Generics

By Aagamuri Sridhar

An article on generics, gives clear picture about generics with examples

Beginner
C#
Windows, .NET (.NET 2.0)
Win32, VS
Dev
Posted 26 Feb 2004

Introduction

Parametric Polymorphism is a well-established programming language feature. Generics offers this feature to C#.

The best way to understand generics is to study some C# code that would benefit from generics. The code stated below is about a simple `Stack` class with two methods: `Push ()` and `Pop ()`. First, without using generics example you can get a clear idea about two issues: a) Boxing and unboxing overhead and b) No strong type information at compile type. After that the same `Stack` class with the use of generics explains how these two issues are solved.

Example Code

Code without using generics:

```
public class Stack
{
   object[] store;
   int size;
   public void Push(object x) {...}
   public object Pop() {...}
}
```

Boxing and unboxing overhead:

You can push a value of any type onto a stack. To retrieve, the result of the `Pop` method must be explicitly cast back. For example if an integer passed to the `Push` method, it is automatically boxed. While retrieving, it must be unboxed with an explicit type cast.

```
Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop(); //unboxing with explicit int casting
```

Such boxing and unboxing operations add performance overhead since they involve dynamic memory allocations and run-time type checks.

No strong Type information at Compile Time

Another issue with the Stack class: It is not possible to enforce the kind of data placed on a stack. For example, a string can be pushed on a stack and then accidentally cast to the wrong type like integer after it is retrieved:

```
Stack stack = new Stack();
stack.Push("SomeName");   //pushing the string
int i = (int)stack.Pop();  //run-time exception will be thrown at this point
```

The above code is technically correct and you will not get any compile time error. The problem does

not become visible until the code is executed; at that point an `InvalidCastException` is thrown.

Code with generics

In C# with generics, you declare class `Stack <T> {...},` where `T` is the type parameter. Within class `Stack <T>` you can use `T` as if it were a type. You can create a `Stack` as `Integer` by declaring `Stack <int>` or `Stack` as `Customer` object by declaring `Stack<Customer>.` Simply your type arguments get substituted for the type parameter. All of the `T`s become ints or Customers, you don't have to downcast, and there is strong type checking everywhere.

```
public class Stack<T>
{  // items are of type T, which is kown when you create the object
   T[] items;
   int count;
   public void Push(T item)    {...}
  //type of method pop will be decided when you creat the object
   public T Pop()
   {...}
}
```

In the following example, int is given as the type argument for `T`:

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int i = stack.Pop();
```

The `Stack<int>` type is called a constructed type. In the `Stack<int>` type, every occurrence of `T` is replaced with the type argument int. The Push and Pop methods of a `Stack<int>` operate on `int` values, making it a compile-time error to push values of other types onto the stack, and eliminating the need to explicitly cast values back to their original type when they are retrieved.

You can use parameterization not only for classes but also for interfaces, structs, methods and delegates.

```
//For Interfaces
interface IComparable <T>

//for structs
struct HashBucket <K,D>

//for methods
static void Reverse <T> (T[] arr)

//for delegates
delegate void Action <T> (T arg)
```

Inside the CLR

When you compile `Stack<T>,` or any other generic type, it compiles down to IL and metadata just like any normal type. The IL and metadata contains additional information that knows there's a type parameter. This means you have the type information at compile time.

Implementation of parametric polymorphism can be done in two ways 1. Code Specialization: Specializing the code for each instantiation 2. Code sharing: Generating common code for all instantiations. The C# implementation of generics uses both code specialization and code sharing as explained below.

At runtime, when your application makes its first reference to Stack <int>, the system looks to see if anyone already asked for `Stack <int>.` If not, it feeds into the JIT the IL and metadata for `Stack <T>` and the type argument int. The .NET Common Language Runtime creates a specialized copy of the native code for each generic type instantiation with a value type, but shares a single copy of the native code for all reference types (since, at the native code level, references are just pointers with the same representation).

In other words, for instantiations those are value types: such as `Stack <int>, Stack <long>,`

`Stack<double>, Stack<float>` CLR creates a unique copy of the executable native code. So `Stack<int>` gets its own code. `Stack<long>` gets its own code. `Stack <float>` gets its own code. `Stack <int>` uses 32 bits and `Stack <long>` uses 64 bits. While reference types, `Stack <dog>` is different from `Stack <cat>,` but they actually share all the same method code and both are 32-bit pointers. This code sharing avoids code bloat and gives better performance.

To support generics, Microsoft did some changes to CLR, metadata, type-loader,language compilers, IL instructions and so on for the next release of Visual Studio.NET (codenamed Whidbey).

What you can get with Generics

Generics can make the C# code more efficient, type-safe and maintainable.

- **Efficiency:** Following points states that how performance is boosted.
  1. Instantiations of parameterized classes are loaded dynamically and the code for their methods is generated on demand [Just in Time].
  2. Where ever possible, compiled code and data representations are shared between different instantiations.
  3. Due to type specialization, the implementation never needs to box values of primitive types.
- **Safety:** Strong type checking at compile time, hence more bugs caught at compile time itself.
- **Maintainability:** Maintainability is achieved with fewer explicit conversions between data types and code with generics improves clarity and expressivity.

Conclusion

Generics gives better performance, type safety and clarity to the C# programs. Generics will increase program reliability by adding strong type checking. Learning how to use generics is straightforward, hopefully this article has inspired you to look deeper into how you can use them.

About Aagamuri Sridhar

Sridhar Aagamuri
MCSD.NET
Wipro Technologies
sridhar.aagamuri@wipro.com

Click here to view Aagamuri Sridhar's online profile.

Discussions and Feedback

14 comments have been posted for this article. Visit **http://www.codeproject.com/csharp/Generics.asp** to post and view comments on this article.