# Miruken-DotNet

**Feb 20, 2021**

# Table of Contents

Miruken handle()'s your application!

CHAPTER 1

# Protocol

A protocol describes a set of expected behavior. In C# we can define protocols using interfaces.

CHAPTER 2

# Context

The context is one of the three major components of Miruken. The other two major components being the protocol and the handler.

In this example we are simply creating a new context:

```
namespace Example.mirukenExamples.context
{
    using Miruken.Context;

    public class CreatingAContext
    {
        public Context Context { get; set; }

        public CreatingAContext()
        {
            Context = new Context();
        }
    }
}
```

At the simplest level a context is a collection of handlers.


Here we are instantiating handlers and adding them to the context:

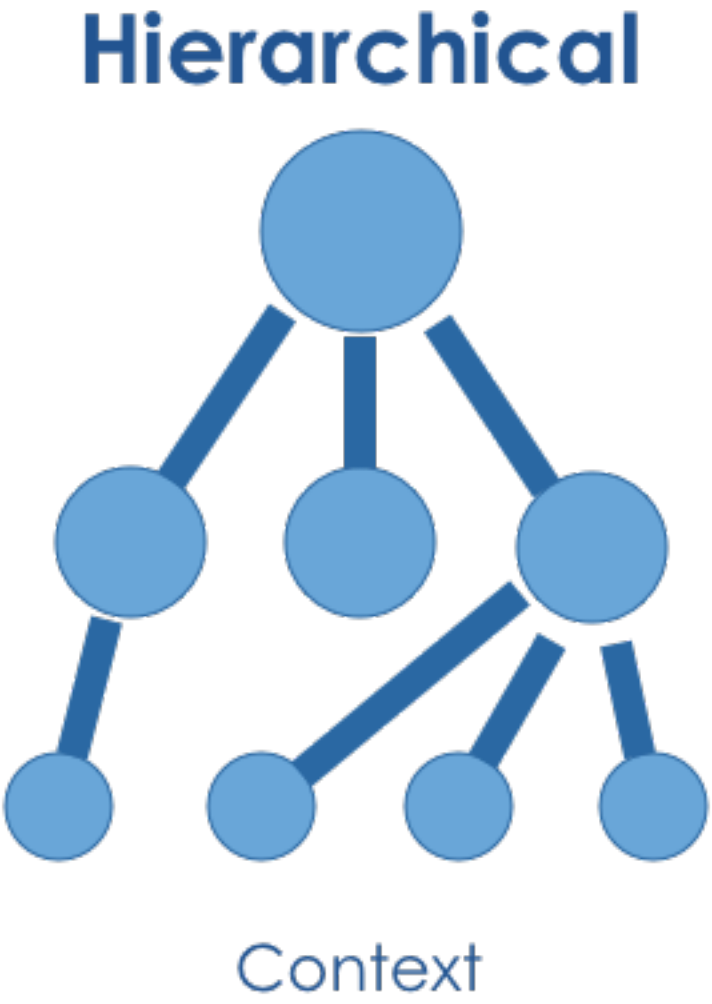
```
namespace Example.mirukenExamples.context
{
    using Miruken.Context;

    public class AContextWithHandlerInstances
    {
        public Context Context { get; set; }

        public AContextWithHandlerInstances()
        {
            Context = new Context();
            Context.AddHandlers(new SomeHandler(), new AnotherHandler());
        }
    }
}
```

You can also rely on a container to create the handler instances. We like to use Castle Windsor, but as you can see by this example you can use any container you want. All you need is a handler in the context that implements *IContainer* and it will create the handler instances for you. My simple container here just instantiates instances and returns them.

```
namespace Example.mirukenExamples.context
{
    using System;
    using Miruken.Callback;
    using Miruken.Concurrency;
    using Miruken.Container;
    using Miruken.Context;

    public class RelyingOnAContainerToResolveHandlers
    {
        public Context Context { get; set; }

        public RelyingOnAContainerToResolveHandlers()
        {
            Context = new Context();
            Context
                .AddHandlers(new ContainerHandler())
                .AddHandler<SomeHandler>()
                .AddHandler<AnotherHandler>();
        }
    }

    public class ContainerHandler: Handler, IContainer
    {
        public T Resolve<T>()
        {
            return (T) Resolve(typeof(T));
        }

        private object Resolve(Type type)
        {
            if (type == typeof(SomeHandler))
                return new SomeHandler();

            if (type == typeof(AnotherHandler))
                return new AnotherHandler();

            throw new ArgumentException("Unknown type");
        }

...
```

## 2.1 Hierarchical

Contexts are also hierarchical. They have a context graph witch mean they know their parent and can create children.



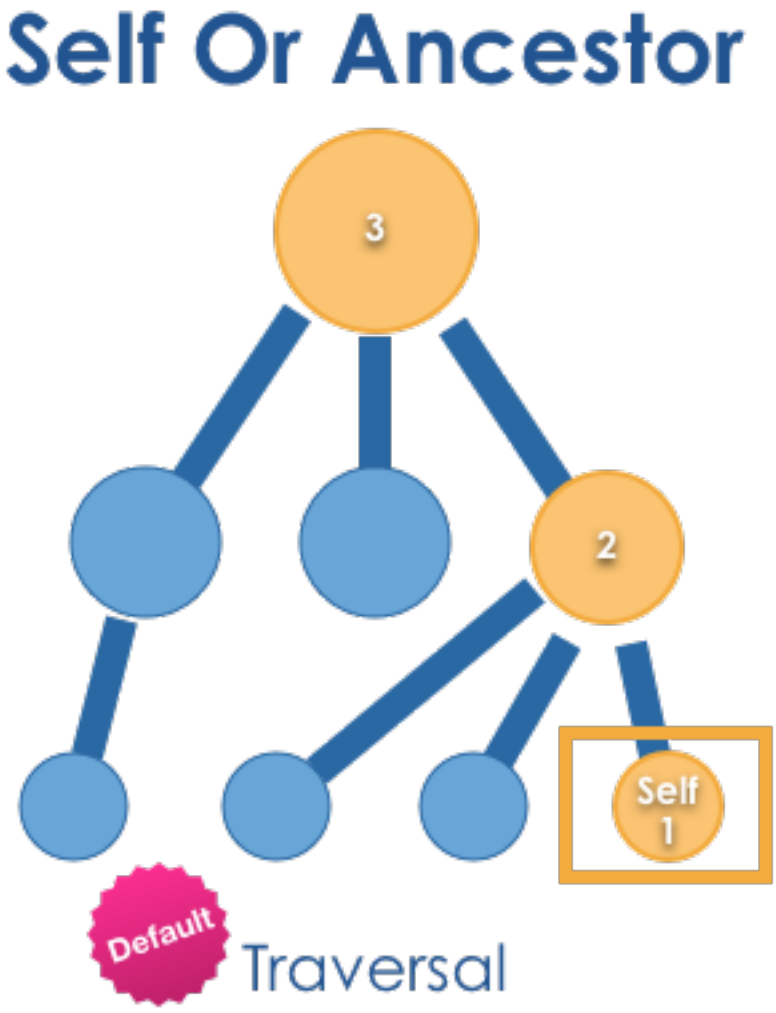

In this example we use the CreateChild() method to create a child context from an existing context:

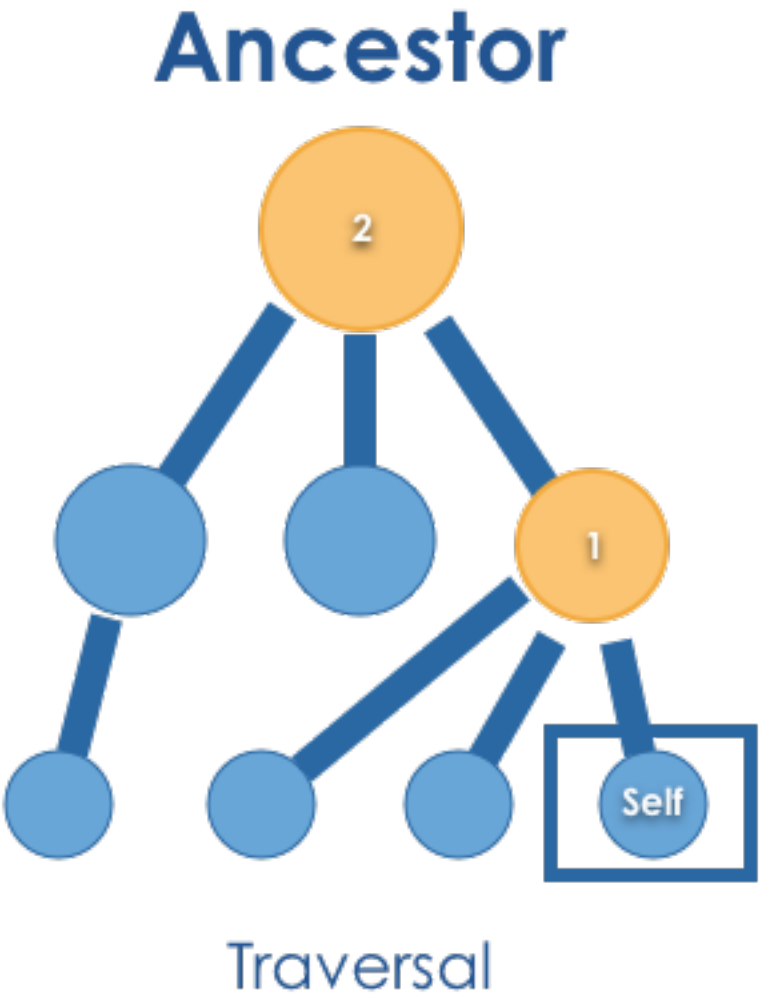
```
namespace Example.mirukenExamples.context
{
    using Miruken.Context;

    public class CreatingAChildContext
    {
        public Context Parent { get; set; }
        public Context Child  { get; set; }

        public CreatingAChildContext()
        {
            Parent = new Context();
            Child  = Parent.CreateChild();
        }
    }
}
```

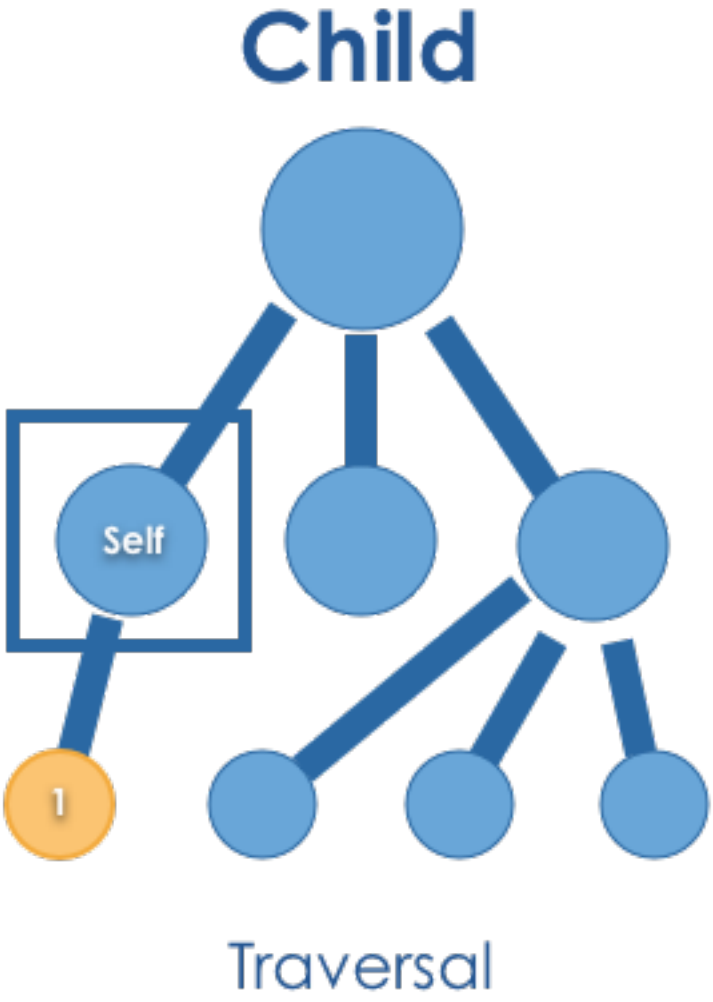
## 2.2 Context Traversal

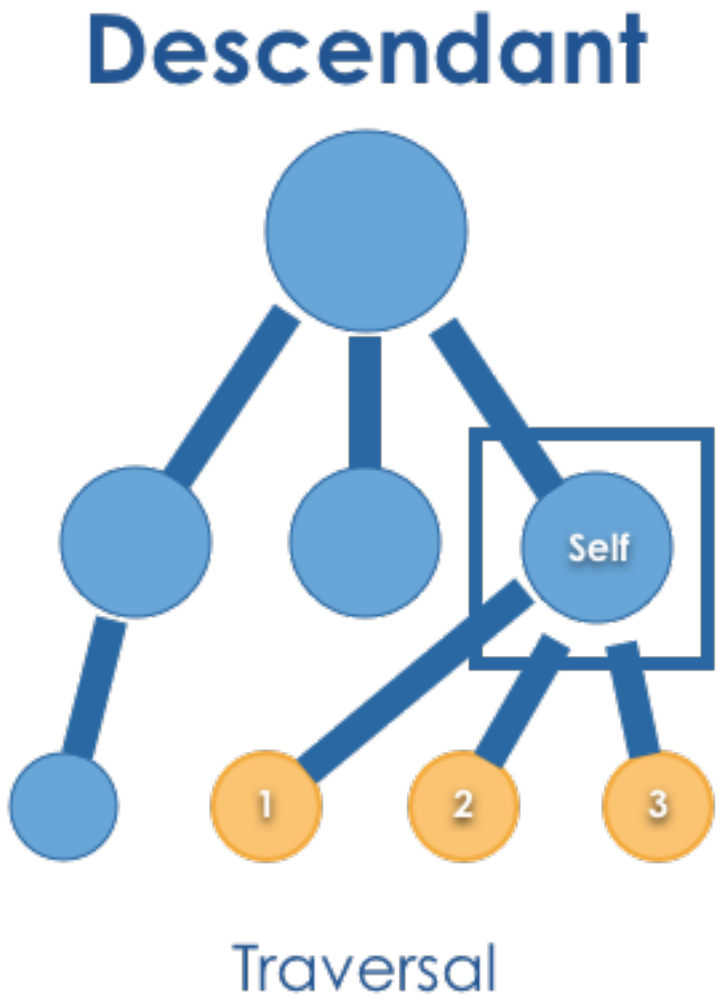
Context traversal is the concept of finding a handler for a message in the current context graph.

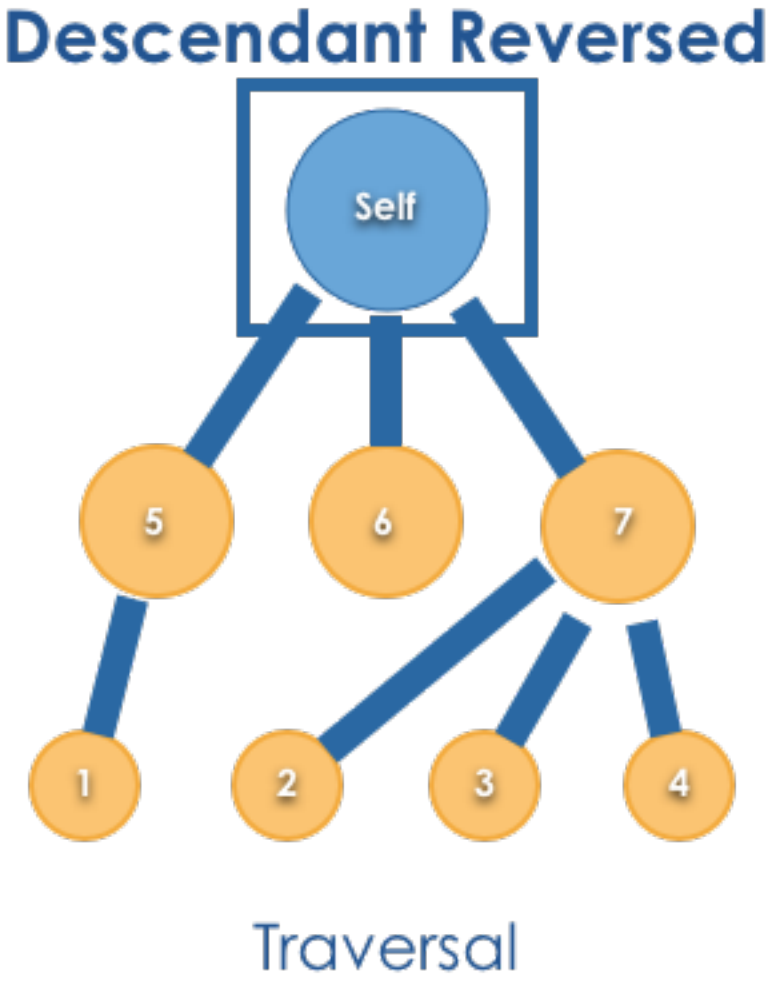
SelfOrAncestor is the default TraversingAxis which means that when Miruken is trying to handle a message it starts with the current context. If the current context cannot handle the message, the message will be passed to the parent to be handled. There are several other TraversingAxis. You can see them all described below.

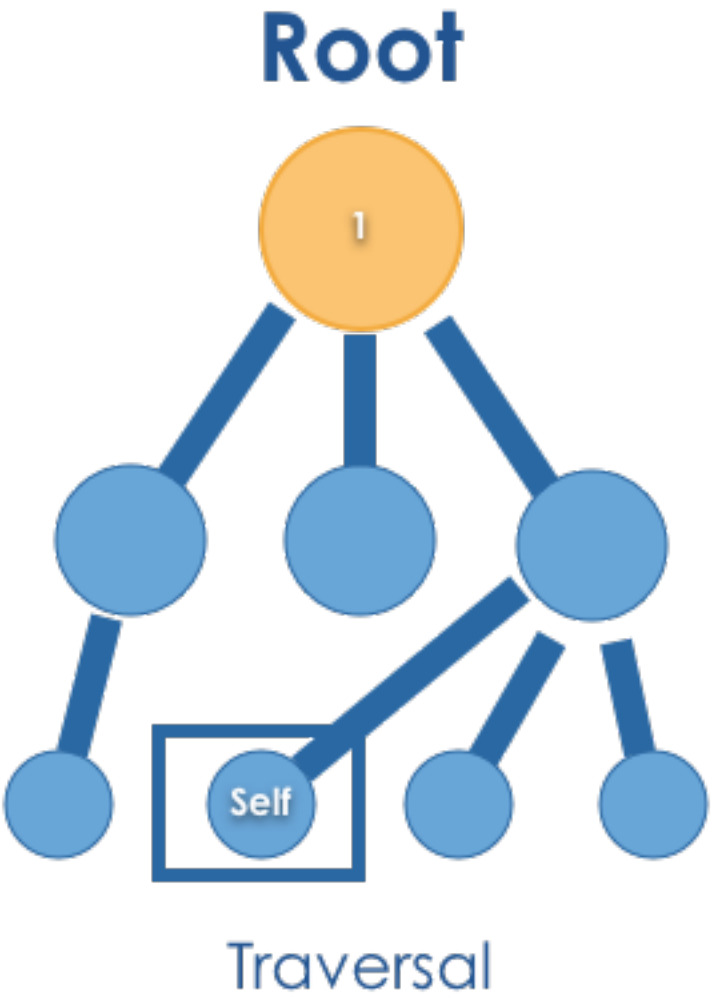


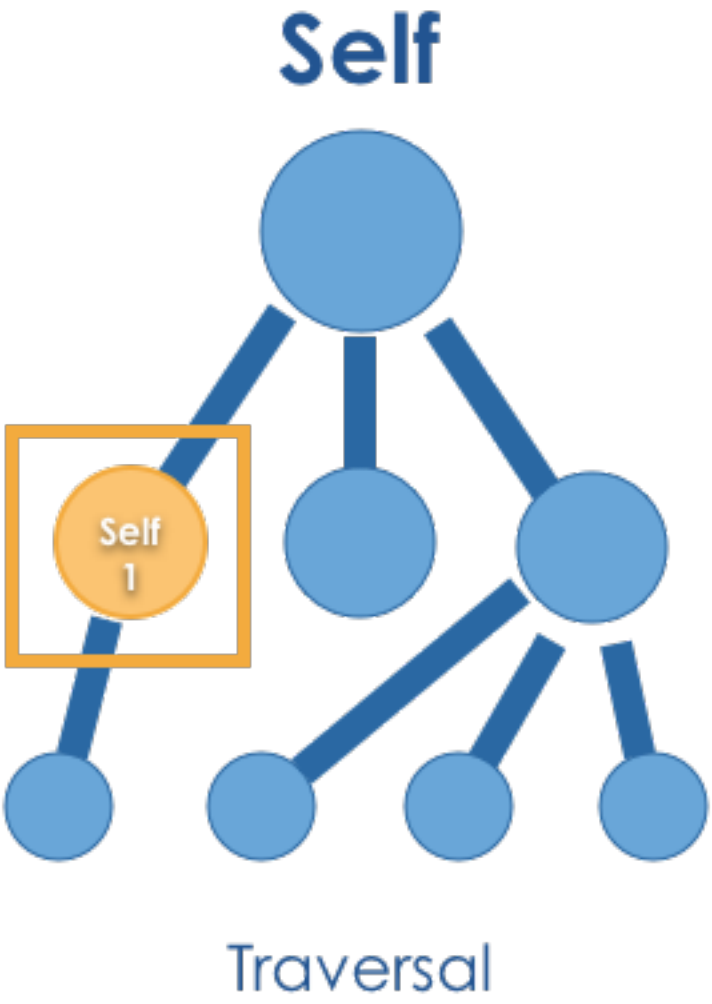

Ancestor
2
1
Self
Traversal

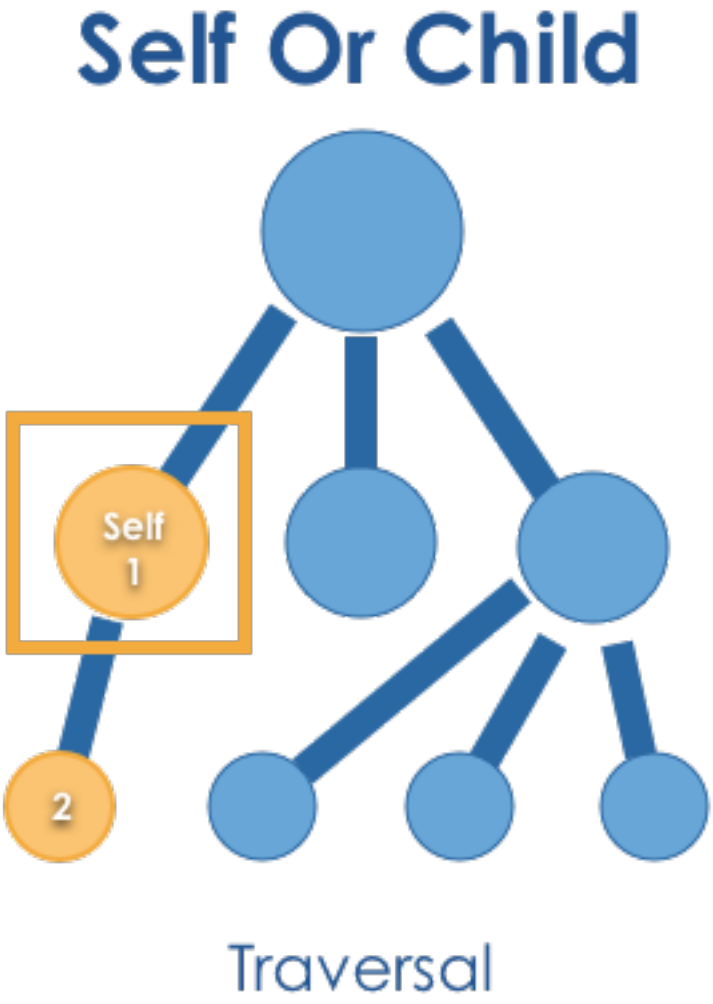

Child
Self
1
Traversal

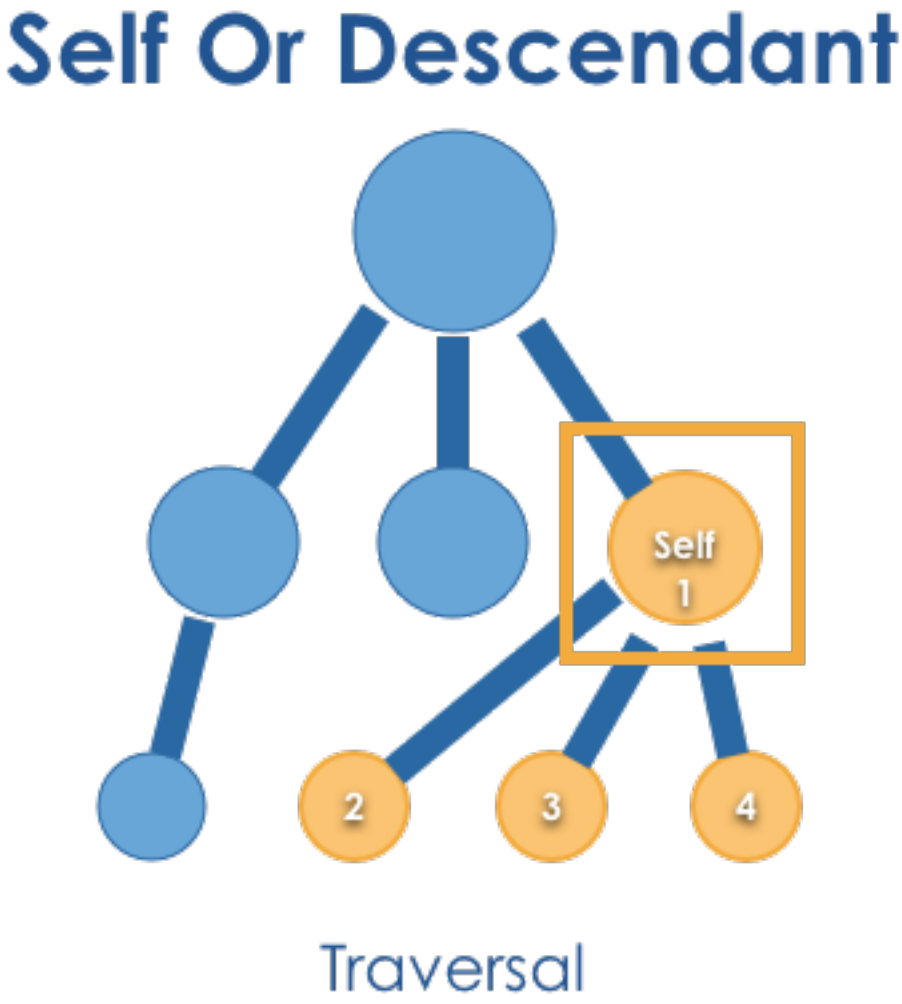

Descendant
Self
1
2
3
Traversal

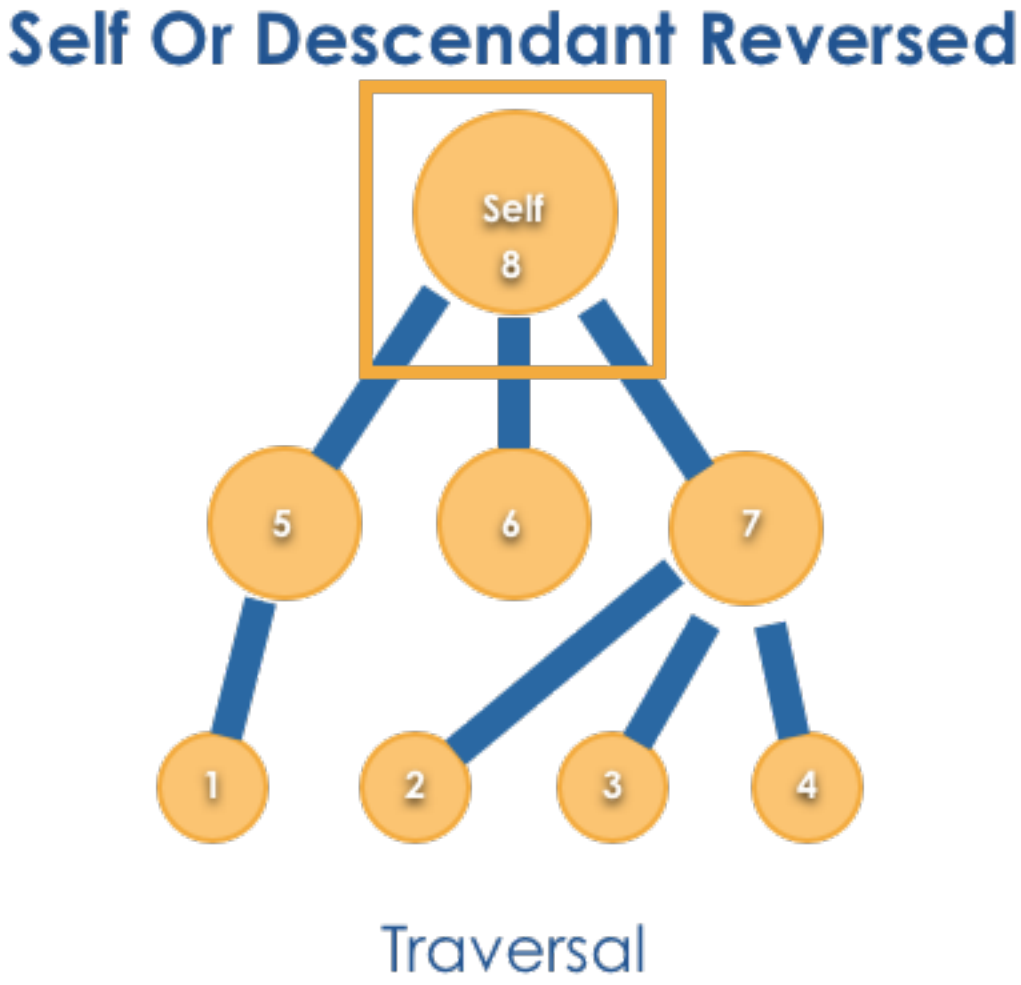

Descendant Reversed
Self
5
6
7
1
2
3
4
Traversal

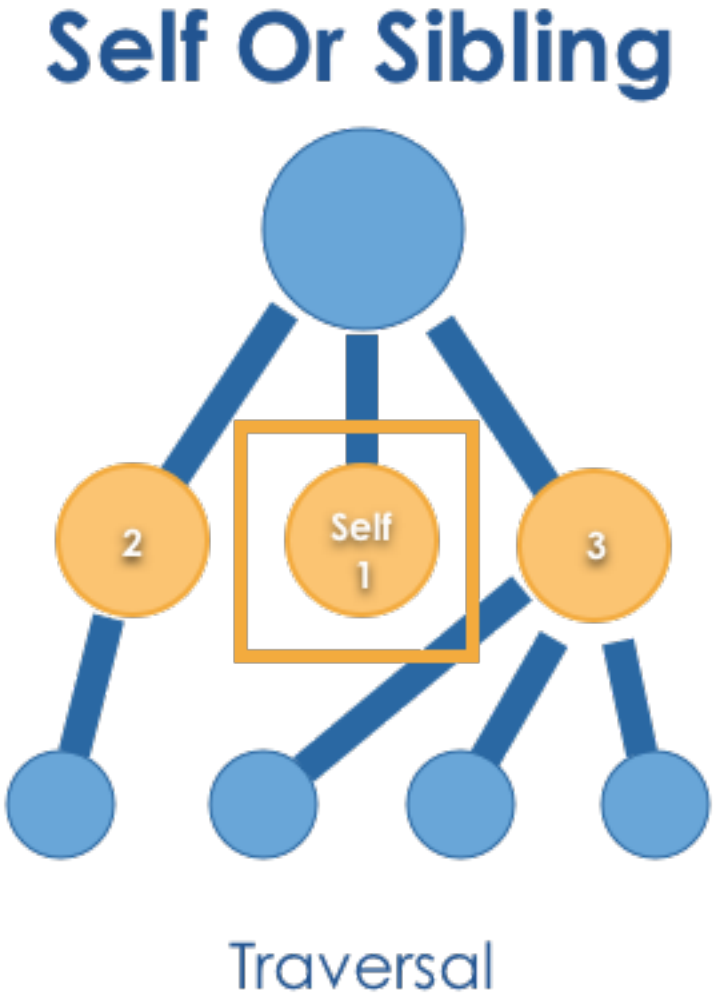

Root
1
Self
Traversal

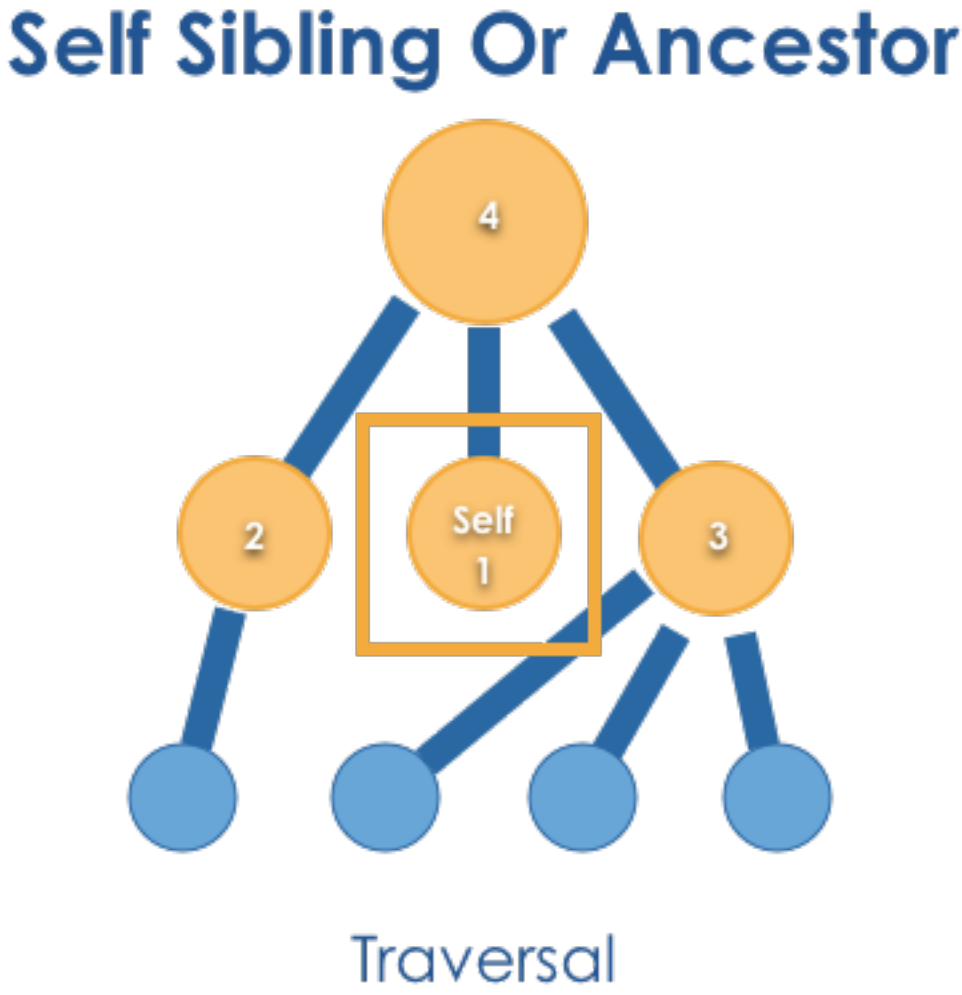

Self
Self
1
Traversal

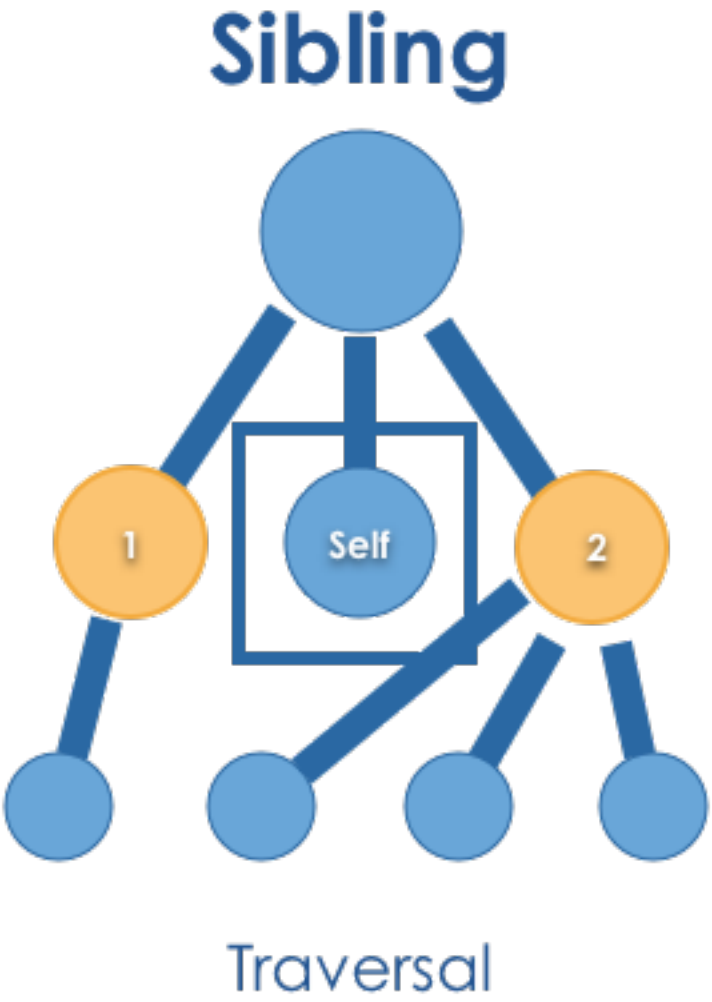

Self Or Child
Self
1
2
Traversal


Self Or Descendant
Self
1
2
3
4
Traversal


Self Or Descendant Reversed
Self
8
5
6
7
1
2
3
4
Traversal


Self Or Sibling
2
Self
1
3
Traversal


Self Sibling Or Ancestor
4
2
Self
1
3
Traversal




## 2.3 Lifecycle

- Context.End
- Context.Ended

CHAPTER 3

# Handler

Handlers implement the application logic described by your protocols.

CHAPTER 4

# Provide

[Provide]

CHAPTER 5

# Promise

We didn't set out to write a promise library in C#. It was born out of necessity. We inherited a C# project written in CE 6 which had no async and await support. We found that doing asynchronous work by hand is verbose and error prone so we needed a better way. Thus Miruken's implementation of the A+ Promise Specification was born. Surprisingly we really like using promises in C#. Especially, since we often switch back and forth between JavaScript and C#. The code looks and feels very simular.

CHAPTER 6

# Configuration

Miruken handle()'s your application!

## 6.1 Configuring Castle Windsor

**Note:** Miruken has first class integration with Castle Windsor, but Miruken does not require you to use Castle Windsor for your container. Miruken does not even require you to use a container. All of that being said, we love Castle Windsor and use it in our own projects.

One of the main ways of configuring a Castle Windsor container is the `Container.Install()` method. It accepts a comma seperated list of `IWindsorInstaller` instances. These installers do all the work of registering objects and configuring the container.

In this very basic Castle Windsor Container all the `IWindsorInstaller` classes in the current assembly will be run. `FromAssembly.This()` returns an `IWindsorInstaller`.

```
namespace Example.mirukenCastleExamples
{
    using Castle.MicroKernel.Resolvers.SpecializedResolvers;
    using Castle.Windsor;
    using Castle.Windsor.Installer;

    public class BasicWindsorContainer
    {
        public IWindsorContainer Container { get; set; }
```

(continues on next page)

(continued from previous page)

```
        public BasicWindsorContainer()
        {
            Container = new WindsorContainer();
            Container.Kernel.Resolver.AddSubResolver(
                new CollectionResolver(Container.Kernel, true));
            Container.Install(FromAssembly.This());
        }
    }
}
```

We used to use this simple form of configuration, but found that we had to list assemblies multiple times. Features and FeatureInstaller solve this problem.

### 6.1.1 Features

At a high level a feature is an implementation of a Miruken concept. It may be a Protocol, Handler, Validator, or Mediator, etc. On a very practical level features are concrete application code implemented across multiple assemblies. The `Features` object has several ways to specify your application assemblies so that they can be installed in the container. Using Features allows you to specify all your assemblies in one place.

### 6.1.2 FeatureInstaller

FeatureInstallers inherit from `FeatureInstaller` and do the container registration and configuration for a Miruken concept across all your feature assemblies.

#### FromAssemblies(params Assembly[] assemblies)

In this example we pass a comma seperated list of application assemblies into:

```
Features.FromAssemblies()
```

| | |
|---|---|
| typeof(CreateTeam).Assembly | Targets the Example.League assembly. |
| typeof(CreateStudent).Assembly | Targets the Example.School assembly. |

Next, we specify which FeatureInstallers the application needs. This example configures the ConfigurationFactory using the ConfigurationFactoryInstaller, and Validation using the ValidationInstaller.

```
namespace Example.mirukenCastleExamples
{
    using Castle.MicroKernel.Registration;
    using Castle.MicroKernel.Resolvers.SpecializedResolvers;
    using Castle.Windsor;
    using League;
    using Miruken.Castle;
    using Miruken.Validate.Castle;
    using School;

    public class FeaturesFromAssemblies
    {
        public IWindsorContainer Container { get; set; }
```

(continues on next page)

(continued from previous page)

```
        public FeaturesFromAssemblies()
        {
            Container = new WindsorContainer();
            Container.Kernel.Resolver.AddSubResolver(
                new CollectionResolver(Container.Kernel, true));

            Container.Install(
                new FeaturesInstaller(
                    new ConfigurationFeature(), new ValidateFeature())
                    .Use(Classes.FromAssemblyContaining<CreateTeam>(),
                         Classes.FromAssemblyContaining<CreateStudent>())
            );
        }
    }
}
```

### FromAssembliesNamed(params string[] assemblyNames)

The FromAssembliesNamed() method allows you to specify the assembly name of the feature assemblies you want installed into the container.

```
namespace Example.mirukenCastleExamples
{
    using Castle.MicroKernel.Registration;
    using Castle.MicroKernel.Resolvers.SpecializedResolvers;
    using Castle.Windsor;
    using Miruken.Castle;
    using Miruken.Validate.Castle;

    public class FeaturesFromAssembliesNamed
    {
        public IWindsorContainer Container { get; set; }

        public FeaturesFromAssembliesNamed()
        {
            Container = new WindsorContainer();
            Container.Kernel.Resolver.AddSubResolver(
                new CollectionResolver(Container.Kernel, true));

            Container.Install(
                new FeaturesInstaller(
                    new ConfigurationFeature(), new ValidateFeature())
                        .Use(Classes.FromAssemblyNamed("Example.League"),
                             Classes.FromAssemblyNamed("Example.School"))
            );
        }
    }
}
```

### InDirectory(AssemblyFilter filter)

The InDirectory() method allows you to specify an AssemblyFilter. An AssemblyFilter takes the string name of a directory and a filter predicate to allow only the assemblies you intend.

```
namespace Example.mirukenCastleExamples
{
    using Castle.MicroKernel.Registration;
    using Castle.MicroKernel.Resolvers.SpecializedResolvers;
    using Castle.Windsor;
    using Miruken.Castle;
    using Miruken.Validate.Castle;

    public class FeaturesInDirectory
    {
        public IWindsorContainer Container { get; set; }

        public FeaturesInDirectory()
        {
            Container = new WindsorContainer();
            Container.Kernel.Resolver.AddSubResolver(
                new CollectionResolver(Container.Kernel, true));

            Container.Install(
                new FeaturesInstaller(
                    new ConfigurationFeature(), new ValidateFeature())
                    .Use(Classes.FromAssemblyInDirectory(new AssemblyFilter("")
                        .FilterByName(x => x.Name.StartsWith("Example."))))
            );
        }
    }
}
```