T, Earl Grey, Hot: Generics in .NET

An overview of generics by JeremyBytes.com

Overview

Generics give us a way to increase type-safety of our types and methods while still keeping them extensible and reusable. Most C# developers have worked with generics to a certain extent, and we've seen the "T" in angle brackets (or "of T" in parentheses for VB.NET) in a number of classes in the base class library (like List<T>). But generics go way beyond adding flexibility to built-in types; we can use them to create our own classes and methods that take advantage of this great framework feature.

We got generics way back in .NET 2.0, and they really were transformational at the time (now we're hopefully used to seeing them). By making our code type-safe, we are more likely to catch errors at compile time and also avoid strange behavior that might come from casting objects to different types. Our code also becomes more extensible -- it can work with types that it may not have been originally intended to work with.

Generics also offer us some performance enhancements (which, granted, seem fairly minor when talking about the processing resources of modern devices) by reducing casting calls and boxing/unboxing of value types.

We'll take a look at how generics are used in the base class library (by comparing generic and non-generic collections), and then we'll add generics to our own code (interfaces, classes, and methods) to take advantage of these benefits. Along the way, we'll learn some details (such as "default" and generic constraints) that can make our code useful in a variety of situations.

What are Generics?

To start, here's a definition of Generics (from the MSDN documentation):

> *Generics are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types that they store or use. A generic collection class might use a type parameter as a placeholder for the type of objects that it stores; the type parameters appear as the types of its fields and the parameter types of its methods. A generic method might use its type parameter as the type of its return value or as the type of one of its formal parameters.*

Let's break this down. First, "Generics are classes, structures, interfaces and methods…" This means that we can apply generics to types (class / struct / interface) and methods. We cannot apply generics to other parts of our type (such as properties, indexers, and events); however, we can use the generic

©Jeremy Clark 2012

parameters declared by the enclosing type. We'll take a look at what this means in just a bit (when we talk about generic interfaces).

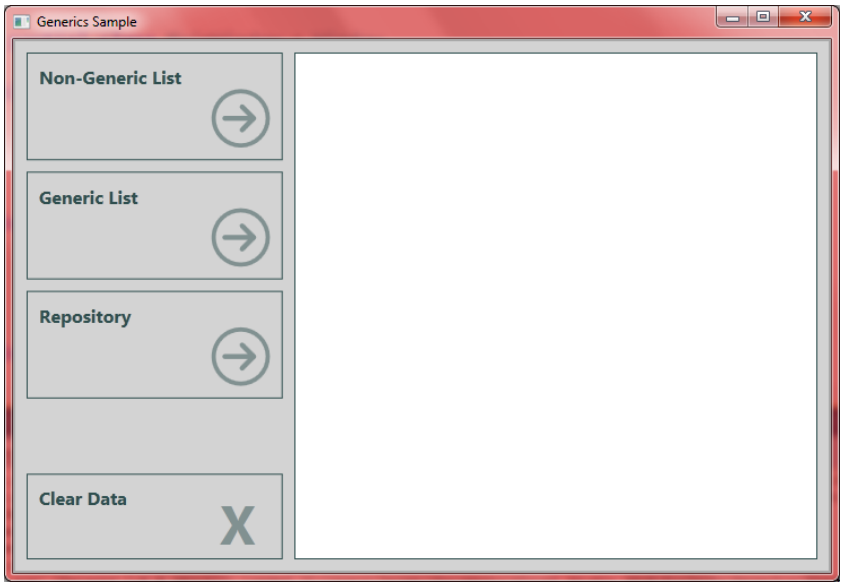
Next, "that have placeholders (type parameters) for one or more of the types that they store or use." The placeholder is the "<T>" (although it doesn't have to be "T" as we'll see). When we use the type or method, we substitute the actual type that we want to work with – whether it is integer, string, DateTime, or some custom type that we have built.

Finally, the definition references generic collections and methods. We'll see both of these as we work through some samples. So let's get started with some generic collections!

Generic and Non-Generic Collections

We'll start by comparing generic and non-generic collections that are offered in the .NET base class library (BCL). You can download the source code for the application from the website: http://www.jeremybytes.com/Downloads.aspx. The sample code we'll be looking at is built using .NET 4 and Visual Studio 2010 (however, everything will work with .NET 3.5 or .NET 4.5). The download consists of a single solution that has multiple projects. Two versions are included: a "starter" solution (if you want to follow along) as well as the "completed" code. To start with, we'll be using just 2 of the projects; we'll add features from the others as we get deeper into the topic.

The `Generics.UI` project is a WPF application that contains our user interface. It includes `MainWindow.xaml` with the following UI:



4 buttons and a list box – if you're interested in the layout of the application, you can take a look at "Metrocizing XAML" (Parts 1 and 2) available here: http://www.jeremybytes.com/Downloads.aspx#MX. (Note: this was written while "Metro" was still being used as the name of the UI style.) The set of

articles describes the styles, data templates, and control templates (in the `App.xaml` file) and also touches on `Converters.cs`.

If we look at the code-behind (`MainPage.xaml.cs`), we find the following placeholders:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void NonGenericButton_Click(object sender, RoutedEventArgs e)
    {
    }

    private void GenericButton_Click(object sender, RoutedEventArgs e)
    {
    }

    private void RepositoryButton_Click(object sender, RoutedEventArgs e)
    {
    }

    private void ClearButton_Click(object sender, RoutedEventArgs e)
    {
        PersonListBox.Items.Clear();
    }
}
```

The only code we have so far (`ClearButton_Click`) clears out our list box.

Switching over to the `Generics.Common` project, we find a `Person` class which we will use as our primary data type. It consists of 4 public properties:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime StartDate { get; set; }
    public int Rating { get; set; }
}
```

Finally, we have a `People` static class that provides us with 2 static methods, one that returns a non-generic collection (`ArrayList`) and one that returns a generic collection (`List<Person>`).

```
public static class People
{
    public static ArrayList GetNonGenericPeople() ...

    public static List<Person> GetGenericPeople() ...
}
```

These methods simply generate some hard-coded data; you can look at the code for details.

A Non-Generic Collection

Before .NET 2.0, we had the `ArrayList` class. The `ArrayList` is a step above using a standard array because it handles dynamically increasing the size of the collection as required. In addition, it implements the `IList` interface which gives us methods like `Add`, `Remove`, `Contains`, `IndexOf`, and others that make the collection easy to work with in code.

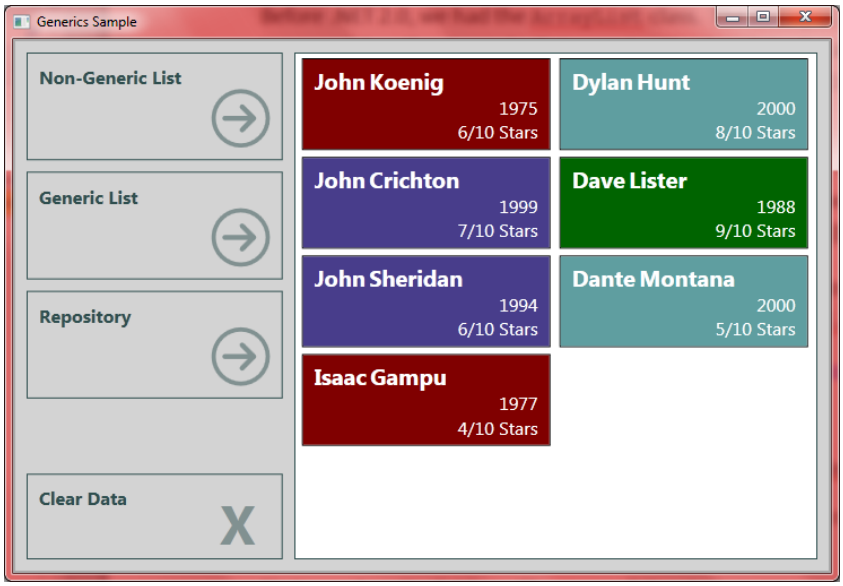
But what can you put into an `ArrayList`? The `ArrayList` stores the items as type `object`. Since pretty much everything in .NET descends from object, this means we can put whatever we want into the list. In our case, we are putting `Person` objects into the list (but the list doesn't know that).

So, let's use our non-generic `ArrayList`. We'll add the following code to our `NonGenericButton_Click` handler in `MainWindow.xaml.cs`:

```
private void NonGenericButton_Click(object sender, RoutedEventArgs e)
{
    PersonListBox.Items.Clear();

    ArrayList people = People.GetNonGenericPeople();
    foreach (object person in people)
        PersonListBox.Items.Add(person);
}
```

First, we clear the list box. Then we call the static `GetNonGenericPeople` method to get our populated `ArrayList`. Then we use a `foreach` loop to populate the list box. Notice that our `foreach` loop uses the type `object` for the collection item (`person`). As a note, we could use the `var` keyword in our `foreach` loop (instead of `object`), but I wanted to make it clear that we are getting an `object` from our collection. If we run the application and click the "Non-Generic List" button, we get the following:



©Jeremy Clark 2012

Now, you might be wondering how the UI can properly layout the `object` items in the UI – after all, it seems like it really wants to lay out `Person` items. The answer to this lies in the WPF data binding model. It turns out that the data-binding model is extremely forgiving. When we data bind to the `FirstName` property, the UI simply looks for that property on the item and binds to it if it finds it. If it doesn't find that property, then it ignores the binding. Since our `object` has the expected properties, the layout works just fine.

A Generic Collection

When generics were added to .NET 2.0, we also got a number of generic collections. Today, we're using `List<T>` (pronounced "List of T" when you say this out loud). As noted earlier, VB.NET uses the syntax `List(Of T)` which, curiously enough, is also pronounced "List of T".

`List<T>` also implements the `IList` interface, so we get the same methods (`Add`, `Remove`, `Contains`, `IndexOf`, and others) that we have in `ArrayList`. The big difference is the generic parameter: `T`. When we use this class, we substitute `T` with whatever type we want. This constrains the list so that it can only accept items of that particular type (we'll see what this means in just a moment).

Since we are working with items of type `Person`, we declare `List<Person>`.

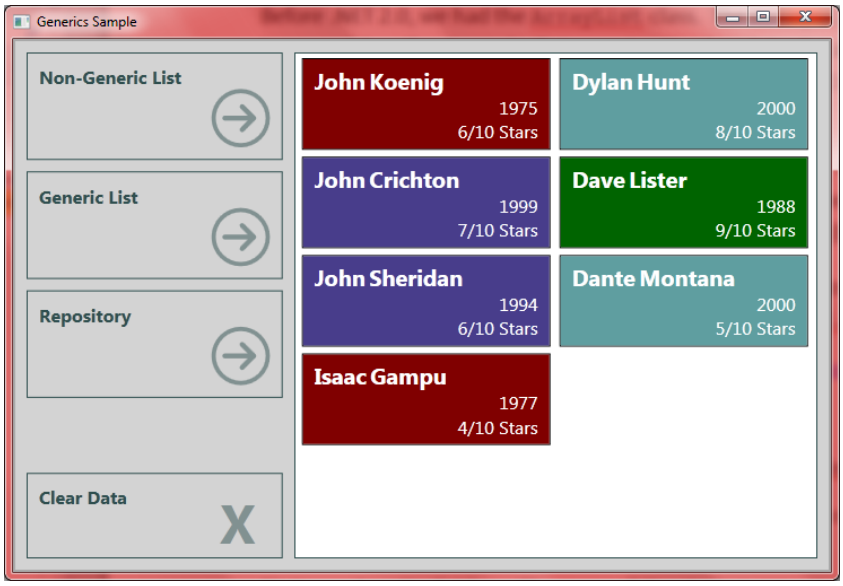
So, let's use our generic `List<T>`. We'll add the following code to our `GenericButton_Click` handler in `MainWindow.xaml.cs`:

```
private void GenericButton_Click(object sender, RoutedEventArgs e)
{
    PersonListBox.Items.Clear();

    List<Person> people = People.GetGenericPeople();
    foreach (Person person in people)
        PersonListBox.Items.Add(person);
}
```

This code is very similar to our non-generic code. The primary difference is that our `foreach` loop is getting `Person` objects from the collection. As you might imagine, our output is similar to what we had above:

©Jeremy Clark 2012



What's the Difference?

Okay, so you're probably thinking, "We've got two code blocks that do exactly the same thing. So what?" But this is where things get interesting.

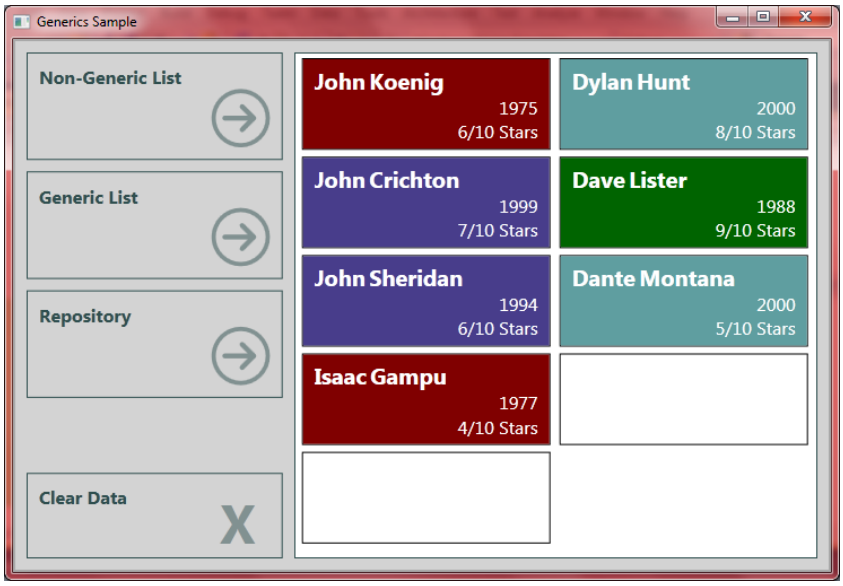
We mentioned that type-safety is a big advantage of generics. First, let's see how our non-generic version handles different types.

Let's add a few lines to our `NonGenericButton_Click` method:

```
private void NonGenericButton_Click(object sender, RoutedEventArgs e)
{
    PersonListBox.Items.Clear();

    ArrayList people = People.GetNonGenericPeople();
    people.Add("Hello");
    people.Add(15);
    foreach (object person in people)
        PersonListBox.Items.Add(person);
}
```

Here, we're adding two more objects to our list. The first is a string, and the second is an integer. Since `ArrayList` works with the `object` type, it accepts these new items. This code compiles just fine. But we get some strange behavior at runtime:



Notice the 2 empty boxes at the bottom of the list. These are our string and integer items. We can check the output window in Visual Studio to see the binding errors:

```
BindingExpression path error: 'StartDate' property not found on 'object' ''String' (H
BindingExpression path error: 'FirstName' property not found on 'object' ''String' (H
BindingExpression path error: 'LastName' property not found on 'object' ''String' (Ha
BindingExpression path error: 'StartDate' property not found on 'object' ''String' (H
BindingExpression path error: 'Rating' property not found on 'object' ''String' (Hash
BindingExpression path error: 'StartDate' property not found on 'object' ''Int32' (Ha
BindingExpression path error: 'FirstName' property not found on 'object' ''Int32' (Ha
BindingExpression path error: 'LastName' property not found on 'object' ''Int32' (Has
BindingExpression path error: 'StartDate' property not found on 'object' ''Int32' (Ha
BindingExpression path error: 'Rating' property not found on 'object' ''Int32' (HashC
```

As noted earlier, the data binding mechanism in WPF is very forgiving. If it doesn't find a property that it is looking for, then it ignores it (after logging it) and moves on. The output window is a great way to debug data binding issues with a WPF application. Since none of the required properties were found on the string or integer objects, we get two empty items in our UI.

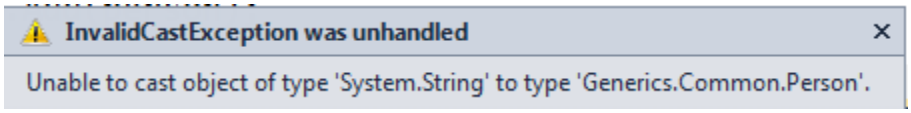
To fix the UI, we might want to ensure that only `Person` objects get to the list box. To do this, we could update our code as follows:

```
private void NonGenericButton_Click(object sender, RoutedEventArgs e)
{
    PersonListBox.Items.Clear();

    ArrayList people = People.GetNonGenericPeople();
    people.Add("Hello");
    people.Add(15);
    foreach (Person person in people)
        PersonListBox.Items.Add(person);
}
```

Here, we've changed the `foreach` loop from `object` to `Person`. This has the effect of doing a cast to type `Person` for each item coming out of the collection.

If we run the code now, we end up with a runtime error:



We could further modify our code to try to protect from this error. But at this point, we're just throwing good code after bad. The much better solution is to use the generic list.

Let's go back to our generic list and try the following code:

```
private void GenericButton_Click(object sender, RoutedEventArgs e)
{
    PersonListBox.Items.Clear();

    List<Person> people = People.GetGenericPeople();
    people.Add("Hello");
    people.Add(15);
    foreach (Person person in people)
        PersonListBox.Items.Add(person);
}
```

This time, we get a compile-time error:

> *"The best overloaded method match for 'System.Collections.Generic.List<Generics.Common.Person>.Add(Generics.Common.Person)' has some invalid arguments."*

The generic list protects us from adding incompatible objects to our list. So rather than getting runtime errors, we get compile time errors. Which would you rather have?

(Note: be sure to remove or comment out these two lines of code before continuing.)

Boxing and Unboxing

Using a generic collection over a non-generic collection can also result in performance improvements. This has to do with how value types are handled in .NET. A value type (struct) is stored on the stack (i.e., the value is stored in the quickly accessible stack memory). A reference type (class) is stored on the heap (i.e., the value is stored in heap memory, and a reference to that memory location is stored on the stack).

Many of the primitive types in .NET are value types. This includes integer, Boolean, character, DateTime, and several other types. All types in .NET (whether value types or reference types) descend from Object. As such, we can cast a value type to an `object`. However, when we cast a value type to an object, the value is "boxed" as a reference type, meaning that the value is put into heap memory and

a reference placed on the stack. When we cast the `object` back to a value type, it is "unboxed", meaning that the value is pulled off the heap and placed into the stack.

This process of boxing and unboxing incurs some overhead. If we use an `ArrayList` (or other non-generic collection) to store value types (such as integers or other numeric types), each time we place a value into the list, it is boxed; and each time we pull a value from the list, it is unboxed. For collections with many members, the result is that a large percentage of the processing time is spent on the boxing/unboxing procedure.

When we use a generic collection with a value type (such as `List<int>`), we do not have to go through the boxing/unboxing process. The result is that generic collections of value types run much more efficiently than their non-generic counterparts.

Reuse and Extensibility

So, we've seen how generics can add type-safety to our code. This helps us catch errors at compile-time instead of runtime and makes our code more robust. So far, we have just used classes that are built into the .NET BCL. We can use generics in our own classes to get the same advantages. Today, we'll look at how generics can help with reuse and extensibility.

The Repository Pattern

If you're writing business applications, you probably have a mechanism set up for CRUD (Create, Read, Update, Delete) operations. The Repository Pattern allows us to build a common interface for these operations so that our application is not tightly coupled to the underlying data store. If you want more information on the Repository Pattern and using it to create different repositories (for a SQL database, a CSV file, and a web service), see "IEnumerable, ISaveable, IDontGetIt: Understanding .NET Interfaces" (available for download here: http://www.jeremybytes.com/Downloads.aspx#INT).

To take a look at generics, we'll use the repository pattern, but we'll restrict ourselves to a single implementation for simplicity.

A Person Repository

In the `Generics.Common` project, we have a folder for interfaces. Let's take a look at `IPersonRepository` to see how the repository pattern works.

```
public interface IPersonRepository
{
    IEnumerable<Person> GetPeople();

    Person GetPerson(string lastName);

    void AddPerson(Person newPerson);

    void UpdatePerson(string lastName, Person updatedPerson);

    void DeletePerson(string lastName);
```

```
        void UpdatePeople(IEnumerable<Person> updatedPeople);
    }
```

We have a number of methods specified here. The first, (`GetPeople`) allows us to get a list of all people in our data store. This is the method that we'll be using in our code. The other methods let us get an individual person and to add, update, and delete person objects. The last method lets us replace our entire collection with a new one.

Notice that we are using the `Person` type in various places (as parameters and return values). Another thing to note is that we are using `string lastName` as our primary key – the way that we locate an existing item.

A Person Repository Implementation

We're working with a web service to handle all of our actual data store operations. These files are located in the `Person.Service` project. This is a standard WCF service that exposes a number of methods. We will use these methods in our repository implementation.

The repository itself is located in the `Generics.Repository` project (the `PersonService` Repository class in the `PersonRepository` folder). Here's the implementation (some methods have been truncated for readability):

```
public class PersonServiceRepository : IPersonRepository
{
    private PersonServiceClient proxy;

    public PersonServiceRepository()
    {
        proxy = new PersonServiceClient();
    }

    public IEnumerable<Person> GetPeople()
    {
        return proxy.GetPeople();
    }

    public Person GetPerson(string lastName)...
    {
        return proxy.GetPerson(lastName);
    }

    public void AddPerson(Person newPerson)...

    public void UpdatePerson(string lastName, Person updatedPerson)...

    public void DeletePerson(string lastName)...

    public void UpdatePeople(IEnumerable<Person> updatedPeople)...
}
```

This `PersonServiceRepository` class implements `IPersonRepository` by including all of the methods from the interface. The actual implementation is quite simple in this case because we are

passing the call through to the web service. (Check "IEnumerable, ISaveable, IDontGetIt" (mentioned above) for some examples of repositories that use different data stores.

A Product Repository

Now that we've seen a repository interface for the `Person` class, let's think about a repository for the `Product` class. `Product.cs` contains our `Product` type:

```
public class Product
{
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public string Category { get; set; }
}
```

And the repository in `IProductRepository.cs`:

```
public interface IProductRepository
{
    IEnumerable<Product> GetProducts();

    Product GetProduct(int productId);

    void AddProduct(Product newProduct);

    void UpdateProduct(int productId, Product updatedProduct);

    void DeleteProduct(int productId);

    void UpdateProducts(IEnumerable<Product> updatedProducts);
}
```

We have methods that are very similar to those in the `Person` repository. The differences include the method names (`AddProduct` vs. `AddPerson`), the item type (`Product` vs. `Person`), and the primary key (`int productId` vs. `string lastName`).

As you can imagine, if we had more types (Customer, Order, Payment, etc.) we would end up with quite a few very similar interfaces. Wouldn't it be great if we could combine these into a single re-usable interface? That's exactly how generics can help us.

A Generic Repository

Since the only differences between the repositories are the method names, item type, and primary key, we can figure out a way to combine these separate interfaces into a single generic interface.

The method names are easy. Instead of "`GetPeople`" and "`GetProducts`" we use a more generalized "`GetItems`". We'll extend this to the other method names as well.

But what about the types? Let's go back to the generics definition that we started with.

> *Generics are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types that they store or use.*

©Jeremy Clark 2012

For the types that need to vary, we have the item type (`Person` or `Product`) and the primary key (string or integer). Let's look at the code and then examine it more closely (in `Generics.Common IRepository.cs`):

```
public interface IRepository<T, TKey>
{
    IEnumerable<T> GetItems();

    T GetItem(TKey key);

    void AddItem(T newItem);

    void UpdateItem(TKey key, T updatedItem);

    void DeleteItem(TKey key);

    void UpdateItems(IEnumerable<T> updatedItems);
}
```

First, notice our interface declaration: `IRepository<T, TKey>`. This specifies that we will have 2 type parameters. `T` is our item type; `TKey` is the type of our primary key.

A quick note about naming type parameters: `T` (for Type) is used by convention if there is a solitary type parameter or as the primary type parameter (if there are multiple parameters). This naming convention is not required, but most developers are used to seeing `T`. For additional parameters, we try to use meaningful names but still prefix with the letter T (like our `TKey` example). This is to add readability and hopefully to give the developers using our generic type a clue to how to use it properly.

If we compare `IRepository<T, TKey>` to `IPersonRepository`, we see that all usages of "`Person`" have been replaced by "`T`" and all usages of "`string lastName`" have been replaced by "`TKey key`".

So what does an implementation look like?

A Generic Repository Implementation

We already have an implementation for the Person repository in our `Generics.Repository` project (called `GenericPersonServiceRepository`):

```
public class GenericPersonServiceRepository : IRepository<Person, string>
{
    private PersonServiceClient proxy;

    public IEnumerable<Person> GetItems()
    {
        return proxy.GetPeople();
    }

    public Person GetItem(string lastName)
    {
        return proxy.GetPerson(lastName);
    }
```

```
        public void AddItem(Person newPerson)...

        public void UpdateItem(string lastName, Person updatedPerson)...

        public void DeleteItem(string lastName)...

        public void UpdateItems(IEnumerable<Person> updatedPeople)...
    }
```

First, notice how we have declared the interface: `IRepository<Person, string>`. This is how we substitute in the types that are important for this particular repository. Everywhere we saw "`T`" in the interface, we have "`Person`"; and everywhere we saw "`TKey`", we have "`string`".

Next, notice the names of the method parameters. Even though the interface specified generic names like "`key`" and "`updatedItem`", we have used "`lastName`" and "`updatedPerson`" for clarity in our class. This is a feature of interfaces: the method signatures need to match (meaning the parameter type(s) and return type) but the actual parameter names can be whatever we like.

The great thing is that we can now use exactly the same interface for our `Product` repository.

Another Generic Repository

In our `Generics.Repository` project, the `GenericProductServiceRepository` is still an empty class:

```
    public class GenericProductServiceRepository
    {

    }
```

Let's start by adding the interface declaration:

```
    public class GenericProductServiceRepository : IRepository<Product, int>
    {

    }
```

This time, we specified "`Product`" as our item type and "`int`" as our primary key type. Now, we can right-click on "`IRepository`" and select "Implement Interface" and then "Implement Interface" again, and Visual Studio will stub out all of our methods:

```
    public class GenericProductServiceRepository : IRepository<Product, int>
    {
        public IEnumerable<Product> GetItems()
        {
            throw new NotImplementedException();
        }

        public Product GetItem(int key)
        {
            throw new NotImplementedException();
        }
```

```
        public void AddItem(Product newItem)...

        public void UpdateItem(int key, Product updatedItem)...

        public void DeleteItem(int key)...

        public void UpdateItems(IEnumerable<Product> updatedItems)...
    }
```

You can see that the implementation includes the types in all the right places. At this point, we can rename the method parameters if we like (but it's not necessary).

All we would need to do is complete the implementation by replacing the "`throw new NotImplementedException()`" with our own code. The implementation is not included in the sample project, but would look very similar to the `Person` repository.

Benefits

As you can see, we now have two custom repositories that share the same generic interface definition. If we had other types (Customer, Order, Payment, etc.), they could all reuse this same interface. The main benefit of all repositories using the same interface is that all of the application code that accesses the repositories will do so in the same way.

Regardless of the type of object we are retrieving, we will always use the same method name (`GetItems`), and we get back a strongly-typed collection. This makes our application code consistent in appearance and functionality.

Generic Methods

So far, we have looked at adding type parameters to classes. But we can also use type parameters to create generic methods. Let's start by looking at a non-generic example.

A Static Factory

For our specific application, we want to use late-binding – meaning, we don't want the application to know what concrete repository we are using. We want the application code to only reference the repository through the interface. Then we will create a factory method that will instantiate a concrete repository based on configuration information. (For a more detailed discussion of this, see "IEnumerable, ISaveable, IDontGetIt: Understanding .NET Intefaces" (here's the link again for convenience: http://www.jeremybytes.com/Downloads.aspx#INT).

Back in `Generics.Common`, we have a folder for our factories. Let's look at `RepositoryFactory.cs`:

```
public static class RepositoryFactory
{
    public static IPersonRepository GetPersonRepository()
    {
        string configString =
            ConfigurationManager.AppSettings["IPersonRepository"];
        Type resolvedType = Type.GetType(configString);
        object obj = Activator.CreateInstance(resolvedType);
        IPersonRepository rep = obj as IPersonRepository;
        return rep;
    }

    public static IProductRepository GetProductRepository()
    {
        string configString =
            ConfigurationManager.AppSettings["IProductRepository"];
        Type resolvedType = Type.GetType(configString);
        object obj = Activator.CreateInstance(resolvedType);
        IProductRepository rep = obj as IProductRepository;
        return rep;
    }
}
```

This static class contains two static methods – one to get the `Person` repository and one to get the `Product` repository. To figure out what concrete type to use, it references the `app.config` file. Here are those settings (in the `Generics.UI` project):

```
    <!-- Resolved by RepositoryFactory.GetRepository -->
    <add key="IPersonRepository"
         value="Generics.Repository.PersonRepository.PersonServiceRepository,
Generics.Repository, Version=1.0.0.0, Culture=neutral"/>

    <add key="IProductRepository"
value="Generics.Repository.ProductRepository.ProductServiceRepository,
Generics.Repository, Version=1.0.0.0, Culture=neutral"/>
```

Note: the line-breaks are a little different if you look at the actual config file. So, how does the factory method work? Let's take it one line at a time.

```
        string configString =
            ConfigurationManager.AppSettings["IPersonRepository"];
```

This gets the "`value`" setting from the configuration file. The value is the fully-qualified type name of the class we want to use. This includes the namespace and type-name (`…PersonServiceRepository`), the assembly name (`Generics.Repository`), plus the version and culture of the assembly. (If you are using strongly-named assemblies, then the public key would be included here, too.)

```
        Type resolvedType = Type.GetType(configString);
```

This resolves the configuration string into an actual `PersonServiceRepository` type that we can use.

```
        object obj = Activator.CreateInstance(resolvedType);
```

This uses the `Activator` to instantiate our selected type. In this case, it will create a `PersonServiceRepository` object. The return type of this method is of type `object`.

```
            IPersonRepository rep = obj as IPersonRepository;
```

The next line casts the object created by the `Activator` into the type we are expecting – the interface type `IPersonRepository`.

```
            return rep;
```

Finally, we return our fully instantiated `IPersonRepository`.

Using the Factory Method

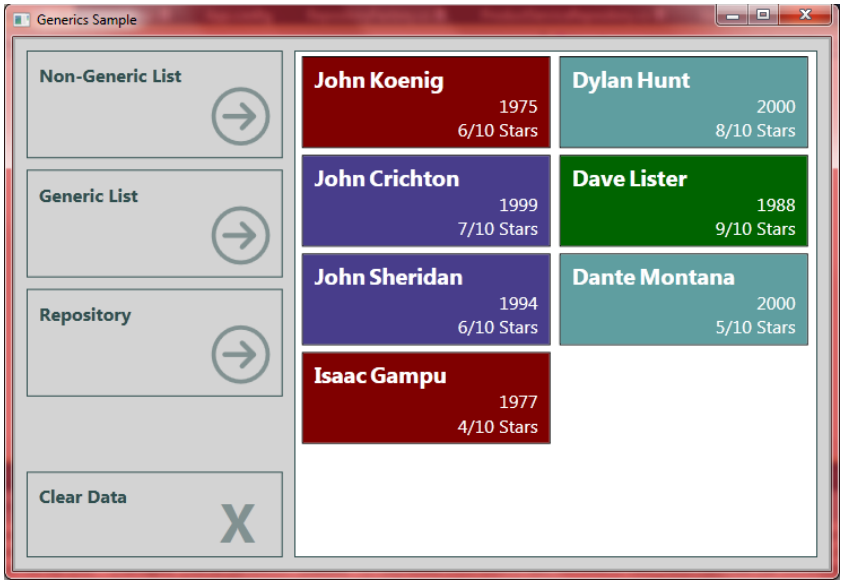
Let's actually put this factory method to work. In our `Generics.UI` project, we'll flip back to the `MainPage.xaml.cs` file and implement our last event handler: `RepositoryButton_Click`.

```
        private void RepositoryButton_Click(object sender, RoutedEventArgs e)
        {
            IPersonRepository repo = RepositoryFactory.GetPersonRepository();
            var people = repo.GetPeople();

            foreach (var person in people)
                PersonListBox.Items.Add(person);
        }
```

First, we use the `RepositoryFactory` to get our Person repository. Then we use the repository to get the list of people. Then we loop through the people and populate the list box. Since we are using an interface reference (`IPersonRepository`) instead of a concrete type, we can change to a different concrete repository type (such as one that uses a SQL server) without needing to change this code.

Our output is just as we would expect:



©Jeremy Clark 2012

One thing to note: our Repository factory is using the non-generic `IPersonRepository`, but we could have just as easily have written it to use the generic `IRepository`. The result would be the same.

A Generic Factory

If we go back to our `RepositoryFactory` and compare the static methods `GetPersonRepository` and `GetProductRepository`, we see that most of the code is identical. The two differences include the type we are creating (`IPersonRepository` vs. `IProductRepository`) and the key we are using for the configuration file. This sounds like something we can easily refactor.

Just like with our interface, we can use a generic type parameter to consolidate this code into a single method. In this case, we'll be using a type parameter with a method instead of a class.

A few naming differences: this code was originally presented as low-budget inversion of control (IoC) in "IEnumerable, ISaveable, IDontGetIt" (which has been mentioned several times so far – maybe you should go read this). Since the generic factory method is a lot like an IoC container, we use a syntax that is similar to the pre-built containers that you can use (and you should use one of these rather than building your own – if you want more information on these containers, see "Dependency Injection: A Practical Introduction" also available on the website).

So, instead of naming the class "`RepositoryFactory`", we have named it "`Container`" (but it is still a static class). And instead of naming the method "`Get…Repository`", we have named it "`Resolve`".

Let's look at the code in `Container.cs` of the `Generics.Common` project:

```
public static class Container
{
    public static T Resolve<T>() where T : class
    {
        string configString =
            ConfigurationManager.AppSettings[typeof(T).ToString()];
        Type resolvedType = Type.GetType(configString);
        object obj = Activator.CreateInstance(resolvedType);
        T rep = obj as T;
        return rep;
    }
}
```

Notice that our Resolve method takes a type parameter (`T`) and uses that as the return type (we'll come back to the "`where`" constraint in just a moment).

The method body is similar. Anywhere we had the type `IPersonRepository`, we now have a `T`. We've had to replace the string reference to the `AppSettings` with something we can get from the type parameter (we'll come back to this in just a moment). When we look at the rest of the methods, there is a straight swap between `IPersonRepository` and `T`.

Using the Generic Factory

Let's update our application so that it uses the generic factory method. The revised `RepositoryButton_Click` looks like this:

```
private void RepositoryButton_Click(object sender, RoutedEventArgs e)
{
    var repo = Container.Resolve<IRepository<Person,string>>();
    var people = repo.GetItems();

    foreach (var person in people)
        PersonListBox.Items.Add(person);
}
```

When calling `Container.Resolve`, we need to pass in the type we want to work with. In our case, we want to use the generic repository which is based on `IRepository<Person, string>` (for the `Person` class). This syntax looks a bit odd since we have nested generics, but once you see this a few times it will start to look normal.

What doesn't look normal is the updated configuration file. Remember that our factory method is using "`typeof(T).ToString()`". Our type is `IRepository<Person,string>`, but when we call "`ToString()`" on this, we get an interesting output:

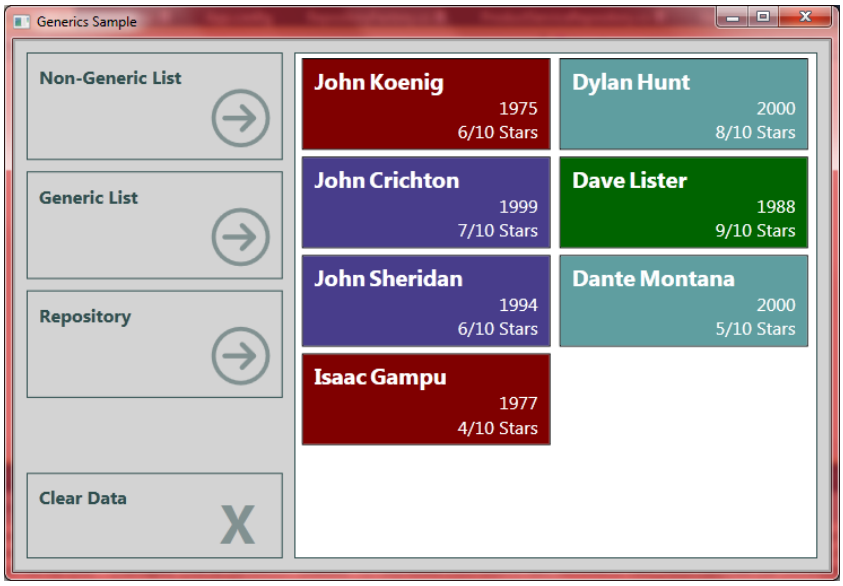
```
Generics.Common.Interface.IRepository`2[Generics.Common.Person,System.String]
```

First we have the namespace. Then we have `IRepository`2` which tells us that we have two type parameters. Inside the brackets, we have the fully-qualified names of the types we specified. So, our entire entry for the configuration item looks like this (again, line-breaks are different from the actual configuration file):

```
<add
key="Generics.Common.Interface.IRepository`2[Generics.Common.Person,System.St
ring]"
value="Generics.Repository.PersonRepository.GenericPersonServiceRepository,
Generics.Repository, Version=1.0.0.0, Culture=neutral"/>
```

The key is different (so that we can extract it from the generic type parameter), but the value is the same as above. One change is that we specify `GenericPersonServiceRepository` (instead of `PersonServiceRepository`), but the rest of the assembly information is the same.

When we run the application, we get the expected output:



Generic Constraints

Let's head back to our generic factory method:

```
public static T Resolve<T>() where T : class
```

So, what does the "`where`" mean? This is a generic constraint. This limits what can be used as the generic type parameter. In this case, we specify that `T` needs to be a `class` (a reference type). We need this constraint for the `Activator.CreateInstance` method. This method call is expecting an object with a default constructor. A value type (such as a `struct`) does not have a default constructor. If we were to pass a value type to `CreateInstance`, the call would fail.

Using the `where` keyword, we can specify a number of constraints:

- `class` – a reference type
- `struct` – a value type
- `new()` – a type with a parameterless constructor
- base class – a type that descends from a specified class
- interface – a type that implements a specified interface

We can even combine these (with commas) to specify multiple constraints. If we have more than one type parameter, we can specify different constraints for each type.

When looking at this list, you might wonder why we didn't use the `new()` constraint rather than the `class` constraint since we noted that `CreateMethod` just needs a parameterless constructor. The answer is that we are also using the `as` keyword to cast the object to our type, and `as` will only operate on a class.

By using these constraints, we make sure that if we have any special requirements on our type parameters, the compiler (and developer) will know about it. If we were to use a value type parameter with our factory method (instead of the required reference type), then the compiler would throw an error because of the constraint. Without the constraint, we could use a value type, compile successfully, and then get a runtime error at the `CreateInstance` call. So, constraints give us a chance to add extra type-safety to our generic code.

One last thing before we leave our generic factory. We can actually combine the last three lines of our method. This changes the code from this:

```
public static T Resolve<T>() where T : class
{
    string configString =
        ConfigurationManager.AppSettings[typeof(T).ToString()];
    Type resolvedType = Type.GetType(configString);
    object obj = Activator.CreateInstance(resolvedType);
    T rep = obj as T;
    return rep;
}
```

To this:

```
public static T Resolve<T>() where T : class
{
    string configString =
        ConfigurationManager.AppSettings[typeof(T).ToString()];
    Type resolvedType = Type.GetType(configString);
    return Activator.CreateInstance(resolvedType) as T;
}
```

In the last statement, we create the instance, cast it to `T`, and then return the value all in one statement. Our code is now more compact but a bit more difficult to debug if something goes wrong.

Default Keyword

One last thing we should mention regarding generics is the "`default`" keyword. Since the type parameters can be either reference types or value types (assuming that we do not have a constraint), it may be difficult if we need to "reset" a value.

For example, let's say that we have a generic class with a single type parameter (`T`). The class contains a local variable of type `T`, let's say "`T _currentItem`". In the constructor, we want to reset this parameter. But how do we do that? We can't just set it to `null` because if it is a value type (like integer), then the assignment to `null` will fail.

This is where `default` comes in. We use it like this:

```
_currentItem = default;
```

If `_currentItem` is a reference type, then it is set to `null`. If it is a value type, then it is set to bitwise zero.

Wrap Up

Generics are an extremely powerful feature in .NET. We have seen how generics can add type-safety to our code, they can make our code re-usable and extensible, and they can move a number of errors from runtime to compile-time. There are still plenty of other areas to explore in the generics world. If you are pining for more, you can look up covariance and contravariance in generics. This lets us control whether we can use descendent types or parent types when the generic parameter types are specified.

Start taking full advantage of generics and you can see these benefits in your own code.

Happy coding!

©Jeremy Clark 2012