Microsoft Corporation

Axum Programmer's Guide

A simple and easy to follow programming guide to learn how to create safe, scalable, and responsive application with the Axum language.

Contents

Introduction

Why Another Language?

Writing a parallel program typically requires partitioning the solution into a number of parallel tasks. Some problems are easily amenable to parallelization because the tasks can run independently of each other. In other problems the tasks have interdependencies and require coordination.

Axum helps you with:

- ✓ Coordinating concurrent components
- ✓ Writing responsive applications in the face of latency.

For example, ray tracing, a method of generating an image by tracing light's path yields itself to a parallel solution because each ray can be implemented as an independent task – processing of one ray has no effect on the processing of another ray.

On the other hand, in a gameplay simulation, the game is modeled using parallel but interacting objects. The state and behavior of an object have impact on the objects around it.

With Axum, we offer a language that allows programmers to arrange coordination between components in a way that is close to their natural conception of the solution. In other words, if you can model your solution in terms of interactive components, encoding it in Axum will be straightforward, and you will likely avoid many common concurrency-related bugs.

An obvious motivation for writing a parallel program is to make it *run faster*. Closely related to that is a desire to make the program *do more* while it's running. This is especially important for interactive applications that must process user input while performing a background task.

Very often, responsiveness of interactive applications is hindered by long-latency components such as I/O or user input. For example, an email client must wait for the data from the server, which might be behind a slow network. It is desirable that such an application remains responsive while requesting data from the server.

One of the goals of Axum is to let you program without worrying about concurrency – your program becomes fast and responsive by default, not as a result of an afterthought or a retrofit.

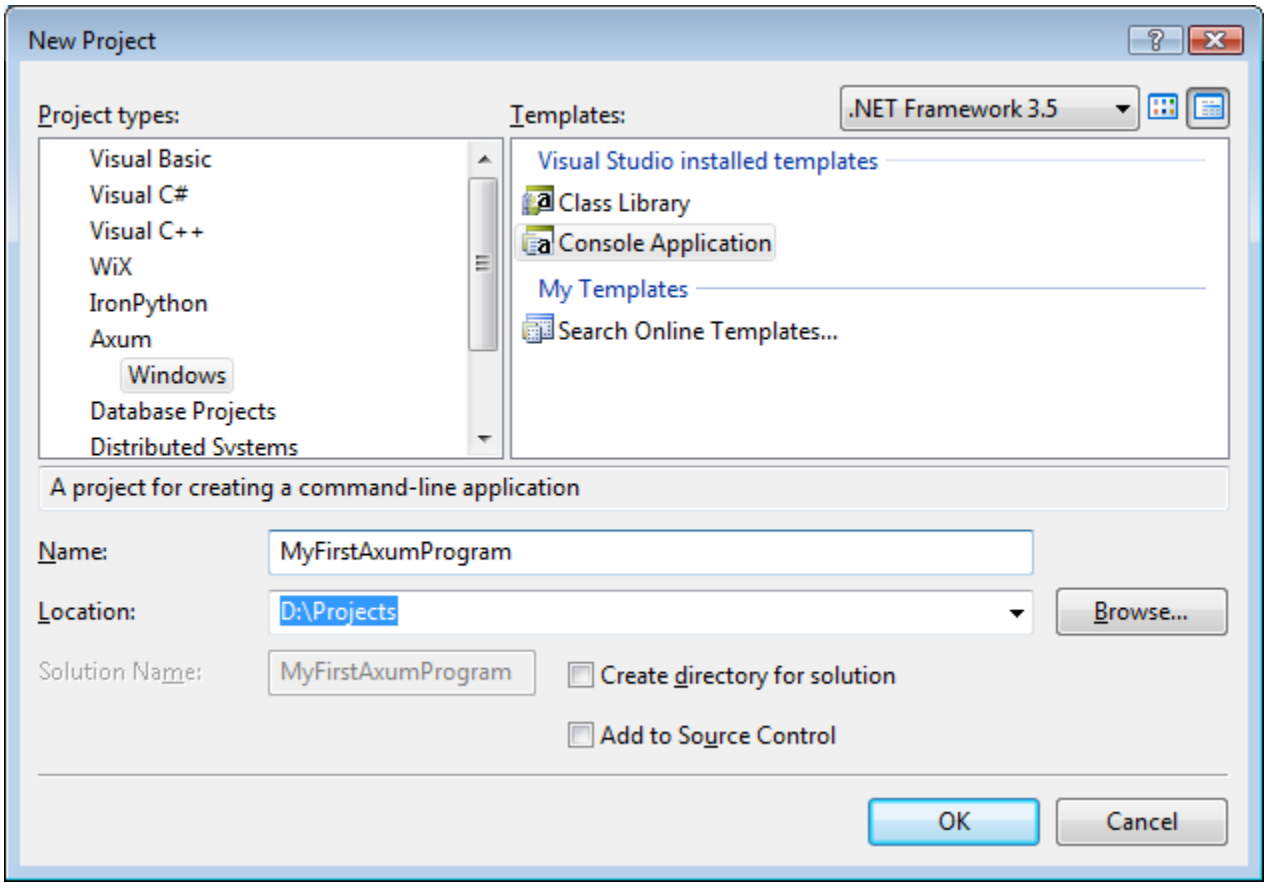
In addition to enabling the new capabilities for working with concurrency, Axum takes away one capability that historically has proven to cause problems – that is unrestricted ability to share and mutate state from different threads. Axum isolation model ensures "disciplined" access to shared state that prevents many common programming errors.

Finally, please remember that Axum is an experiment. We want to make it better and we need to know what you think. We will appreciate your comments about the language, and how you think you can use it for building your own software. Please share your thoughts, comments and suggestions with us and your fellow Axum users via MSDN forums at http://social.msdn.microsoft.com/Forums/en/axum.

The Basics

Hello, World!

To get started with Axum, launch Visual Studio and select *File* | *New* | *Project* from the menu. In the dialog that comes up, select Axum project type, and choose a name for your application:



When you click OK, Visual Studio will generate some boilerplate code that is useful to create a new Axum application. To keep things simple, and to follow the time-honored tradition, our first Axum program will do nothing but print “Hello, World” on the console.

Replace the generated code with the following:

```
using System;

agent Program : Microsoft.Axum.ConsoleApplication
{
    override int Run(String[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Let's look at this code a little closer.

The program starts with the keyword *agent*. The concept of an agent in Axum derives from what is known in computer science as the "actor model". In this model, actors represent autonomous entities that communicate with each other via messages, act on the data they receive from other actors, spawn off other actors and so on.

Axum program defines agents and their interactions

In Axum, actors are represented by agents. Writing a program in Axum is all about defining agents and arranging interaction between them.

Agent-based programming is different from object-oriented programming in many important ways. First of all, unlike objects, agents do not provide public methods or exhibit their state. You cannot "reach into" an agent and modify any of its fields. You cannot call a method on an agent and wait for it complete. Instead, you can send it a message, and arrange for the agent to "get back to you" with a response.

Axum comes with a supporting class library that includes an agent called *ConsoleApplication*. This agent implements the necessary workings of a console application – the startup, setting up the command line parameters and shutdown.

Our sample above makes use of *ConsoleApplication* by deriving an agent from it. When you derive from *ConsoleApplication*, you will need to override the *Main* method and place your application logic there – as it is done in our sample.

Axum is a .NET language and can use libraries written in other .NET language such as C#, Visual Basic or F#.

Being a .NET language, Axum can naturally use libraries written in any other .NET language such as C#, VB.Net or F#. In our example, we're calling the *WriteLine* of the *System.Console* class from the .NET base class library (BCL)

Message-Passing

To make the previous example a little more interesting, we will introduce the concept of *channels* and implement an agent that sends a message to the channel's *port*.

Earlier we said that agents are components that perform actions on data – that data normally comes in and goes out of the agent via a channel. To accommodate different types of data, a channel has one or more ports.

Later when we talk about agents in more detail, we will see how to define a channel – but first, let's look at how to use a channel to send and receive messages.

An example:

```
using System;
agent Program : channel Microsoft.Axum.Application
{
    public Program()
    {
        // Receive command line arguments from port CommandLine
        String [] args = receive(PrimaryChannel::CommandLine);

        // Send a message to port ExitCode:
        PrimaryChannel::ExitCode <-- 0;
    }
}
```

Here we have an agent called *Program* that implements a channel *Microsoft.Axum.Application*.

Implementing a channel is different – syntactically and semantically – from deriving from a base agent. When an agent derives from another agent, it merely extends it by overriding some virtual methods, and potentially adding more of its own.

Channels have two ends: the implementing end and the using end

However, when an agent implements a channel (notice the *channel* keyword after the colon in the agent declaration), it "attaches" itself to the *implementing* end of that channel and becomes the "server" of messages on that channel. The other end of the channel – known as the *using* end – is only visible to the "client", or the component (typically another agent) on the other end of the channel. Figure 1 shows this:

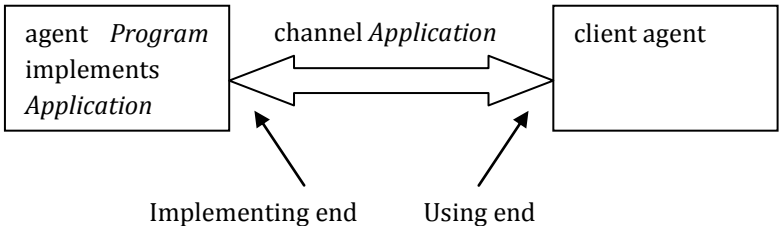


Figure 1: Two ends of a channel

Channels exist to transmit messages between agents. Channels define what kind of data can go into and out of them, but unlike agents, they don't perform any transformation of that data.

Returning to our example, the using end of *Application* channel is implemented in the Axum runtime. The runtime instantiates the agent implementing channel *Microsoft.Axum.Application,*

sends command line parameters to the channel's *CommandLine* port, and then waits for a message on port *ExitCode.* When the message is received, the application shuts down.

The agent *Program* waits for a message to arrive on its *CommandLine* port (the *receive* statement), and then signals completion by sending a message to port *ExitCode* (operator <--). The agent Program has a built-in property *PrimaryChannel* to access the channel being implemented – sometime we will also call it "agent's primary channel". The double-colon "::" is used to access the channel's port.

Notice the phrase "waits for a message" used above. Receiving a message is a blocking operation in Axum – meaning that *receive* statement that attempts to read from an empty port stalls until a message arrives to that port. On the other hand, send is asynchronous – the sender of the message doesn't wait for it to arrive to the destination.

Asynchronous Programming with Messages

With messages being the main means of communication between agents, we need some systematic ways of dealing with them. As a whole, we refer to this as *orchestration*. Axum offers two distinct approaches to orchestration: control-flow-based and data-flow-based orchestration. Often, the two are combined for ultimate expressiveness and power.

In Axum, the messages are sent to and received from the *interaction points*. An interaction point from which a message originates is called the *source*, and the destination is called the *target*. An interaction point can be both a source and a target, meaning that it can both send and receive messages. This allows composition of multiple interaction points into *dataflow networks*.

Independent nodes in a dataflow network can execute concurrently

Simply put, a dataflow network is a messaging construct that receives data, does something with it – in other words, performs a *transformation* – and produces a result. Using a dataflow network can be advantageous if some nodes of the network are independent of each other and therefore can execute concurrently.

Unlike control-flow logic that is based on conditional statements, loops, and method calls, data-flow networks base their logic on forwarding, filtering, broadcasting, load-balancing, and joining messages that pass through the network. It's a different and complementary approach to handling messages.

Let's look at an example of a dataflow network that calculates multiple Fibonacci numbers:

```
using System;
using Microsoft.Axum;
using System.Concurrency.Messaging;

agent MainAgent : channel Microsoft.Axum.Application
{
    function int Fibonacci(int n)
    {
```

```
        if( n<=1 ) return n;
        return Fibonacci(n-1) + Fibonacci(n-2);
    }

    int numCount = 10;

    void ProcessResult(int n)
    {
        Console.WriteLine(n);
        if( --numCount == 0 )
            PrimaryChannel::ExitCode <-- 0;
    }

    public MainAgent()
    {
        var numbers = new OrderedInteractionPoint<int>();

        // Create pipeline:
        numbers ==> Fibonacci ==> ProcessResult;

        // Send messages to numbers:
        for( int i=0; i<numCount; i++ )
            numbers <-- 42-i;
    }
}
```

Here we've defined a method *Fibonacci* with the keyword *function*. In Axum, a function is a method that does not modify any state outside of itself – in other words, it leaves no *side effects* of its execution. For instance, if you try to modify member *numCount* or send a message in the Fibonacci function, the compiler will issue an error.

Now, let's take a look at the constructor of *MainAgent*. The very first statement creates an instance of *OrderedInteractionPoint<int>*, which is an interaction point that acts as both a source and a target. The word "ordered" means that the order of messages is preserved – the messages are queued up in the order they arrive, and leave in the same order.

Next, the agent sets up the dataflow network using the *forwarding* operator ==>. The statement:

```
numbers ==> Fibonacci ==> PrintResult;
```

should be understood as "whenever a message arrives at interaction point *numbers*, forward it to a transformation interaction point implemented by function *Fibonacci*, then forward the result to the method *PrintResult*."

A pipeline is the simplest form of a dataflow network.

It turns out that dataflow networks that forward messages from one node to another are quite common, and have a name – *pipelines*.

The fact that *Fibonacci* is a side-effect-free function allows Axum runtime to execute many transformations in parallel, spawning off as many threads as it deems necessary for the most efficient execution of the program.

In Axum, functions are often used to accomplish parallel execution of nodes in pipelines and other types of networks.

It's important to note that the even though the nodes of the pipeline can execute in parallel, the order of the messages in the pipeline is preserved. In other words, *PrintResult* will receive the results in the same order that the corresponding inputs have entered the *numbers* interaction point.

Programming with Agents

We saw on the Fibonacci example above how to build a trivial dataflow network. Such networks work well for simple "data comes in, data goes out" scenarios, but they don't specify exactly how the data travels through the network, and don't allow different types of data to come in or go out of the network.

It turns out that agents and channels give us just what we need to build sophisticated dataflow networks.

Channels and Ports

The two agents communicating over a channel are decoupled from each other: one doesn't know or care how the other one is implemented. The "contract" between them is specified by the channel only. To borrow an analogy from the OOP, the channel acts as an interface, and the agent as the class implementing the interface.

> Channels are to agents what interfaces are to classes

When using a channel, you send data into the *input* ports, and receive data from the *output* ports. That is, input ports act as targets, and output ports as sources.

When implementing a channel, the input ports are seen as sources and the output ports as targets.

If this duality sounds confusing, consider a mailbox with incoming and outgoing slots. The postman delivers the incoming mail into the incoming slot and takes out the outgoing mail from the outgoing slot. The resident, on the other hand, deposits the mail into the outgoing slot and takes out the incoming mail from the incoming slot. In other words, the postman sees the incoming slot as a source and outgoing slot as a target; the resident has the opposite view of the slots.

Let's illustrate the idea with an example. Consider a channel *Adder* that takes two numbers and produces the sum of these numbers. The user of the channel sends the numbers to the input ports *Num1* and *Num2*, and receives the result from the output port *Sum*. The channel is used by agent *MainAgent* and is implemented by agent *AdderAgent* (see Figure 2).

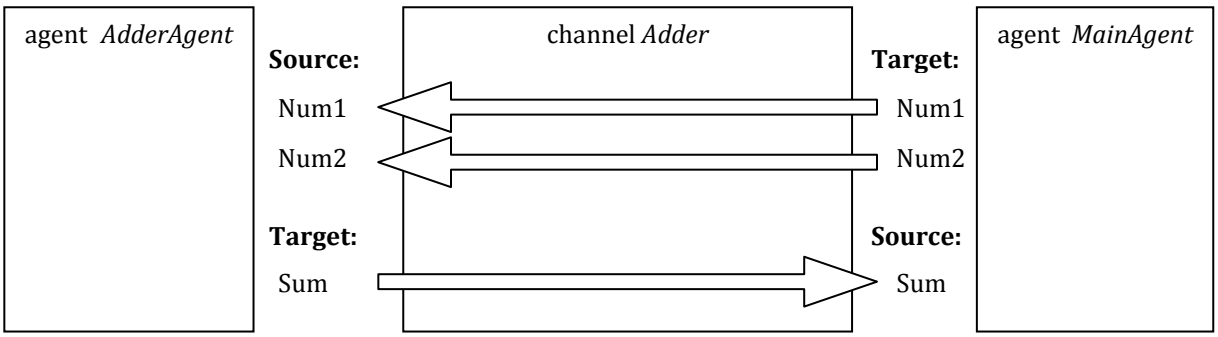


Figure 2: Duality of ports

Here is the implementation and use of the *Adder* channel:

```
using System;
using System.Concurrency;
using Microsoft.Axum;

channel Adder
{
    input int Num1;
    input int Num2;
    output int Sum;
}

agent AdderAgent : channel Adder
{
    public AdderAgent()
    {
        int result = receive(PrimaryChannel::Num1) +
            receive(PrimaryChannel::Num2);
        PrimaryChannel::Sum <-- result;
    }
}

agent MainAgent : channel Microsoft.Axum.Application
{
    public MainAgent()
    {
        var adder = AdderAgent.CreateInNewDomain();
        adder::Num1 <-- 10;
        adder::Num2 <-- 20;
        // do something useful ...
        var sum = receive(adder::Sum);

        Console.WriteLine(sum);

        PrimaryChannel::ExitCode <-- 0;
    }
}
```

The channel is defined with the keyword *channel*, and its ports are defined with *input* or *output* keywords.

In the constructor of the *MainAgent* we first create an instance of channel *Adder* by calling the static method *CreateInNewDomain* on *AdderAgent*[1]. This instantiates two ends of the *Adder* channel (which are instances of different types), creates an instance of *AdderAgent* that implements the channel, then returns the using end of the channel.

Next, we send messages to the ports *Num1* and *Num2* and wait for the result from port *Sum*.

[1] There is another way to instantiate a channel, which requires hosting – we will describe it on page 18.

The motivation for using agents in this example lies in our ability to overlap the execution of *MainAgent* and *AdderAgent.* We can do something (admittedly, little, in this example) while *AdderAgent* performs the computation.

Schemas

Not every kind of data can be sent over a channel. The reason for that is twofold:

- First, in a distributed scenario, the agents can reside in two different processes, or even two different computers. The data passed between the two agents must therefore be deeply serializable (meaning the object and all transitively referenced objects must be serializable).
- Second, the Axum isolation model (which we will talk about on page 17) requires that two agents executing concurrently don't have access to shared mutable state. To satisfy this, the data in the message must be either not shared, or not mutable; otherwise, the agents cannot execute concurrently.

To support these requirements, Axum introduces the concept of *schema*. A schema is a type that defines *required* and/or *optional* fields, but no methods. The concept is very similar to XML schema, or, to a lesser degree, a *struct* in C#.

Schema fields are deeply serializable and immutable

When an instance of a schema crosses a process or computer boundary in a message, the data payload is deeply serialized. This means that you can only declare fields in schema whose types can be serialized using .NET-serialization. When the message is sent locally (in-process), deep serialization is not required – however the fields in the schema are considered immutable.

Schemas are declared with the keyword *schema*, followed by the schema's name, followed by a semicolon-separated list of fields enclosed in curly-braces:

```
schema Customer
{
    required string Name;
    optional string Address;
}
```

The distinction between the required and the optional fields allows for loose coupling between different schemas, or between schemas and different types with the same "shape"[2]. This is useful when the communicating parties cannot all use the same type of schema – for example, when one component was updated with a new version of the product but the other was not.

A type can be converted (or *coerced*) to another type of the same shape using the *coerce* operator. Consider a C# class *CustomerData* defined as:

[2] That is, having the same required fields but not necessarily the same optional fields

```
class CustomerData
{
    public string Name;
}
```

Then, an instance of schema *Customer* can be coerced to type *CustomerData*, even though it is missing the optional field *Address*:

```
Customer c = new Customer{Name = "Artur", Address = "One Microsoft Way"};
CustomerData cd = coerce<CustomerData>(c);
```

Axum compiler will generate an error if it is known at compile time that the coercion will fail. If, on the other hand, there is a chance that it can succeed, the compiler will generate code that may or may not succeed at runtime.

Schemas are also useful for combining different data types into a single container type, instances of which can be sent in one message. In the next example, we define a schema as a tuple of two integers, which we then send over a channel.

Request-reply ports

Let's revisit the example with channel *Adder* above. As it is, it can only accept one request, after which the agent *AdderAgent* shuts down.

An obvious solution would be to modify *AdderAgent* to not shut down after the first request, but to keep accepting new requests indefinitely, or until some condition is met.

> Sending a message to a request-reply port returns a correlator, from which you receive the requested value.

This is a good start, but what if we want the user to able to submit multiple requests, and then retrieve results of these requests in arbitrary order? To achieve this, we need a way to correlate the requests with results produced by the *AdderAgent*. One way to do this is to return to the client a "ticket" that can later be used to "claim" the result. In Axum, such a ticket is called the *request correlator*, and ports that support request correlators are called *request-reply ports*.

Now, let's rewrite the *Adder* sample using request-reply ports:

```
using System;
using System.Concurrency;
using Microsoft.Axum;

schema Pair
{
    required int Num1;
    required int Num2;
}

channel Adder
{
    input Pair Nums : int; // input request-reply port
```

```
}

agent AdderAgent : channel Adder
{
    public AdderAgent()
    {
        while(true)
        {
            var result = receive(PrimaryChannel::Nums);
            result <-- result.RequestValue.Num1 +
                       result.RequestValue.Num2;
        }
    }
}

agent MainAgent : channel Microsoft.Axum.Application
{
    public MainAgent()
    {
        var adder = AdderAgent.CreateInNewDomain();
        var correlator1 = adder::Nums <-- new Pair{ Num1=10, Num2=20 };
        var correlator2 = adder::Nums <-- new Pair{ Num1=30, Num2=40 };

        // do something useful here ...

        var sum1 = receive(correlator1);
        Console.WriteLine(sum1);
        var sum2 = receive(correlator2);
        Console.WriteLine(sum2);

        PrimaryChannel::ExitCode <-- 0;
    }
}
```

As you can see, submitting a request to the adder yields a request correlator (*correlator1* and *correlator2*), which then can be used to receive the result (naturally, using *receive* expression).

In this example, since we've submitted more than one request to the adder, we have more time to do something useful while these requests are being processed.

Protocols

Let's return again to the *Adder* sample on page 11. It so happens that if *Adder* is used incorrectly, *AdderAgent* and *MainAgent* can each end up waiting for a message from one another, resulting in a *deadlock*.

Consider what happens when a user forgets to send a number to port *Num1*, before sending a number to *Num2*. The *MainAgent* will then proceed to receive from *Sum*, but at the same time, the *AdderAgent* will still be "stuck" waiting for a message from *Num1*, which never comes.

We've run into a classic case of a deadlock: the *MainAgent* is waiting for a message from *AdderAgent*, but the *AdderAgent* is waiting for a message from *MainAgent*. Neither can make any progress.

Intuitively, we understand that the first message is "supposed to be" sent to port *Num1*, followed by a second message sent to *Num2*.

We can formalize our intuitive understanding of how a channel must work by specifying the channel's *protocol*. The protocol is a finite state machine, with *states* and *transitions* between those states defined by the designer of the channel.

> Protocols can turn hard-to-diagnose errors such as deadlocks into explicit protocol violations.

The protocol always starts in a special state called *Start*. When a new message arrives on any of the channel ports, the protocol can either transition to another valid state – if that transition is specified in the protocol – or trigger a protocol violation exception if no valid transition exists.

Here is how we can write a protocol for channel *Adder* – look at the last three lines of the channel's definition:

```
channel Adder
{
    input int Num1;
    input int Num2;
    output int Sum;

    Start: { Num1 -> GotNum1; }
    GotNum1: { Num2 -> GotNum2; }
    GotNum2: { Sum -> End; }
}
```

The protocol starts with a state *Start* and transitions to state *GotNum1* after receiving a message on port *Num1*. Getting a message on any other port at this point would result in a protocol violation.

Next, having transitioned to *GotNum1*, the next port that is expected to be used is *Num2*, which triggers transition to state *GotNum2*.

Finally, message on port *Sum* triggers transition to the built-in state *End*. At this point the protocol is considered closed and any attempt to send a message on any of its ports would trigger an exception.

You can define more elaborate transitions that enable rich protocols. For example, one could say "when a message arrives on port X or Y, transition to state Z", or "transition to state X only if the value of the message is greater than 10" and so on. For brevity, we will not describe the syntax here, but you can look it up in the language specification document.

Let's give it a try. Re-define the channel as shown above and comment out this line:

```
adder::Num1 <-- 10;
```

Next, build and execute the sample. Now, instead of the deadlock you will get a runtime exception saying:

Invalid use of 'Num2' at this point in the conversation, i.e. in state 'Adder.Start'

Unfortunately, not every deadlock can be caught by a stringent protocol.

Consider what happens when a user sends a number to *Num1*, but forgets to send a second number to *Num2* before attempting to receive the result from port *Sum*. Like before, neither *AdderAgent* nor *MainAgent* can make further progress at this point. However, this should not be considered a protocol violation since a message can still be sent to *Num2*, unblocking both *AdderAgent* and *MainAgent*.

Domains and State Sharing

Message-passing is an excellent communication mechanism, but it requires the data in the message to be either deeply copied or immutable. Sometimes it is more efficient, or simpler, to let agents share the data – provided that we can do it safely, of course.

This is where *domains* come into picture. A domain's *raison d'etre* is to allow a group of agents to safely share state, at the same time isolating that state from everyone else.

> Domains allow agents to share state with one another

Sharing State between Agents

Like regular classes, domains can contain fields and methods, but more importantly, domain declarations can include agent declarations. The instances of agents declared within a domain can access that domain's fields.

Here is an example:

```
domain Chatroom
{
    private string m_Topic;
    private int m_UserCount;

    reader agent User : channel UserCommunication
    {
        // ...
    }

    writer agent Administrator : channel AdminCommunication
    {
        // ...
    }
}
```

This declares the domain *Chatroom* that encloses two types of agents – the *User* and the *Administrator*. The chatroom lets multiple users exchange messages with each other (or with the administrator), and read domain state – such as *m_Topic* – but not to change domain state. The administrator is the only type of agent that can modify domain state.

Let's look at the distinction between readers and writers a little closer.

Reader-Writer Semantic

A reader-writer lock is a commonly used synchronization construct that grants access to a shared resource to either multiple readers, or a single writer. This ensures that readers always see the consistent state of the resource, and that no two writers can interfere with each other.

Agents in Axum follow the same principle. An agent declared with the *reader* keyword is only allowed to read domain state, while *writer* agents can both read and write the state. An agent declared as neither reader nor writer can only read the immutable state of the domain.

Keywords *reader* and *writer* specify agent's access mode to the enclosing domain

Within an agent, the enclosing domain's fields and methods can be accessed via the *parent* keyword. An instance of *User* agent can read domain's state thus:

```
reader agent User : channel UserCommunication
{
    public User()
    {
        if( parent.m_Topic.Contains("Axum") )
        {
            // Start talking to other users in this chatroom
        }
        else
        {
            // Do nothing and leave
        }
    }
}
```

If the agent were to try to modify the chatroom's topic, however, the code would not compile:

```
    if( parent.m_Topic.Contains("Axum") )
    {
        // Start talking to other
        // users in this chatroom
    }
    else
    {
        // Let's talk about Axum!
        parent.m_Topic = "Axum Discussion"
    }
```

```
Error: cannot modify
paent.m_Topic from reader
agent User
```

Similar to the *this* reference, *parent* can be omitted. The keyword *parent* is only required when you need to disambiguate between domain members and other symbols with the same name, but otherwise using *parent* is a matter of taste.

The agents are similarly restricted from mutating the global state of the system. For example, an agent cannot write to a static field or call a method that can (even potentially) modify a static field.

When you start programming in Axum, you will notice that some code (especially if it's using a third-party library) may not compile until you mark the agent as writer, or mark the class isolated (see 32 for more details on this). While this might be fine in some cases, bear in mind that writer agents within the same domain cannot execute in parallel. An application with a single domain where all agents are declared as writers is effectively single-threaded!

Axum has an "escape hatch" that lets you execute code that potentially modifies shared state – the *unsafe* keyword. As the name suggests, using *unsafe* is, well, not safe, and you should only use it if you know for sure that the code either does not modify shared state, or such modification is benign. It's always better to be a little slower and correct than to have a program full of really efficient bugs!

Example:

```
reader agent User : channel UserCommunication
{
    private WriteLog(string str)
    {
        unsafe
        {
            Logger.WriteLine(str);
        }
    }
}
```

Here we might want to use unsafe if the third-party class *Logger* did its own synchronization, or if we didn't care about potential interleavings of lines in the log file.

Hosting Agents

Recall our first *Adder* sample on page 11. A new instance of channel *Adder* was created thus:

```
var adder = AdderAgent.CreateInNewDomain();
```

This created two ends of the channel *Adder*, the agent *AdderAgent* that implements it, and returned the using end of channel *Adder.* As the name suggests, the *CreateInNewDomain* method also creates an instance of that domain, and makes *AdderAgent*'s *parent* reference point to it.

Hosting associates the agent type with the domain and the address.

Creating a new domain instance every time you instantiate an agent in this way means that the agents cannot share state – their *parent* references point to the different instances of the domain type. Clearly, this is a problem and we need another way to instantiate agents and associate them with domains.

Hosting is a way to associate the agent's type with a particular instance of the domain. By hosting an agent's type, written as:

```
Host<AdderAgent>("my adder");
```

we say, in effect, "when the user asks for the "Adder" service hosted at address 'my adder', instantiate agent *AdderAgent,* attach it to the current domain, and return channel's *Adder* using end"

The address at which the agent's type is hosted can be any expression as long as it supports equality comparison. We've chosen to use a string, because it is descriptive.

Because *Host* associates agents with the current domain (it is a domain method), it can only be executed in the context of a domain (and not in an agent declared outside of a domain).

Now the agent can be instantiated like this:

```
var adder = new Adder("my adder");
```

You can read this statement as "create the using end of the channel *Adder* implemented by a service hosted at address 'my adder'".

> An agent is a service that implements a channel. Multiple agents types can implement the same channel type.

Notice that the user doesn't know the type of the agent – all it cares is the type of the channel and the address of the service that implements this channel.

To better illustrate the idea of the agent as a service, consider a channel that given an integer, reports the corresponding Fibonacci number. Unlike addition, which can only be done in one meaningful way, Fibonacci can be calculated in a number of different ways – some more efficient than others.

The same channel *Fibonacci* can be implemented by two different agents, let's call them *FibonacciSlow* and *FibonacciFast*. These two agent types can be hosted at different addresses:

```
Host<FibonacciSlow>("slow");
Host<FibonacciFast>("fast");
```

Now depending on the user's need, she may decide to instantiate a *Fibonacci* channel that is implemented by one agent or another:

```
Fibonacci fibChannel;
if(gotTimeToSpare)
{
    fibChannel = new Fibonacci("slow");
}
else
{
    fibChannel = new Fibonacci("fast");
}
var request = fibChannel::Number <-- 42;
...
```

When the service provided by the agent is no longer required, the agent type can be evicted from the domain:

```
Evict("slow");
Evict("fast");
```

Note that evicting an agent type from the domain doesn't affect instances of that agent type that have already been created. Instead, it makes it impossible to create more such instances using the operator *new*. Once the agent type has been evicted, an attempt to instantiate it will result in a runtime exception.

Programming with Dataflow Networks

In a program where the agents are communicating with each other by sending and receiving messages, their communication can be driven by the application logic, the state of the program, the data received from the other agents and so on.

Contrast this control-flow style model with the dataflow model, where execution of the program is driven only by the availability of the data entering the dataflow network, and the computations are performed as the data moves through the network.

Axum supports two communication models:

- ✓ Control-flow driven communication between agents;
- ✓ Dataflow driven communication with networks.

In Axum, dataflow networks are built using network operators. A network operator is a binary expression, with a source sub-expression as a left-hand operand and a target expression as a right-hand operand. The operands can be either scalar or vector forms, where by "scalar" we understand a single-valued data type, and by "vector" either an array or an *IEnumerable* of a type.

Below, we will classify the network operators by the type of their operands.

One to One: Forward

We have seen previously (on page 7) how to build a pipeline using forward operator ==>. This operator takes a single source interaction point and forwards the result to a target interaction point.

Similar to forward, *forward once* operator --> forwards a message from the source to the target, but then disconnects after the first message.

Forward operator ==> sends each message produced by the source to the target.

Forward once operator --> forwards a single message to the target, then disconnects.

`In addition to building pipelines, the operator has other uses – for example, it could be used to build an event-driven system where different actions are performed in response to the events.

Consider a GUI application that handles events such as a mouse click, a key press, and a window paint request. These events can be represented as ports on a channel:

```
channel GUIChannel
{
    // MouseEvent is an enum with values Up and Down
    input MouseEvent Click;

    // Key describes which key was pressed
    input Key KeyPress;

    // Rect is a rectangle to repaint
    input Rect Paint;
}
```

Now the agent *GUIHandler* that handles the requests can be implemented like this:

```
agent GUIHandler : channel GUIChannel
{
    public GUIHandler()
    {
        PrimaryChannel::Click ==> HandleClick;
        PrimaryChannel::KeyPress ==> HandleKeyPress;
        PrimaryChannel::Paint ==> HandlePaint;
    }
    void HandleClick(MouseEvent mouseEvent)
    {
        ...
    }
    void HandleKeyPress (Key key)
    {
        ...
    }
    void HandlePaint (Rect rect)
    {
        ...
    }
}
```

The constructor of *GUIHandler* sets up three simple networks that forward incoming messages to the corresponding handler methods.

Many to One: Multiplex and Combine

The following two operators take a vector of sources and produce a single target.

The *multiplex* operator >>- takes a vector of sources as the left-hand operand and forwards data from each into a single target as soon as it arrives at any of the sources.

Similarly, the *combine* operator &>- takes a vector of sources, receives a message from each, then packages them into an array (thus the name "combine") and

Multiplex operator >>- sends messages from a vector of sources to a single target.

Combine operator &>- joins multiple messages from sources and propagates them to a target as an array.

forwards the result to the right-hand operand. Unlike multiplex, combine waits for all sources to have a message, before joining them together and propagating them all to the target.

Let's illustrate this with an example. Consider an array of two interaction points, containing numbers 10 and 20, respectively:

```
var ip1 = new OrderedInteractionPoint<int>();
var ip2 = new OrderedInteractionPoint<int>();

ip1 <-- 10;
ip2 <-- 20;

var ips = new OrderedInteractionPoint<int>[] { ip1, ip2 };
```

Now, we can set up a multiplex network expression:

```
    ips >>- PrintOneNumber;
```

This expression sends the data from the array *ips* to a node created from method *PrintOneNumber*:

```
void PrintOneNumber(int n)
{
    Console.WriteLine(n);
}
```

Alternatively, we could set up a combine expression:

```
    ips &>- PrintManyNumbers;
```

This expression would forward all numbers as one message to a node taking an array of *int*s:

```
void PrintManyNumbers(int[] nums)
{
    foreach(var i in nums)
        Console.WriteLine(i);
}
```

Combine is also useful when you need to wait for multiple messages that can arrive in any order. For example, the following expression combines output from two interaction points *ip1* and *ip2* and passes the result on to an interaction point *twoNumbers*:

```
    receive( { ip1, ip2 } &>- twoNumbers );
```

The expression above uses curly braces for array creation. In Axum, implicit array creation is a convenient syntactic construct that is used often when building network expressions. We will see below how it is used to build a more complex network.

One to Many: Broadcast and Alternate

The last two network operators take a single source interaction point, and propagate the data to multiple targets.

> **Broadcast** operator -<< copies one message to multiple targets
>
> **Alternate** operator -<: round-robins messages between multiple targets

The *broadcast* operator -<< accepts a single interaction point on the left and a collection of interaction points on the right; it propagates all data from the left operand to all the interaction points on the right.

Propagating data to multiple sources can be used to implement a publisher-subscriber scenario where data from a publisher is propagated to a number of subscribers.

Finally, the *alternate* operator -<: propagates the data from the single source to the targets in the right-hand collection in round-robin order.

Alternate is useful in the scenarios where we want to load-balance work. For example, we might want to have a pool of "workers" to handle data coming into the source.

To put it all together, let's revisit our first adder example from page 11, and re-implement it using a dataflow network.

```
agent AdderAgent : channel Adder
{
    public AdderAgent()
    {
        var ipJoin = new OrderedInteractionPoint<int[]>();

        { PrimaryChannel::Num1 ==> ShowNumber, PrimaryChannel::Num2 ==> ShowNumber }
            &>- ipJoin -<: { GetSum, GetSum } >>- PrimaryChannel::Sum;
    }

    private int ShowNumber(int n)
    {
        Console.WriteLine("Got number {0}", n);
        return n;
    }

    private function int GetSum(int[] nums)
    {
        return nums[0] + nums[1];
    }
}
```

An explanation is in order. The first node in the network is an array of two elements, where each element forwards the incoming data to the transfer method *ShowNumber*. We use this method for debugging purposes, to keep track of the messages as they arrive at the input ports.

Then, the array node is combined into an indirection point *ipJoin* of array of *int*s. The goal of this node is take the two numbers and combine them into a tuple that can be processed by the method *GetSum*.

The next node in the network is again an array, which contains two transformation nodes implemented by the method *GetSum*. Because *GetSum* is declared as a function, its execution can be scheduled concurrently, which improves the throughput of the network.

Finally, the output of the *GetSum* transformations is multiplexed into the output port *Sum*.

Distributing an Axum Application

Axum has great support for writing distributed applications. In fact, one of the reasons for taking such a hard line on isolation is so that domains can interact locally or remotely with no change in the model. By borrowing a page from how the Web is programmed and making it scale to the small, we can easily go back to its roots and interact across networks.

With domains being services of a SOA application, agents the protocol handlers, and schema the payload definitions (b.t.w. schema are XML-schema compliant), we have an easy time mapping Axum to web services.

In the Axum runtime, we have support for local, in-process channels as well as WCF-based channels.

To reach an agent within a domain, you have to give it an address; this is true in local and remote scenarios alike. Within a process, it's a bit easier, because the agent type name itself acts as a "default" address if nothing else, but in the distributed scenario, we have to do a bit more. But it's just a little bit.

The Axum runtime does this through an interface called *IHost*, which allows you to give the agent an address within a domain. To be precise, what we associate with an address is a factory for agents of the hosted type, which is used to create agent instances when someone creates a new connection. Each underlying communication / service hosting framework has to have its own implementation of *IHost*; Axum comes with one for WCF and one for in-process communication.

The address may be associated with an existing domain instance, in which case created agent instances are associated with that domain, or it may be associated with no domain instance, in which case created agent instances are associated with a new domain instance, one for each new connection.

For example, if you are building an Axum-based server, you can host domains as services with the following code:

```
channel Simple
{
    input string Msg1;
    output string Msg2;
}
```

```
domain ServiceDomain
{
    agent ServiceAgent : channel Simple
    {
        public ServiceAgent ()
        {
            // Do something useful.
        }
    }
}
agent Server : channel Microsoft.Axum.Application
{
    public Server ()
    {
        var hst = new WcfServiceHost(new NetTcpBinding(SecurityMode.None, false));

        hst.Host<ServiceDomain.ServiceAgent>("net.tcp://localhost/Service1");
    }
}
```

Each time some client connects to the address "net.tcp://localhost/Service1," a new instance of *ServiceAgent* will be created, associated with a brand new *ServiceDomain* instance. If instead, we wanted created agents to be associated with a single domain instance, we have to pass one in to 'Host':

```
hst.Host<ServiceDomain.ServiceAgent("net.tcp:...", new ServiceDomain());
```

There is a corresponding interface for the client side, called *ICommunicationProvider*. This is used to create a new connection to an Axum service (or any service, for that matter, we have no knowledge that it's written in Axum, a consequence of loose coupling). It, too, must have a version for each underlying communication framework and the Axum runtime comes with one for WCF and one for in-process communication.

Connecting to the service above would look like this:

```
var prov = new WcfCommunicationProvider(new NetTcpBinding(SecurityMode.None, false));

var chan = prov.Connect<Simple>("net.tcp://localhost/Service1");
```

Of course, you don't have to create a new communication provider for each connection, or a new host for each *Host* call.

That's pretty much it – you just make sure that you choose the right WCF binding, and it's off to the races with WCF doing all the hard work for us. If you are using schema types to define your channel payloads, they are already *DataContract*-compliant and safe to use for both inter- and intra-process communication.

As it turns out, this is no different from how you program Axum within a process boundary: the only different is what concrete *IHost*/*ICommunicationProvider* implementations you use, and what

the addresses you create for your agents look like. In other words, the programming model for distributed and local concurrency in Axum applications is identical.

Appendix A: Asynchronous Methods

In Axum, there are few sources of blocking:

- Direct or indirect use of .NET synchronization primitives such as *Monitor.Enter*.
- Direct or indirect use of synchronous I/O, e.g. *Console.ReadLine*.
- receive expressions (discussed later)
- receive statement (discussed later)

Blocking is an unfortunate limitation, but a reality of programming .NET. Furthermore, it is quite cumbersome to program library-based solutions to avoid all blocking. Axum can automatically transform methods into asynchronous constructs, aided by the programmer. It does not, however, avoid blocking issues related to *Monitor.Enter* and other non-Axum synchronization primitives. These should not be used in Axum, which uses messages and empty/full variables for all data-exchange synchronization needs. Synchronization for protection purposes are handled declaratively.

Writing non-blocking code by hand is typically complex, tedious, and error-prone. The Axum compiler takes care of the tedious and subtle rules of doing it and allows you to concentrate on the logic of the algorithm itself.

It does so in methods and functions declared as 'asynchronous.' In such methods, receive expressions and statements are implemented without blocking the thread, and control-flow constructs are also transformed using transformations of continuation points.

Within an asynchronous method, transformations are made not just for receives, but also for any calls where there is an asynchronous alternative that adheres to the .NET Asynchronous Programming Model (APM). That model relies on splitting an operation XXX into a *BeginXXX* and an *EndXXX* call, one to start the operation, one to collect the results.

For example, *System.IO.Stream* has a method *Read* defined on it which has an APM alternative. Code within an Axum asynchronous method referring to *Stream.Read* would instead be using *Stream.BeginRead /EndRead* and avoid blocking the thread.

```
private asynchronous void ReadFile(string path)
{
    Stream stream = new Stream(...);

    int numRead = stream.Read(...);
    while (numRead > 0)
    {
       ...
       numRead = stream.Read(...);
    }
}
```

Using asynchronous methods in your Axum code can significantly reduce the cost of message-passing and doing I/O and thus improve its scalability immensely[3].

Using asynchronous methods is often overkill and has a performance penalty for small methods and functions which do not do I/O or messaging. Therefore, the rule is that methods are synchronous and have to be explicitly identified as asynchronous. The only methods that are asynchronous by default are agent constructors.

Most Axum constructs are available for use in both synchronous and asynchronous methods.

The rule of thumb around declaring asynchronous methods is as follows:

1. Any method containing a receive expression or statement should be asynchronous.
2. Any method calling an API that is known to have an APM variant should be asynchronous.
3. Any method calling an asynchronous method should be asynchronous.

[3] The author was able to observe 500,000 simultaneously blocked agents on his laptop without seeing the thread pool create any additional threads (there were 6 threads before they were started, 6 threads when they were all sitting blocked). He didn't try more than that...

Appendix B: Defining Classes

Axum is a coordination language, and as such, does not accept class definitions. If you would like to add class definitions to your application there are two ways to do so: via creating a separate C#, F#, Managed C++, or Visual Basic.NET project in the same solution as your Axum project, or by directly compiling C# source in your Axum project.

Using a Separate Managed Language Project

Axum is a .NET language that builds upon the Common Language Runtime (CLR) and, therefore, can interact with classes defined in any .NET language. To define a class for use in your Axum project, simply:

1. Add a new C#, Visual Basic, or any other CLR-based language project to your solution.
2. Right-click your Axum project and select "Add Reference…"
3. Choose the "Projects" tab from the "Add Reference" window.
4. Select the other project you just added and click "OK".

You should now be able to declare any type you have defined in your non-Axum project.

Using C# within an Axum Project

Sometimes, for simple Axum projects, it's either unnecessary or inconvenient to add an additional project just to define a new type. To make this process simpler, the Axum project system can compile C# files. To add a C# file to your project:

1. Right click on your Axum project and select "Add/New Item…"
2. In the "Add New Item" window, select "C# Class".
3. Replace "Class1.qcs" with the name of your new class

Note: The Axum project system will compile C# files with a special C# 3.0 compiler that contains necessary support for some Axum concepts. This compiler can be used to evaluate the Axum language. None of the features in this special C# compiler are indicative of potential features on the official Microsoft C# compiler roadmap.

Appendix C: Understanding Side-Effects

Isolated Classes

When adding a new class to your Axum project, you will notice that the generated class declaration will contain a new keyword, *isolated*:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MyProgram
{
    // Isolated classes cannot modify statics
    isolated class MyClass
    {
        // ...
    }
}
```

The Axum project system will invoke an experimental version of the C# compiler that supports the new keywords such as *isolated* and others as described below.

The methods of an isolated class cannot modify any static fields. So if you were to define a method *f* that modifies a static field *n*, the compiler would produce an error:

```
isolated class MyClass
{
    static int n;
    public static void f()
    {
       n=1; // Static field 'MyProgram.MyClass.n' cannot be assigned to in an isolated method
    }
}
```

Both classes and methods can be marked isolated. When applied on a class, the keyword *isolated* means all that methods of the class are isolated. Isolated methods cannot call non-isolated methods.

Only isolated methods can be safely used with Axum. The reason for that is simple: imagine two reader agents running concurrently, instantiating and using different instances of the class *MyClass*. Now if a method of the class were to modify a static field, we'd be having a data race – exactly the kind of problem Axum aims to solve.

We can restrict the effects of the method even further by declaring it *readonly*. A readonly method cannot modify any non-static fields of the class it is declared in. Readonly methods are also considered isolated.

When a reader agent calls a method on a domain-level field, in order to avoid data races, we need to guarantee that the method does not modify the object itself (in addition to the guarantee that it doesn't modify any statics, which is given by the method being isolated). Here is an example:

```
domain D
{
    Employee employee;
    reader agent A : channel Microsoft.Axum.Empty
    {
        public A()
        {
            int age = employee.GetAge(); // safe?
        }
    }
}
```

For the call to *GetAge* to be safe, we need to know that it doesn't mutate the domain field *employee*. This guarantee is provided by the *readonly* keyword:

```
isolated class Employee
{
    public readonly int GetAge() { return 21; }
}
```

Isolation Attributes

Only the experimental C# compiler used with Axum supports *isolated* and *readonly* keywords. Because that compiler has not undergone the same level of testing as the production C# compiler that ships with Visual Studio 2008, it might be desirable to keep using the production C# compiler but to augment the types and methods you want to expose to Axum with special custom attributes.

The Axum compiler will recognize these attributes and treat the types and methods decorated with these attributes as if they were defined with the *isolated* and *readonly* keywords.

The custom attributes are defined in the TseCorelib.dll assembly. You will need to include this assembly into your project and augment the types and methods you want to use from Axum with *Strict* and *Readable* attributes, which correspond to, respectively, *isolated* and *readonly* keywords. Here is how you would define the class *Employee* from the example above:

```
[Strict]
class Employee
{
    [Readable]
    public int GetAge() { return 18; }
}
```

This assembly that defines the class can now be compiled using the production C# compiler and referenced in your Axum project.

Contract Assembly

What if you cannot re-compile the assembly with neither the production nor the experimental C# compiler? For example, you might not have the sources of the assembly, or you might not want to deploy and service that assembly.

The solution is to create a *contract assembly*. The contract assembly defines the same types and methods as the assembly you want to reference in your Axum project. The contract assembly does not substitute the original assembly, and does not need to be deployed with the original assembly. Its only purpose is to provide additional information to the Axum compiler, when it references the "real" assembly.

The types in the contract assembly should be enclosed in the *Contracts* namespace. Since the types in the contract assembly are only used at the compile time, and not supposed to be instantiated, it's a good idea to prevent the user from accidentally instantiating the types or calling the methods of the contract assembly. This could be achieved by either marking the methods abstract or making the methods throw an exception[4].

> Types and methods of the contract assembly are merely placeholders for the isolation attributes; they are not supposed to be instantiated or invoked.

A contract assembly can be included into the project in the same way as any other referenced assembly – that is, either by using a /r compiler switch or by adding the assembly to the list of referenced assemblies when compiling using Visual Studio environment.

A contract assembly is recognized by the Axum compiler by the presence of the special *ContractAssembly* attribute. Place the following line

```
[assembly: System.Diagnostics.Effects.ContractAssembly]
```

By convention, such attributes are usually placed in the *AssemblyInfo.cs* file, which is created automatically by the Visual Studio New Project wizard.

One contract assembly that comes with Axum is called TseContracts.dll. This contract assembly describes some types from the .NET Base Class Library (BCL). Because this assembly is implicitly referenced by all Axum project, you can use some of the most common BCL classes in your Axum projects[5].

The following is an excerpt from TseContracts.dll that describes the effects of the indexer and the *Contains* method of the generic class *List*:

```
using System.Diagnostics.Effects;

namespace Contracts
{
    namespace System.Collections.Generic
```

[4] Not every method can be made abstract – for example, static methods.
[5] The Tsecontracts.dll assembly is by no means complete; there are many side-effect-free classes in BCL that are not included into the assembly.

```
    {
        [Strict]
        public abstract class List<T>
        {
            public abstract int this[int index] { [Readable] get; }

            [Readable]
            public abstract bool Contains(T value);
        }
    }
}
```

As you can see, the class List is marked with the *Strict* attribute which indicates the class is isolated. The indexer and the *Contains* method are marked with the *Readable* attribute, since they don't mutate the list itself and can be called concurrently.

When attributing an existing assembly, or defining your own contract assembly, keep in mind that the Axum compiler cannot verify the validity of the attributes in the contract assembly – that remains solely the responsibility of the author of the assembly. For example, one could include a readable method *Add* to the contract of the class *List* above – however that would be incorrect (for instance, it would make it possible for the two reader agents running concurrently to call *Add* on a domain-level field, which would be unsafe)