{

static void Main(string[] args)

{

//Testing the base class:

//Car myCar1 = new Car("VW", "Jetta 1.6", 2007, "Blue");

//Overridden version of ToString():

//Console.WriteLine(myCar1);

//Testing the derived class:

//...... and if it works, then you can modify it to be interactive....

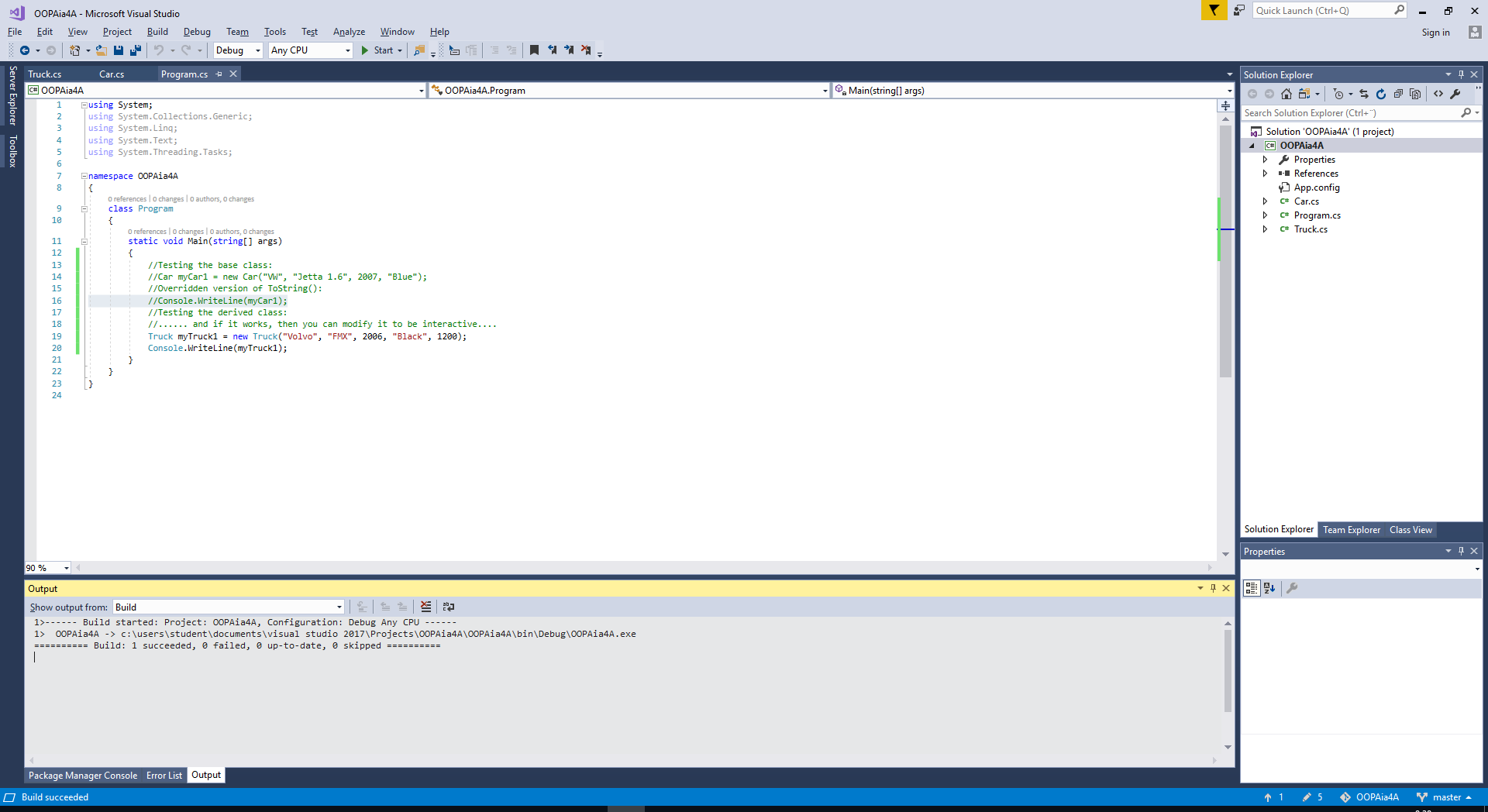
Truck myTruck1 = new Truck("Volvo", "FMX", 2006, "Black", 1200);

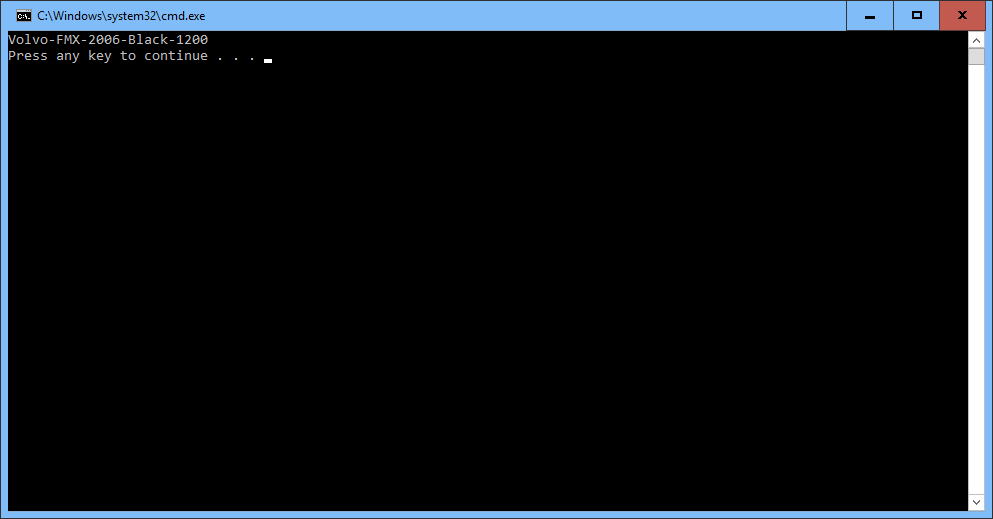
Console.WriteLine(myTruck1);

}

}

}





**Add object to the list:**

//Making it interactive....

List<Car> allCars = new List<Car>();

List<Truck> allTrucks = new List<Truck>();

bool stillMore = true;

do

{

Console.Write("Would you like to enter car or truck info? (C/T)");

String receivedInfo = Console.ReadLine();

if (receivedInfo.StartsWith("C") || receivedInfo.StartsWith("c"))

{

Console.Write("Enter the Make, Please: ");

string make = Console.ReadLine();

Console.Write("Enter the Model, Please: ");

string model = Console.ReadLine();

Console.Write("Enter the year, Please: ");

string valueReceived = Console.ReadLine();

int year = 0;

while(!Int32.TryParse(valueReceived, out year))

{

Console.Write("Not a valid value, try again: ");

valueReceived = Console.ReadLine();

}

Console.Write("Enter the color , Please: ");

string color = Console.ReadLine();

Console.Write("Enter the Towing capacity, Please: ");

valueReceived = Console.ReadLine();

int towingcapacity = 0;

while(!Int32.TryParse(valueReceived, out towingcapacity))

{

Console.WriteLine("Not a valid value, try again: ");

valueReceived = Console.ReadLine();

}

}

else if (receivedInfo.StartsWith("T") || receivedInfo.StartsWith("t"))

{

Console.Write("Enter the Make, Please: ");

string make = Console.ReadLine();

Console.Write("Enter the Model, Please: ");

string model = Console.ReadLine();

Console.Write("Enter the year, Please: ");

string valueReceived = Console.ReadLine();

int year = 0;

while (!Int32.TryParse(valueReceived, out year))

{

Console.Write("Not a valid value, try again: ");

valueReceived = Console.ReadLine();

}

Console.Write("Enter the color , Please: ");

string color = Console.ReadLine();

Console.Write("Enter the Towing capacity, Please: ");

valueReceived = Console.ReadLine();

int towingcapacity = 0;

while (!Int32.TryParse(valueReceived, out towingcapacity))

{

Console.WriteLine("Not a valid value, try again: ");

valueReceived = Console.ReadLine();

}

allTrucks.Add(new Truck(make, model, year, color, towingcapacity));

}

else

break;

//More of this or END this!?:

Console.Write("More car info to process (Y/N)?");

String moreOfThis = Console.ReadLine();

if (moreOfThis.StartsWith("Y") || moreOfThis.StartsWith("y"))

stillMore = true;

else

stillMore = false;

} while (stillMore);

//Output everything you got:

//Find here more sophistucated way of finding out if your list instances are empty...

bool isEmpty1 = !allCars.Any();

bool isEmpty2 = !allTrucks.Any();

if (isEmpty1 && isEmpty2)

{

Console.WriteLine("You entered nothing.");

}

else

{

Console.WriteLine("You got these:");

foreach (Car myCars in allCars)

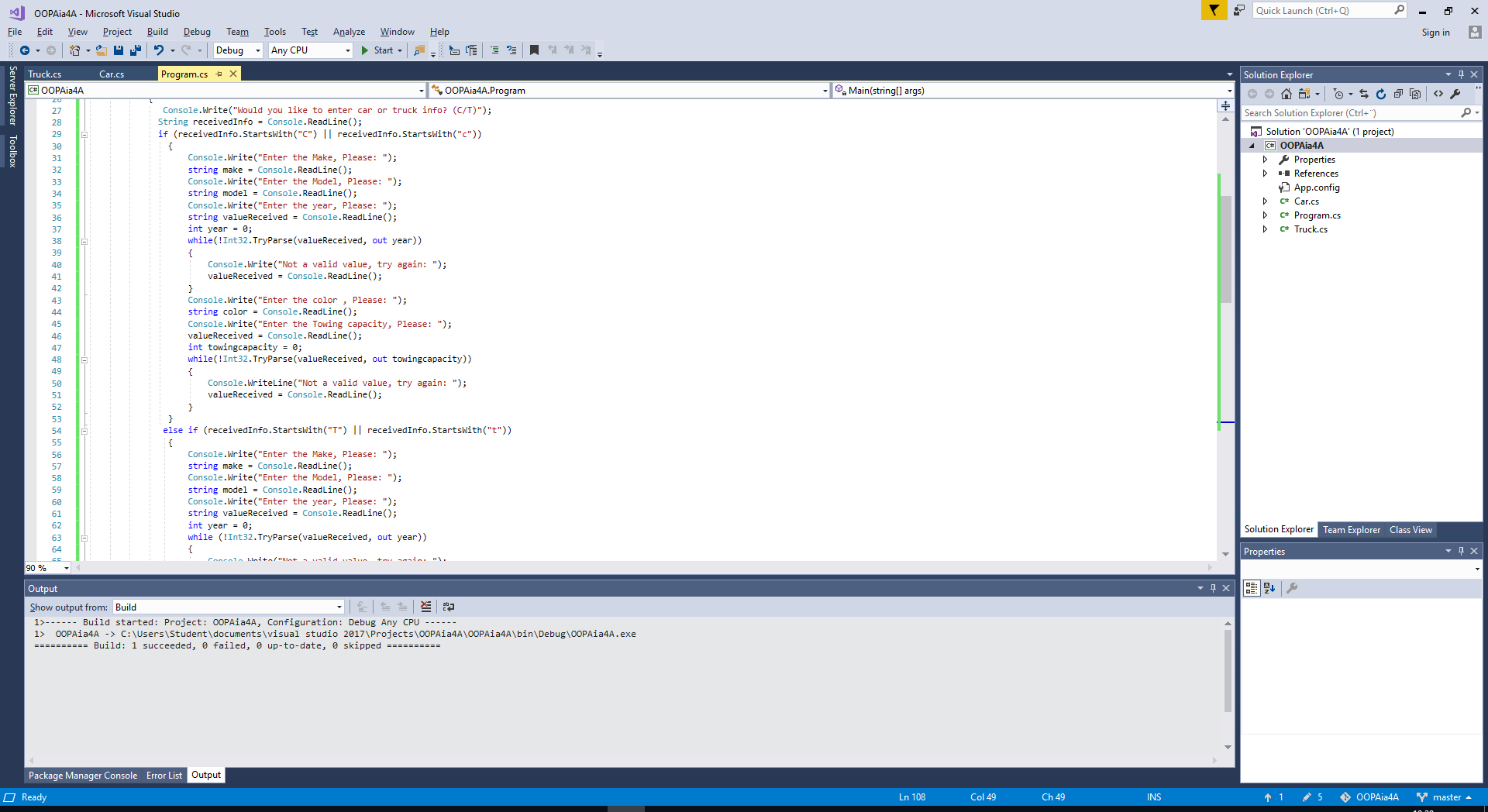
Console.WriteLine(myCars);

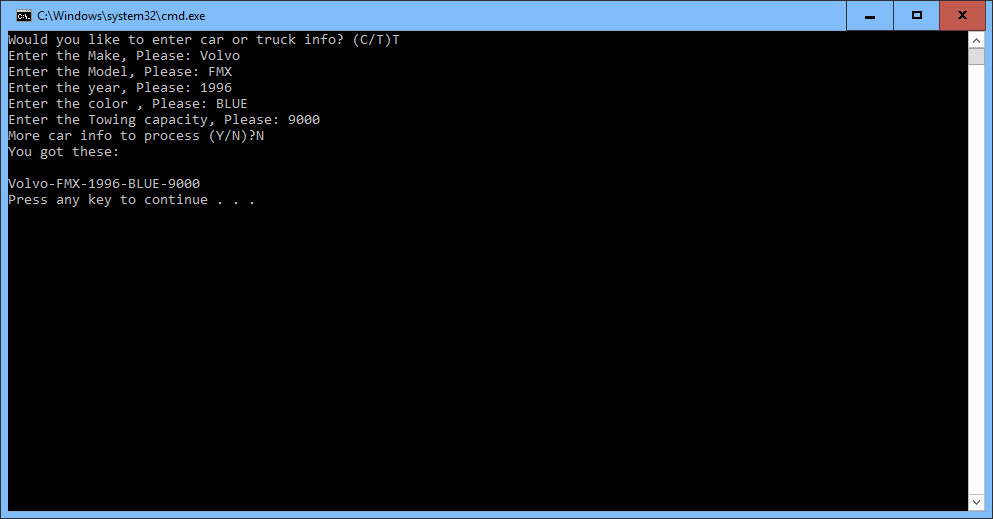
Console.WriteLine();

foreach (Car myTrucks in allTrucks)

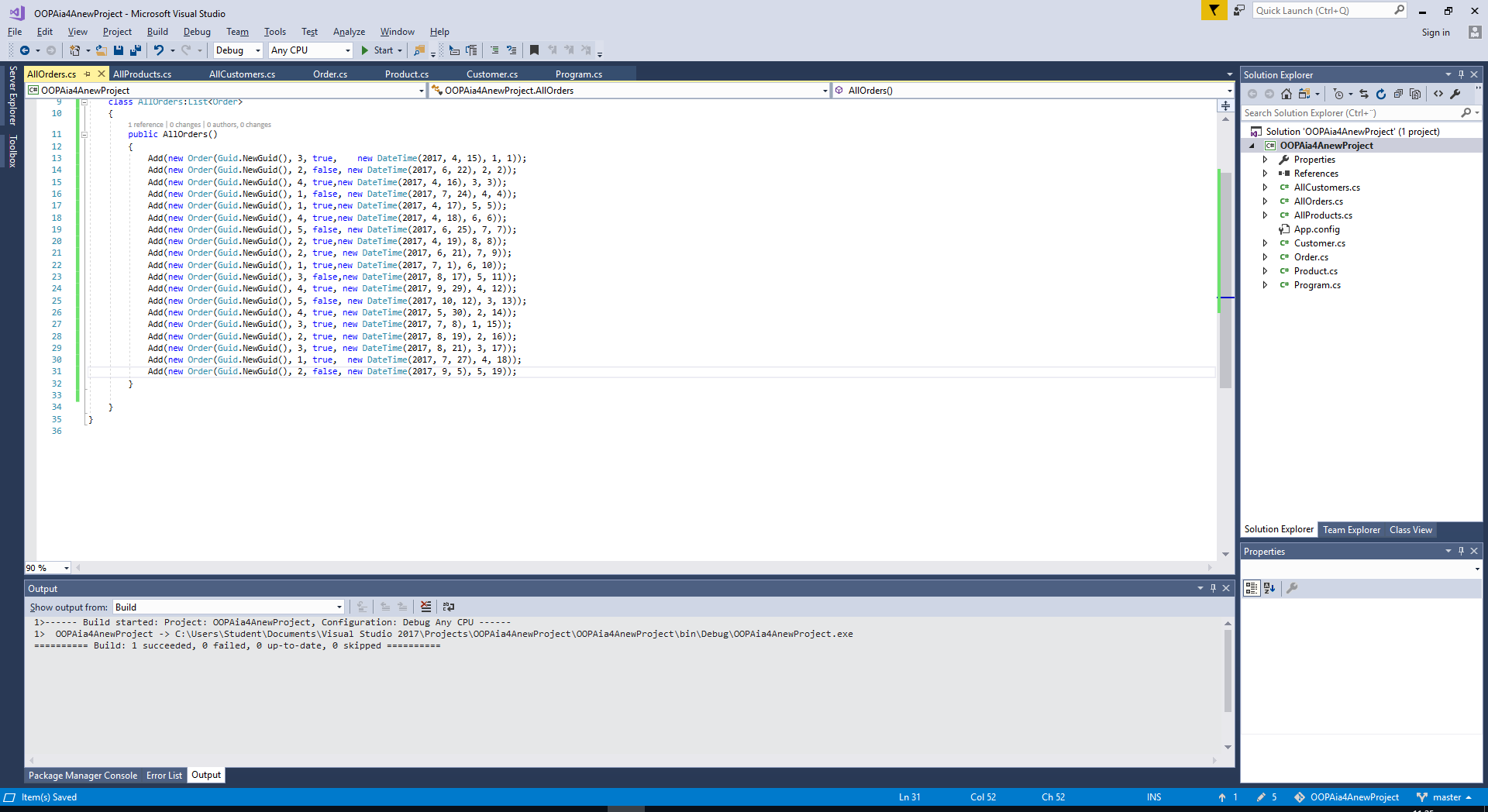
Console.WriteLine(myTrucks);

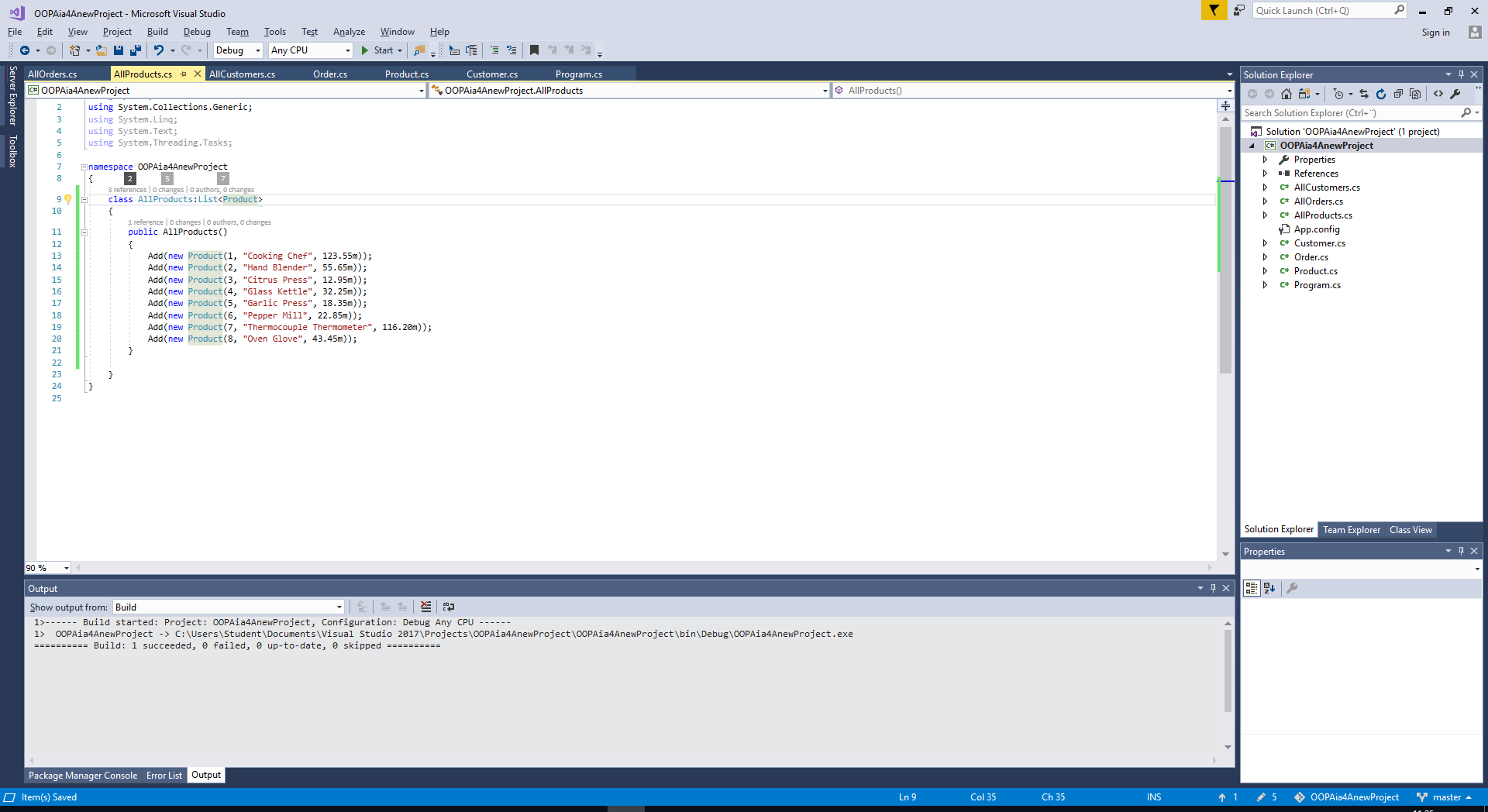
}

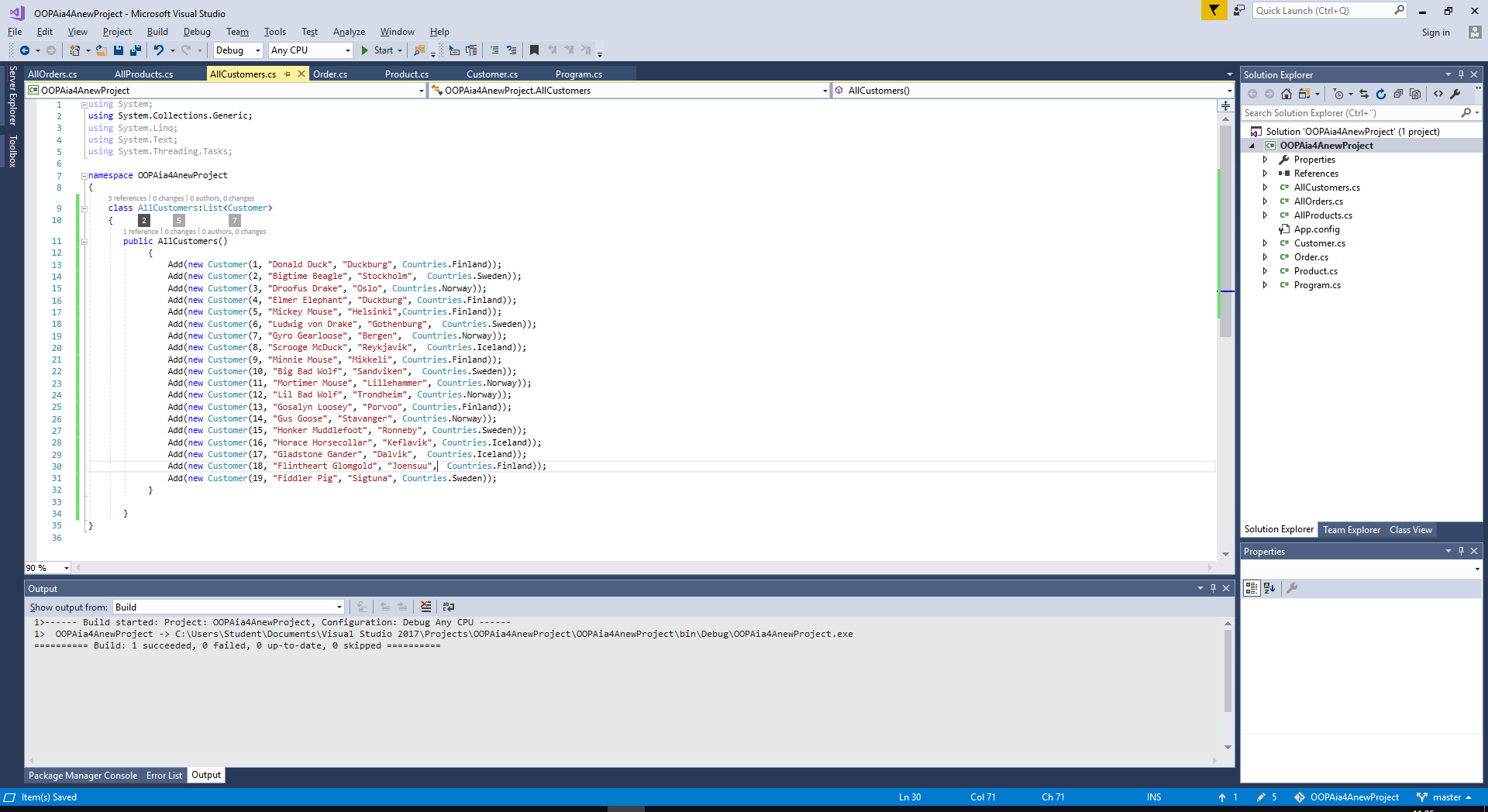


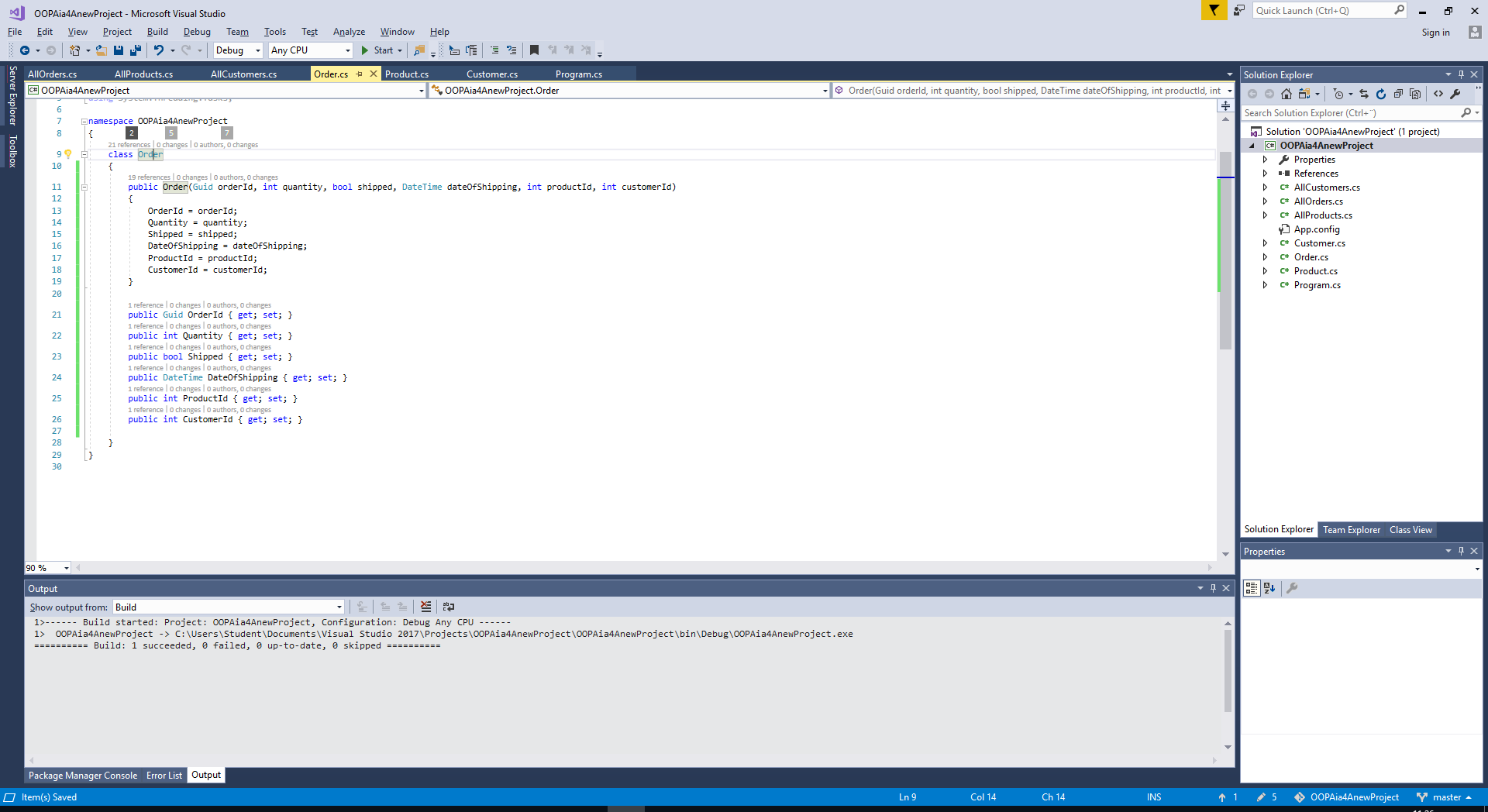


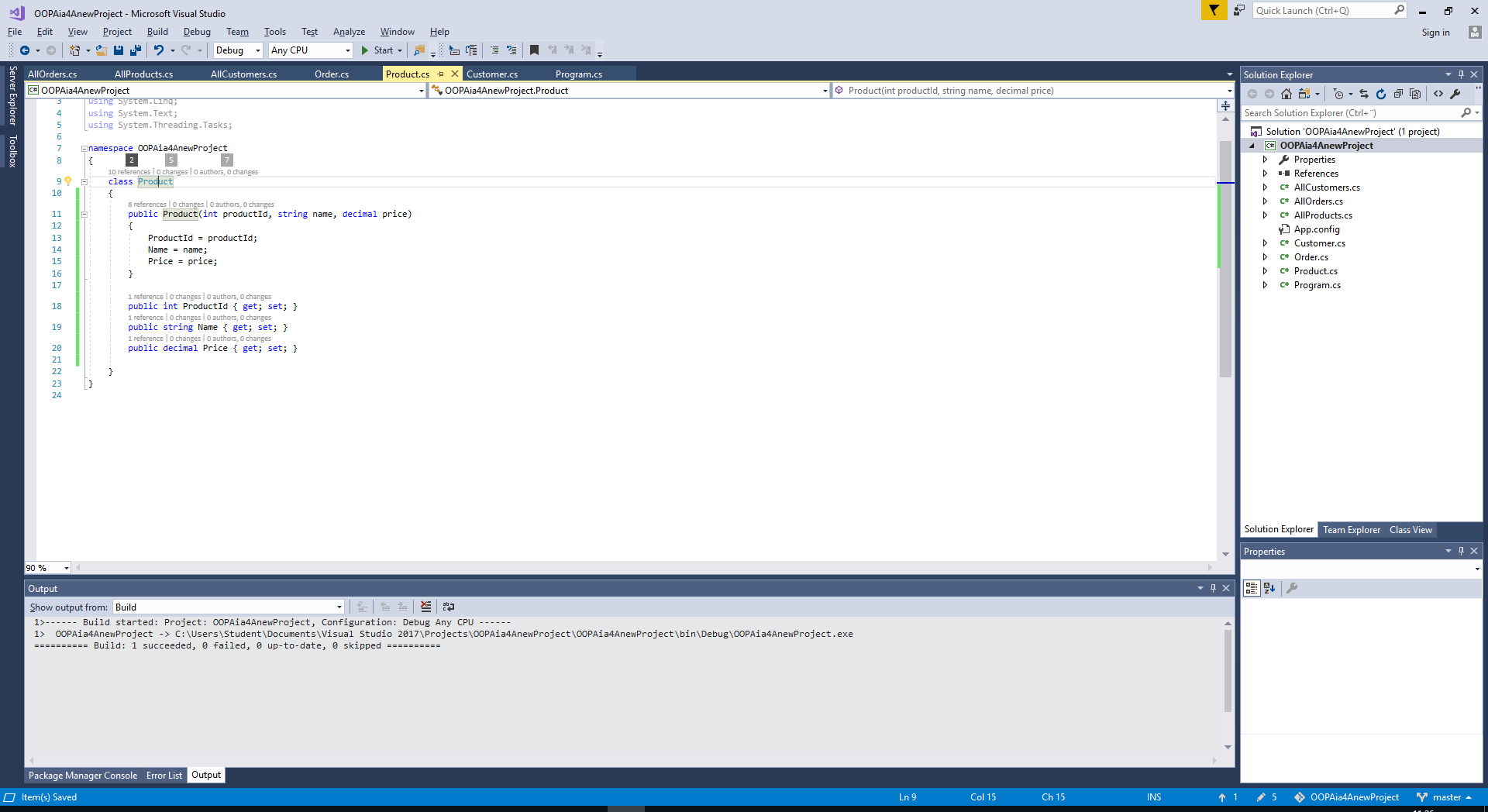
**2.Using inheritance when initializing list instances. You can expect to have the following class definitions:**

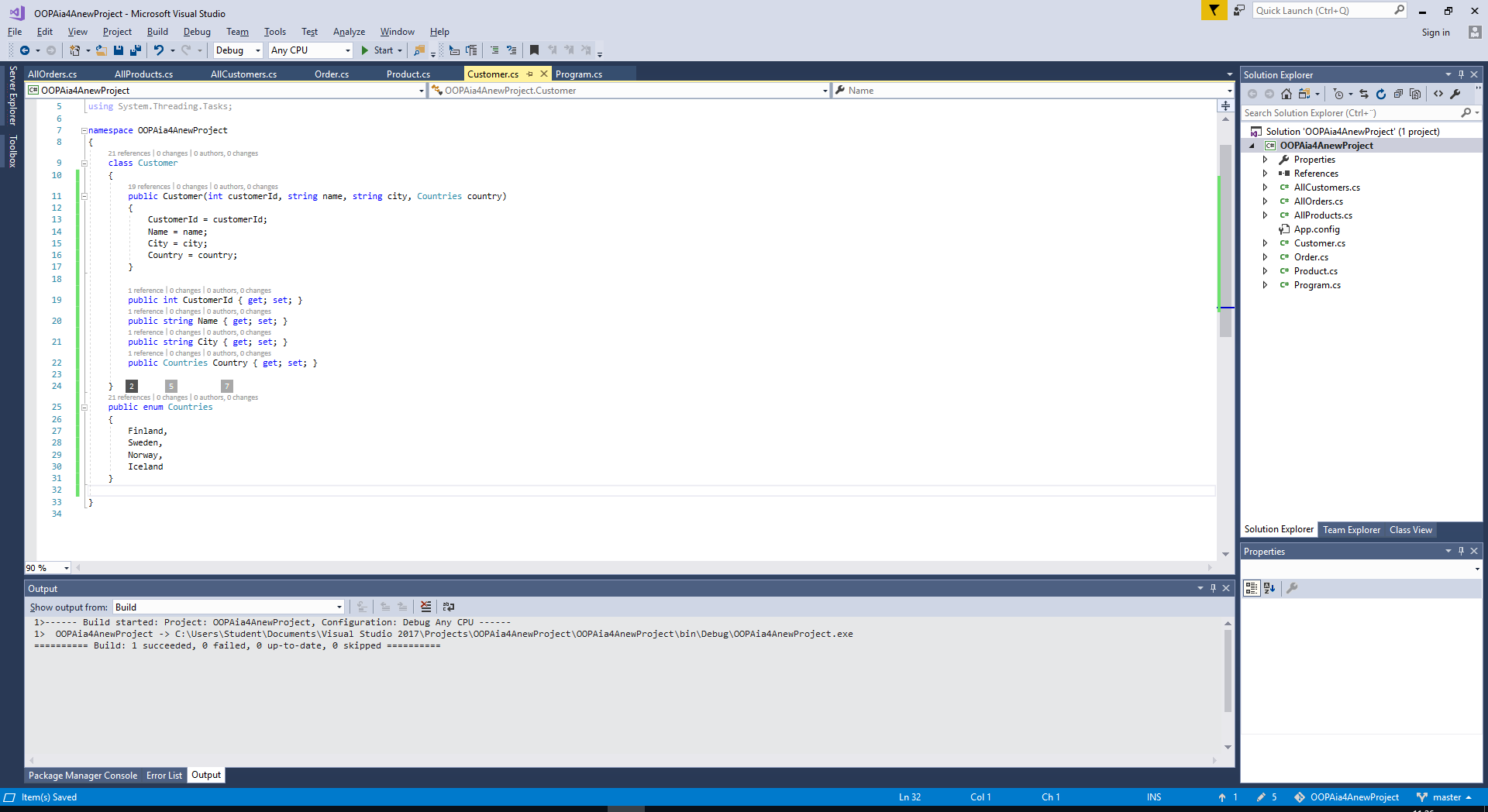


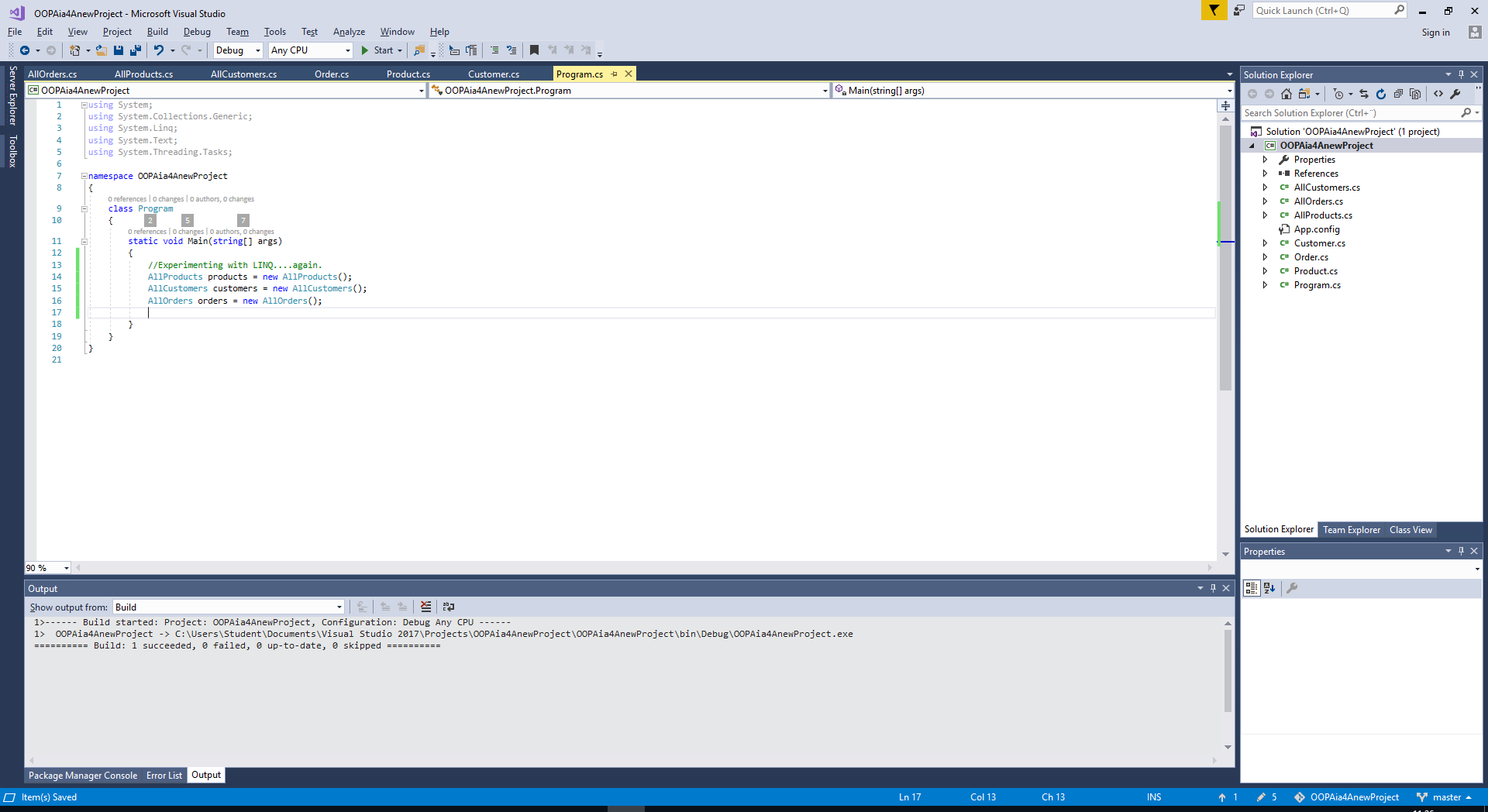












namespace OOPAia4AnewProject

{

class Program

{

static void Main(string[] args)

{

//Experimenting with LINQ....again.

AllProducts products = new AllProducts();

AllCustomers customers = new AllCustomers();

AllOrders orders = new AllOrders();

var customersInFinland= from cust in customers

where cust.Country == Countries.Finland

select cust;

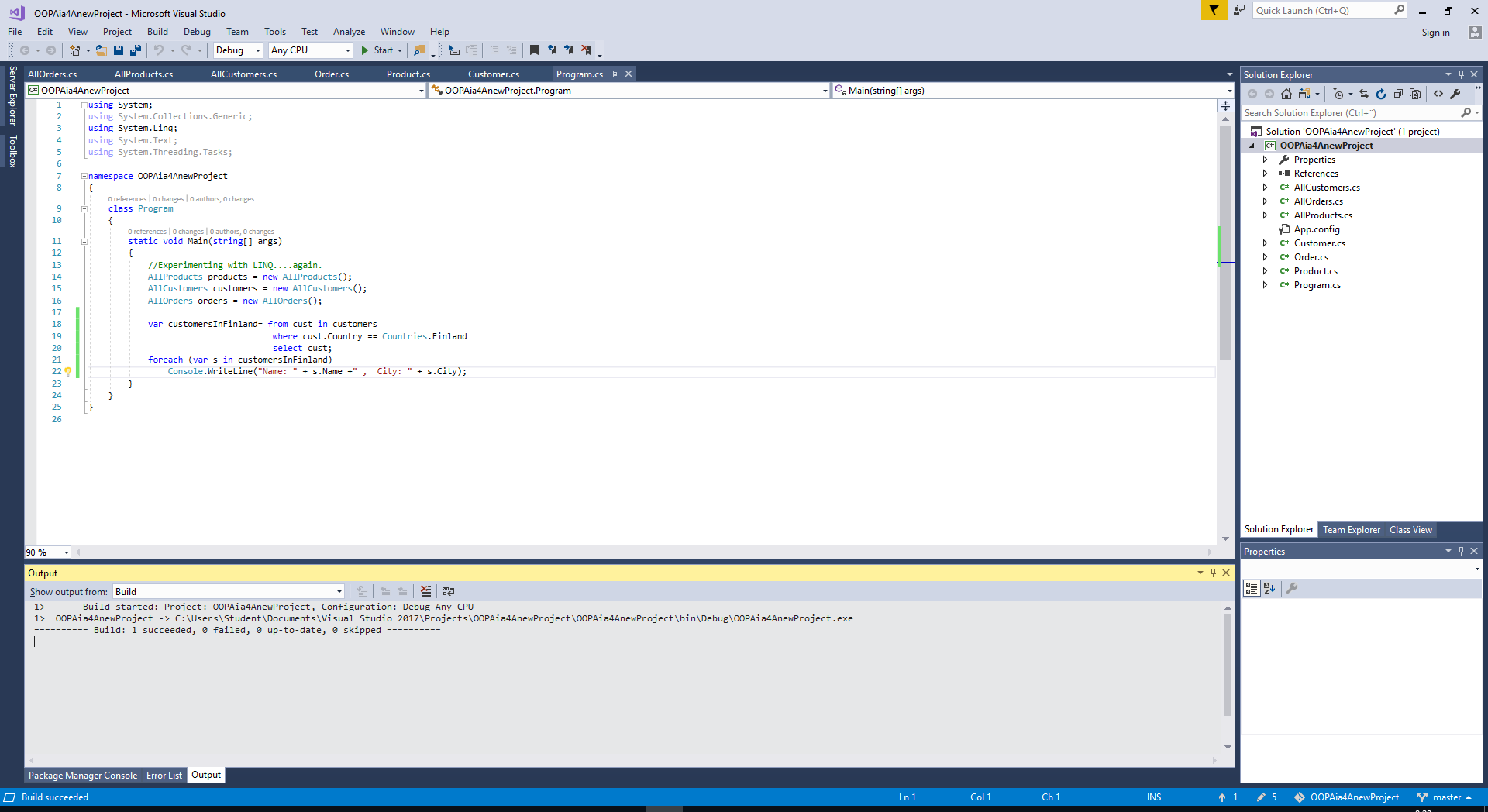
foreach (var s in customersInFinland)

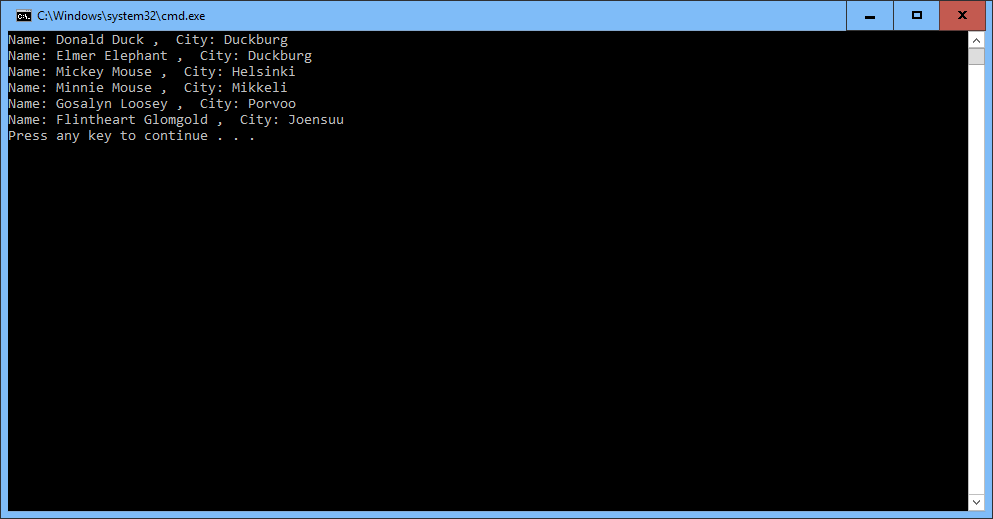
Console.WriteLine("Name: " + s.Name +" , City: " + s.City);

}

}

}





DateTime threshold =DateTime.Parse("20.6.2017");

var newestOrders = from ord in orders

where ord.DateOfShipping > threshold

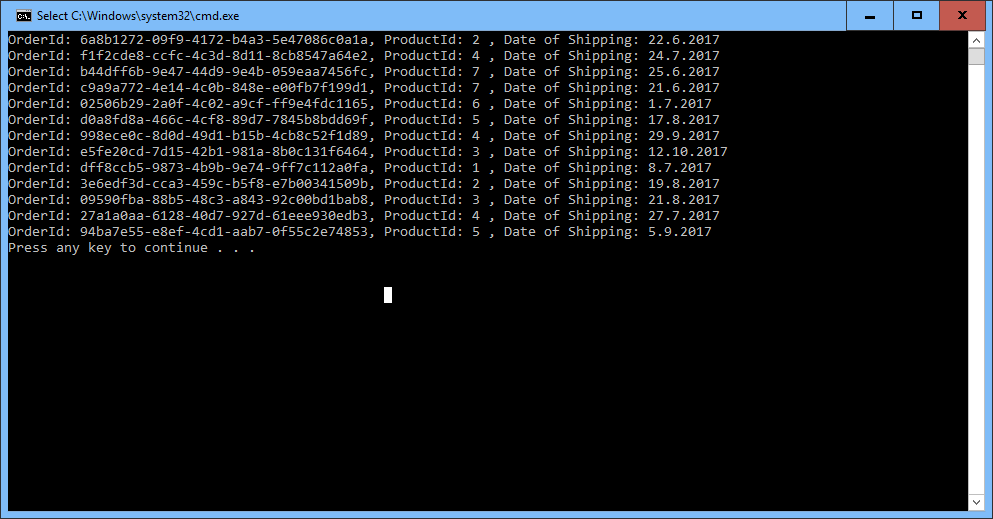
select ord;

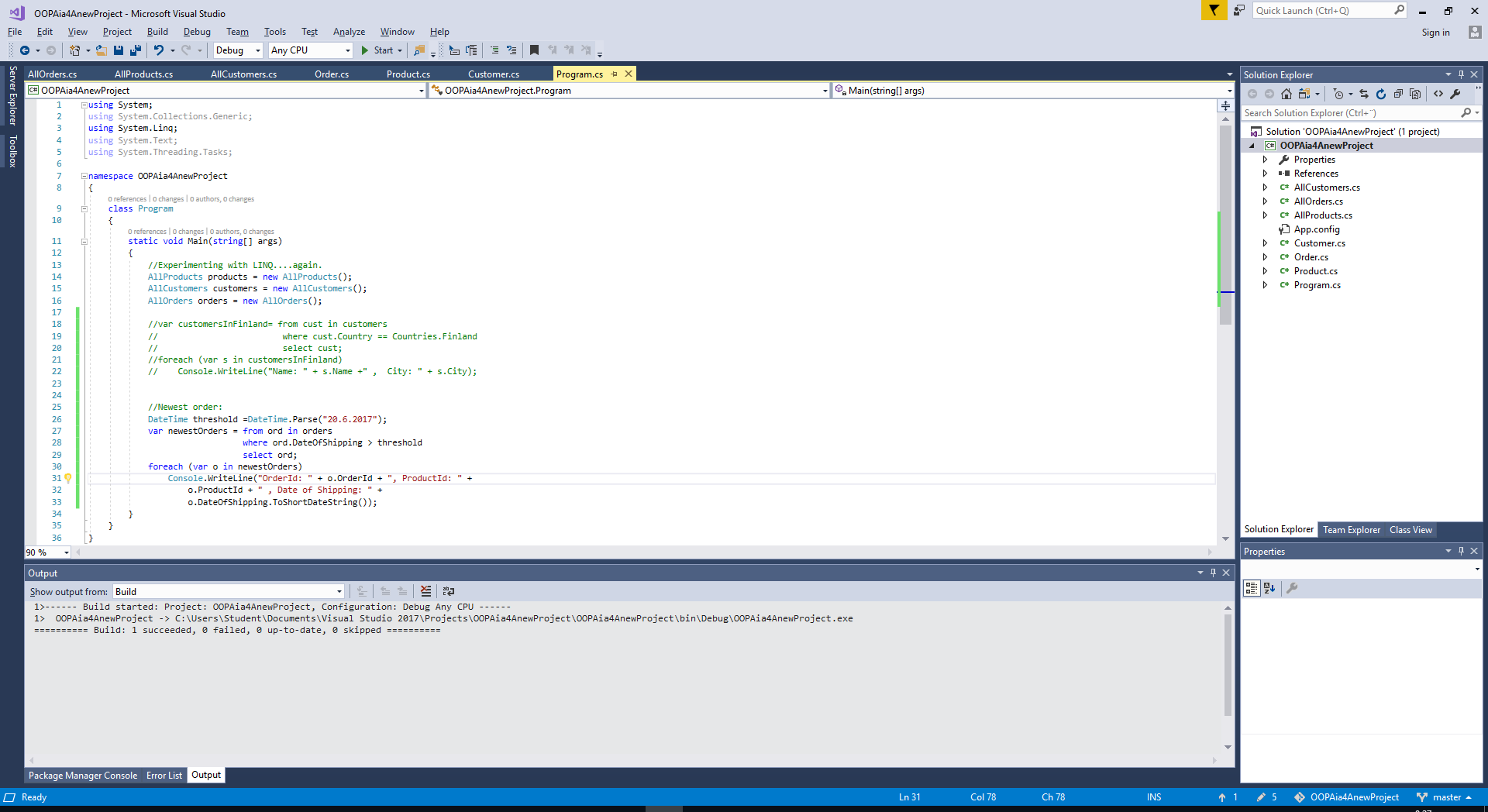
foreach (var o in newestOrders)

Console.WriteLine("OrderId: " + o.OrderId + ", ProductId: " +

o.ProductId + " , Date of Shipping: " +

o.DateOfShipping.ToShortDateString());





var customersInFinlandA = from Cust in customers

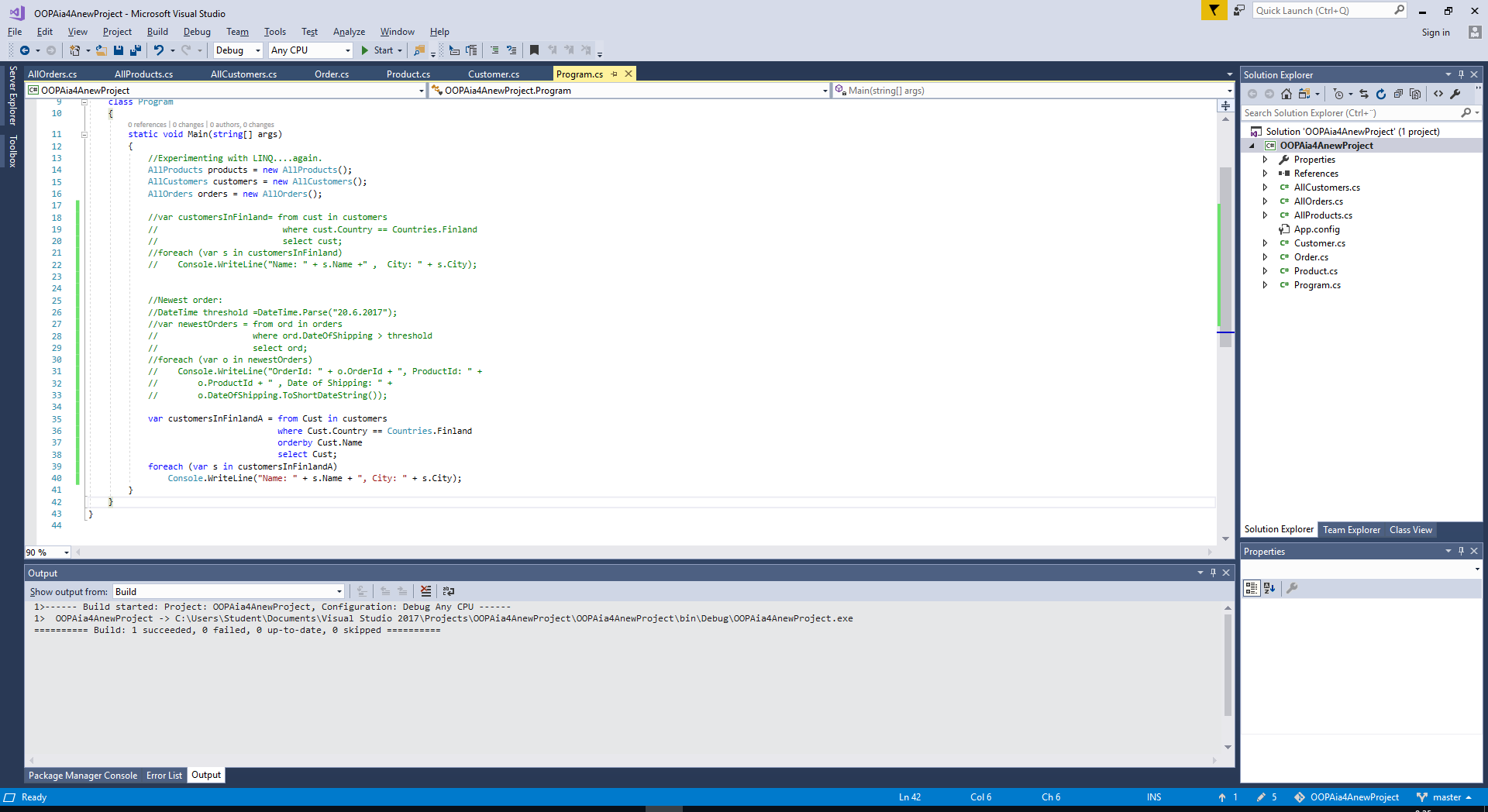
where Cust.Country == Countries.Finland

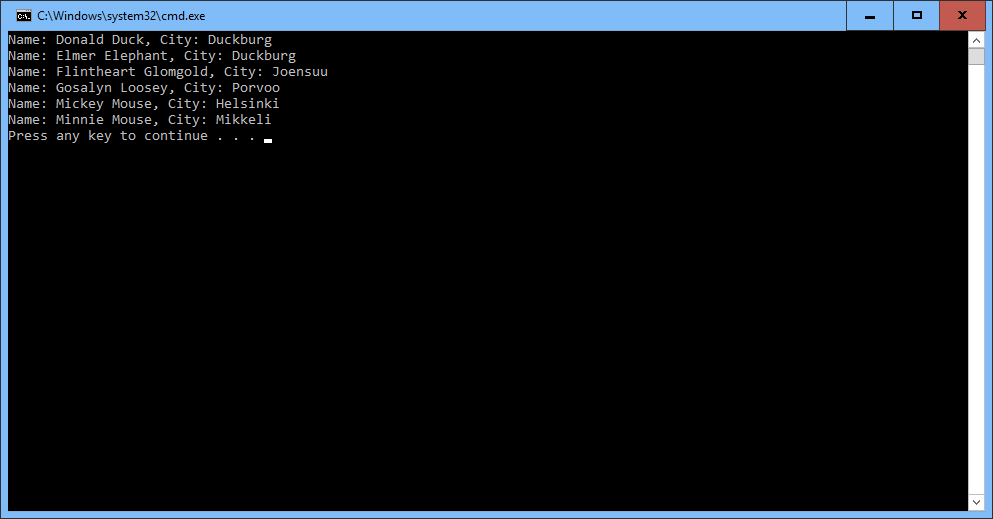
orderby Cust.Name

select Cust;

foreach (var s in customersInFinlandA)

Console.WriteLine("Name: " + s.Name + ", City: " + s.City);





string selectionCriterion = "Mouse";

var certainCuostomers = from cust in customers

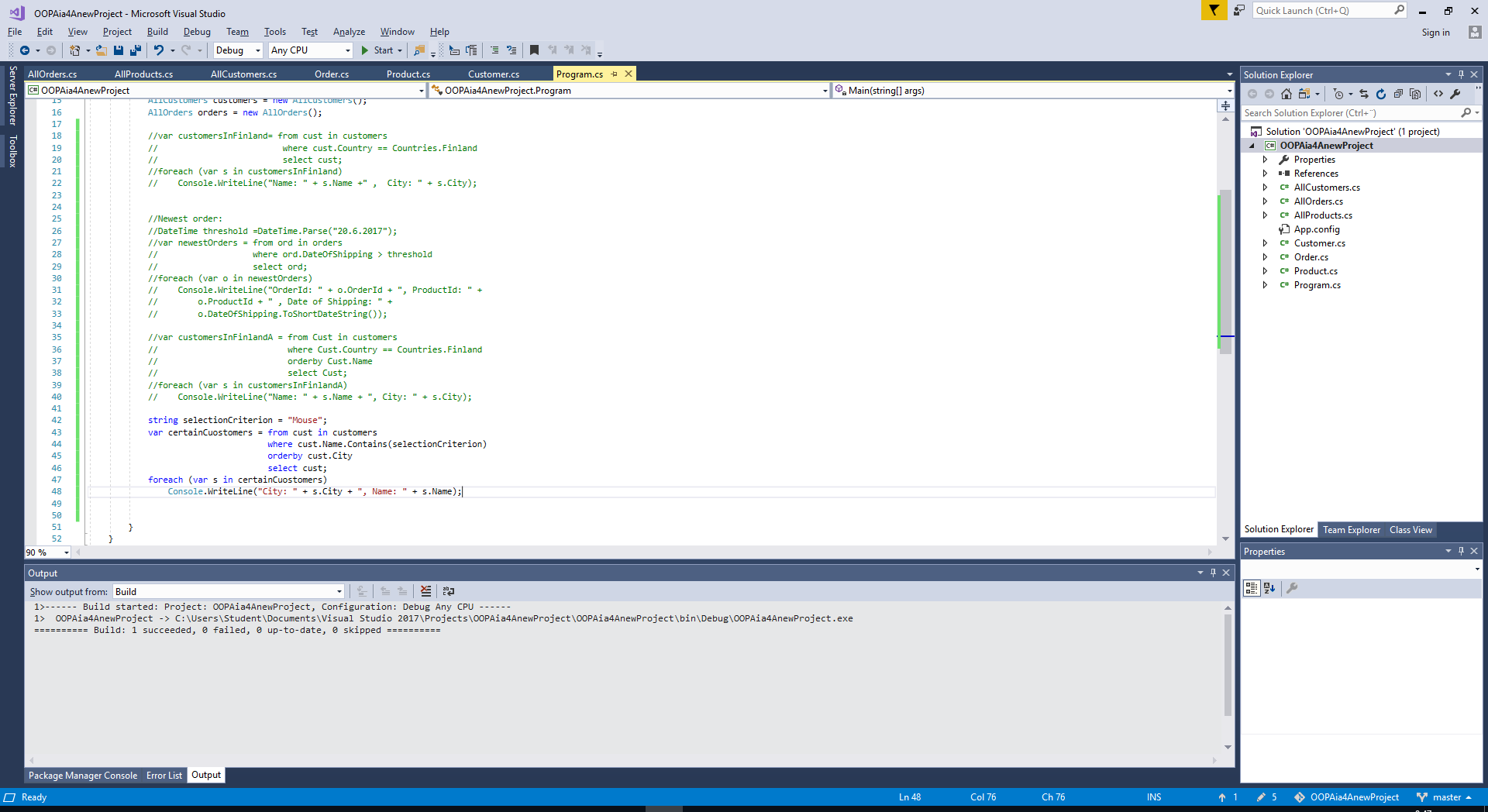
where cust.Name.Contains(selectionCriterion)

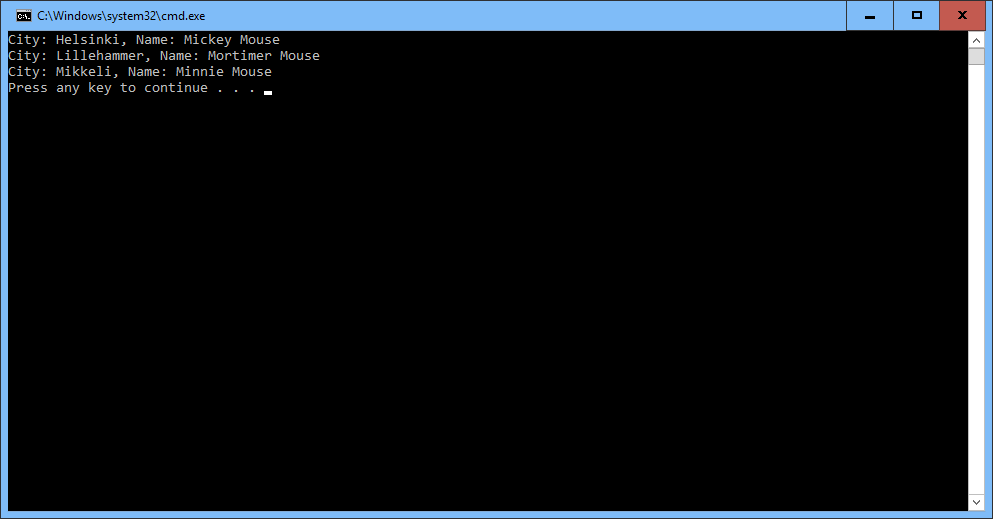
orderby cust.City

select cust;

foreach (var s in certainCuostomers)

Console.WriteLine("City: " + s.City + ", Name: " + s.Name);





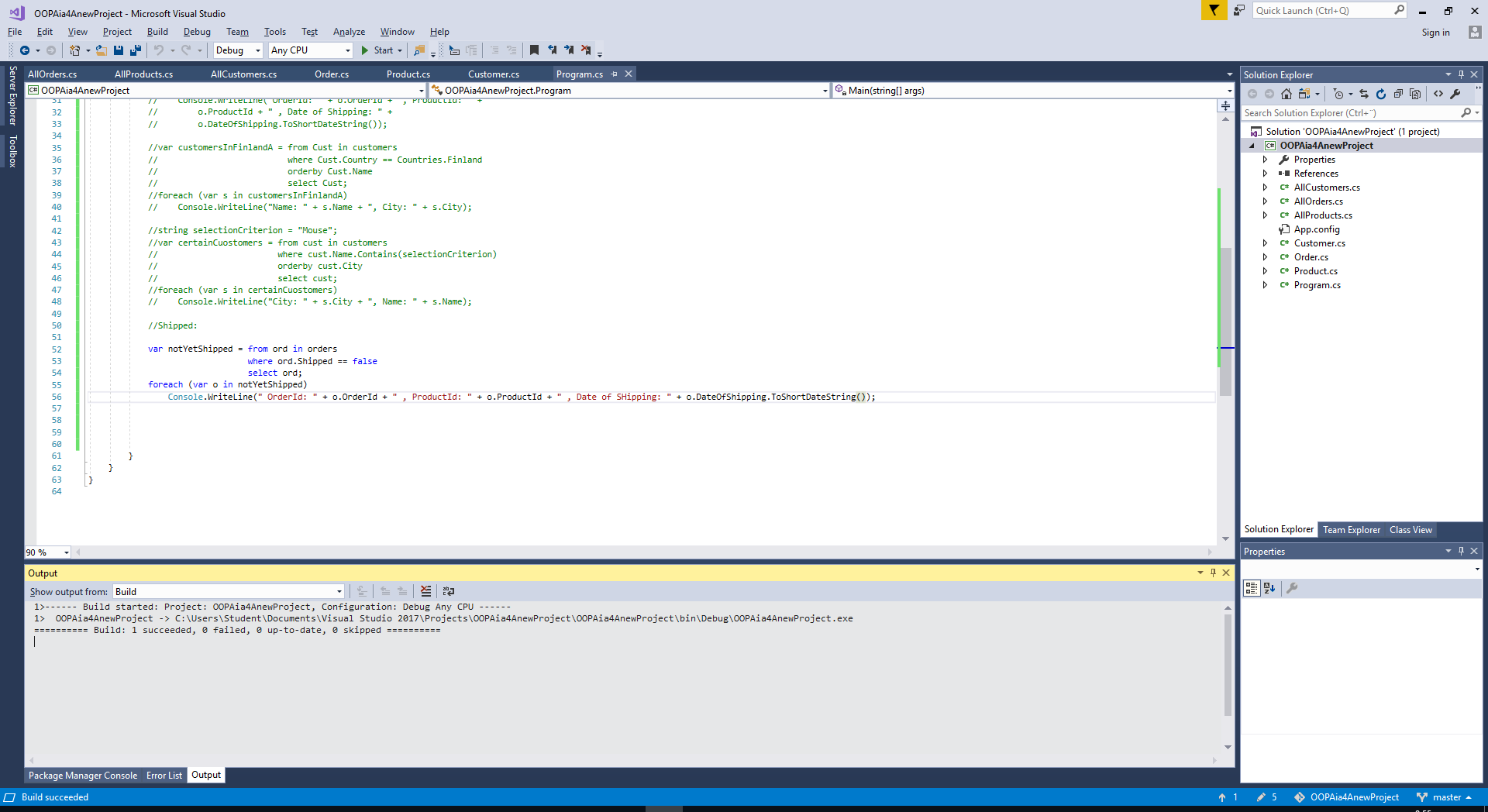
var notYetShipped = from ord in orders

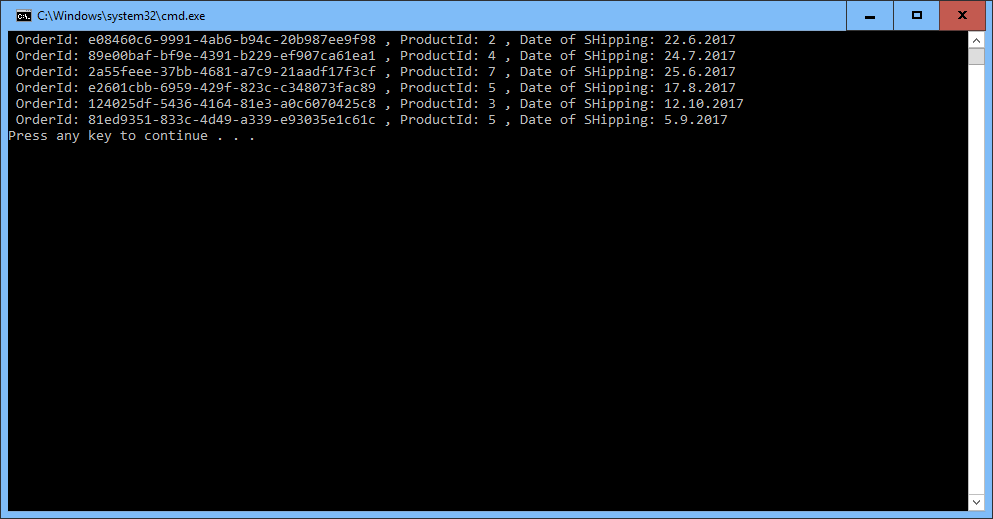
where ord.Shipped == false

select ord;

foreach (var o in notYetShipped)

Console.WriteLine(" OrderId: " + o.OrderId + " , ProductId: " + o.ProductId + " , Date of SHipping: " + o.DateOfShipping.ToShortDateString());





3. Abstract classes. Here you experiment with abstract base class:

class Circle:GeometricShape

{

//Constructor:

public Circle(string colorFilled, DateTime dateWhenCreated, double radius) :

base(colorFilled, dateWhenCreated)

{

Radius = radius;

}

//Properties:

public double Radius { get; set; }

public override double GetArea()

{

return Math.PI \* Radius \* Radius;

//throw new NotImplementedException();

}

public override double GetPerimeter()

{

return 2 \* Math.PI \* Radius;

//throw new NotImplementedException();

}

public override string ToString()

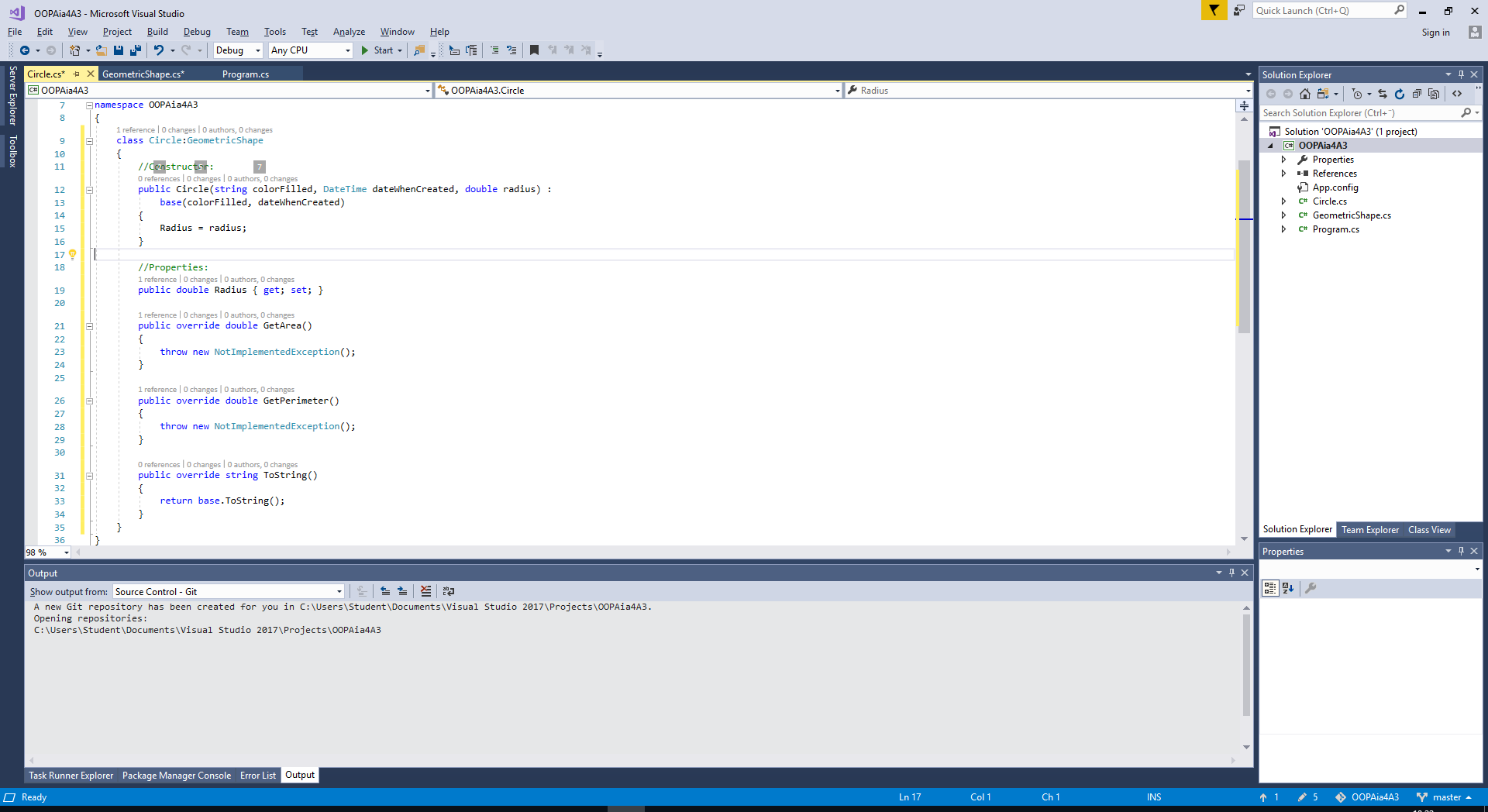
{

return base.ToString();

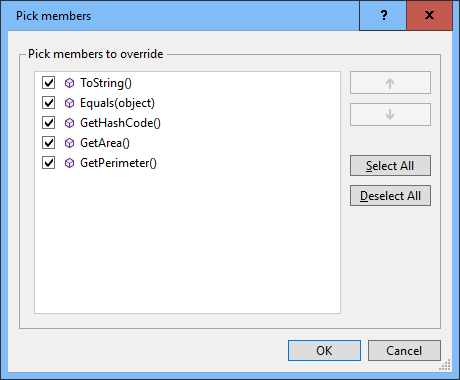
}

}

}



**public override**



class Rectangle:GeometricShape

{

//Constructor:

public Rectangle(string colorFilled, DateTime dateWhenCreated, double length, double width) :

base(colorFilled, dateWhenCreated)

{

Length = length;

Width = Width;

}

//Properties:

public double Length { get; set; }

public double Width { get; set; }

public override double GetArea()

{

return Length \* Width;

//throw new NotImplementedException();

}

public override double GetPerimeter()

{

return 2 \* Length + 2 \* Width;

//throw new NotImplementedException();

}

public override string ToString()

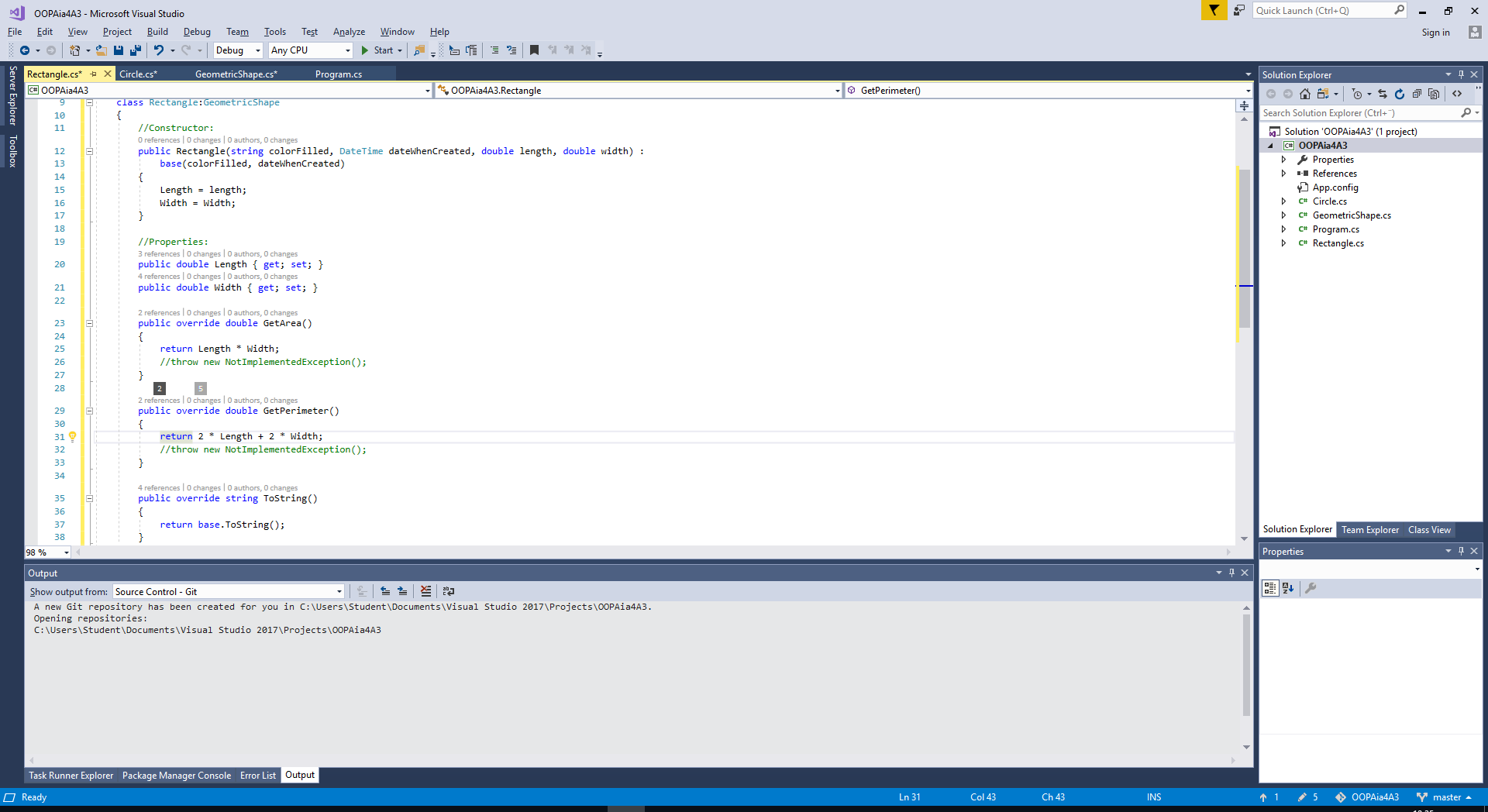
{

return base.ToString();

}

}

}



namespace OOPAia4A3

{

class Program

{

static void Main(string[] args)

{

bool stillMore = false;

int yourChoice = -1;

double radius = -1;

string colorOfTheShape;

do

{

Console.WriteLine("Geometry Calculator\n");

Console.WriteLine("\t1.Calcutate with Circle");

Console.WriteLine("\t2.Calcutate with Rectangle");

Console.Write("\tEnter your Choise (1 or 2) : ");

string receivedValue = Console.ReadLine();

while(!Int32.TryParse(receivedValue, out yourChoice)||

yourChoice<1 || yourChoice > 2)

{

Console.Write("\tNot accepted, try again: ");

receivedValue = Console.ReadLine();

}

switch (yourChoice)

{

case 1:

Console.Write("Enter the radius (double): ");

receivedValue = Console.ReadLine();

while (!Double.TryParse(receivedValue, out radius) ||

radius <0)

{

Console.Write("\tNot accepted, try again: ");

receivedValue = Console.ReadLine();

}

Console.Write("Enter the color: ");

colorOfTheShape = Console.ReadLine();

Circle myCircle = new Circle(colorOfTheShape, DateTime.Now, radius);

Console.WriteLine("Area is {0}.", myCircle.GetArea());

Console.WriteLine("Perimeter is {0}.", myCircle.GetPerimeter());

break;

}

Console.Write("More of this (Y/N): ");

string moreOfThis = Console.ReadLine();

if(moreOfThis.StartsWith("Y") || moreOfThis.StartsWith("y"))

stillMore = true;

else

stillMore = false;

} while (stillMore);

}

}

}

