Review of C#2 Features – Part I

Nico Ludwig (@ersatzteilchen)

TOC

- Review of C#2 Features – Part I
  - Separate Getter/Setter Property Access Modifiers
  - Static Classes
  - Dissection of the .NET 1 Type System
  - Generic Types

- Sources:
  - Jeffrey Richter, Applied Microsoft .NET Framework Programming
  - Jon Skeet, CSharp in Depth

Let's take a look at Properties in C#1

```
private int _age;
public int Age // Public property Age.
{
      get { return _age; }
      set
      {
            if (CheckAge(_age))
            {
                  _age = value;
            }
      }
}
```

- So far, so good. What if we want the setter to be only privately accessible?
  - The setter could be removed, but we don't want to loose *CheckAge*()'s call!
  - A special private setter function could be written, but this not the .NET way!

- Couldn't we simply make the setter private?
  - No! That is exactly the missed feature in C#1.
  - We can do so in C#2!

Separate getter/setter Property Access in C#2

```
private int _age;
public int Age // Public property Age with private setter.
{
      get { return _age; }
      private set
      {
            if (CheckAge(_age))
            {
                  _age = value;
            }
      }
}
```

- Just to confirm that we get this right:
  - The property's access modifier (public in this case) is applied to the getter/setter.
  - For either the getter or the setter we can declare an access type, being more restricted than the property's one.
  - This feature is available for static properties and indexers as well.
  - It is not available for event adder/remover!

Static Classes

- Often we have to create utility functions, which aren't methods in a sense.
  - We can't associate them with a specific type.
- To manage such functions we'll write a new type with following traits:
  - All its functions are static (except a private constructor (ctor)).
  - The type derives directly from *Object*.
  - There's no state at all.
  - There are no visible ctors.
  - The type is sealed if the developer remembers to do so.
- But we have to remember well all these traits, else our type may be misused.
- C#2 adds static classes to the language, which force all these traits.

- In VB the behavior of static classes is available as Modules.
- static nested class in Java have nothing to do with static classes in C#.

Static Classes in Action

```
public static class MyUtilities // A static class.
{
    // Does input match against rexToMatch?
    public static bool StringMatches(string input, string rexToMatch)
    {
        return Regex.IsMatch(input, rexToMatch);
    }
}
```

- Important features of static classes:
  - Acts implicitly abstract and sealed.
  - Can neither derive from any type, nor implement any interface.
  - Can't include non-static members, incl. ctors.
  - Can't include any operators.
  - Can have other static members (methods, fields and properties) as well.
  - Can have private static members as well.
- Can't be used as declared types, e.g. as parameter type.

The Type System in .NET 1 (presented in C#1)

- Explicitly typed – i.e. typenames must be explicitly written at their declaration:

```
double sum = 5.2; // Type name "double" must be explicitly written for the variable sum.
```

- Safely typed – i.e. unrelated types can't fool the compiler.
  - Cross casting and other pointer stuff is generally illegal.

- Statically typed – i.e. variables' types are determined at compile time:

```
int count = 42; // count is of type int, we can only perform int operations on it!
```

- Sorry? But the same variable can hold different sub type objects at run time!
  - Indeed! We can define variables that may hold e.g. any derived type:

```
Object o = new Form(); // Type Form is derived from type Object.
```

  - .NET differs the static (*Object*) and the dynamic type (*Form*) of an instance.
  - The idea of static/dynamic types is used for run time polymorphism in .NET.

- Before we talk about generic types, let's review the type system of .NET 1.
- Pointer tricks can be done in unsafe contexts in C#.

Lack in the .NET 1 Type System in Action

See accompanying project <GenericTypes>
Presents the lack in the .NET 1 type system with object-based collections.

- Some definitions of strong typing wouldn't consider C# being a strongly typed language, because it allows type conversions (e.g. casts) to happen. In strongly typed languages type conversions are not present, but sophisticated type inference makes the type system easy to use (e.g. Haskell).
- Type systems in programming languages should not be categorized as being strongly or weakly typed. – This kind of terminology is not clearly defined.

Lack in the .NET 1 Type System (esp. Collections)

- Run time polymorphism (i.e. LSP) is the key of some .NET features:
  - 1. Cosmic hierarchy: *Object* is the base type of all .NET types.
  - 2. .NET 1 Collections: Hold items of static type *Object*, their dynamic types can vary.

- In fact Collections must be able to hold any type in order to be versatile.
  - .NET 1 "resorted" to run time polymorphism (i.e. LSP) to get this versatility.
  - So in a .NET 1 Collection, objects of any dynamic type can be stored.

- Object-based Collections works great, as long as...
  - we'll only store objects of dynamic types that dependent code awaits,
  - we'll only cast down to dynamic types that dependent code put in,
  - we'll never put mixed unrelated dynamic types into the same Collection,
  - or we finally use run time type checks all over,
  - well as we don't fall into a type problem during run time.

- *Object* based collections are sometimes called "weakly typed collections".

What are the Problems when using Object for Versatility?

- Valuetype objects will be boxed implicitly, which can be very time consuming.
- We have to know about the contained dynamic types.
  - Whenever *Object* is used in an interface, some convention must be around.
  - This convention describes the dynamic type, we're really dealing with.
  - The contributed programers have just to obey the convention... ;-)
- We have to cast down or unbox to the type we await.
  - The cast is needed to access the interface of a static type (e.g. *Form.Focus*()).
  - These casts indicate that the programer knows more than the compiler!
  - The compiler wears "cast contact lenses"; it can't type check, it trusts the programmer!
  - These casts are type checked at run time, so they are consuming run time!
- Type errors will happen at run time, not at compile time.

Fixing the Lacks in the Collection Type System

See accompanying project <GenericTypes>
Presents how to make code type safe with generic collections.

Generics to the Rescue

- What could we do to stay typesafe?
  - Use arrays as collections if possible, they're typesafe.
  - Resort to specialized typesafe collections, e.g. *StringCollection*.

- .NET 2 introduced generic types to get rid of the presented type problems.
  - Generic types are types that leave a part of type information open.
  - Generics are an outstanding feature in .NET 2.

- Generics leave type information open? How should this help me out?
  - The open infos are accessible by type parameters. (generic type and open type)
  - As programmers we can fill the type parameters by concrete type arguments.
  - By setting type arguments of a generic we create a type. (constructed type)
  - => Net effect: The type argument will be of static type, so the formerly open type information can be checked by the compiler.
  - Typesafety means that types are checked by the compiler.

- Another perception of generic types: They define a structure and some or all data types in that structure may change.
- The most important details of unbound, open and closed types are discussed in the attached examples.

Generic Collection Types

- .NET 1 Collection counterparts live in namespace *System.Collections.Generic*.
- The generic counterpart of type *ArrayList* is *List<T>*, *T* is the type parameter.
  - *List<T>*'s interface is set to open type *T* in parameter- and return-types.

```
public class List<T>
{ // "sort of" unbound generic type
      public void Add(T item) { /* pass */ }
}
```

  - The type parameter *T* acts as a placeholder for the actual type argument.
- When a type is constructed, *T* is replaced by a type argument.
  - *List<string>*'s interface is set to static type string in parameter- and return- types.

```
public class List<string>
{ // constructed type
      public void Add(string item) { /* pass */ }
}
```


- We should use these generic collections on a regular basis, if we start a new project. If we want to replace old style collections by their generic counterparts in existing code, be sure to have unit tests in place testing this refactored code. The problem is that some of the generic types have a slightly different behavior than their old style counterparts. (E.g. if we try to get a value for an unknown key in a *Hashtable* (with the indexer), null will be returned, but if we try the same (indexer) on a *Dictionary<T, U>* a *KeyNotFoundException* will be thrown.)
- As a matter of fact arrays have been updated to be based on generics in .NET 2 as well.

Constructed Types in .NET 2

- Similarities between C++' constructed types (templates) and those in .NET 2:
  - Generic instances can be directly created from the constructed type in .NET 2.

```
IList<string> names = new List<string>();
```

  - Also we can define a "typedef" as closed generic type alias:

```
using IntList = System.Collections.Generic.List<int>; // IntList as alias of List<int>.
```

- Differences between C++' constructed (template) types and those in .NET 2:
  - We can't create explicit or partial generic specializations in .NET 2.
  - We can't apply default type arguments or objects as arguments in .NET 2.
  - *List*<*string*> refers the same constructed type in every context in .NET 2.
    - E.g. in C++ *std::list*<*std::string*> can be a different type in different executables!
- In opposite to Java the type argument won't be erased at run time.
  - In .NET 2, the type *List*<*string*> exists at run time, not only at compile time.

- Here we discuss the difference of .NET's approach to generic types compared to C++ and Java.
- C++ templates are something like a sophisticated macro mechanism (before templates where in avail in C++, macros were used to mimic their behavior). C++ creates a new type per template instance at compile time, specialized types result from this procedure.
- Notice that we have to write the fully qualified namespace of the generic type to be aliased.
- This is valid for .NET 2 at run time: typeof(*List*<*int*>) != typeof(*List*<*string*>). In Java's run time it would result to: *List.class* == *List.class* due to the type erasure, the argument type was erased (type erasure) and the raw type *List* remained.
- Constructed types are created at run time (i.e. they are not present in the IL code), when they are needed. Only the "unbound generic type" (not an official term for type declarations) exists at compile time.
- In opposite to C++ (template instances) and Java (type erasure), .NET generics remain generic types at run time.
- In C# we can't use the type void as generic type argument (this is allowed in C++ and Java (as *java.lang.Void*)).

Using and creating new Generic Types

- Use present generic types: There's a rich set of existing generics to be used.
  - Generic Collections (*List<T>*, *Dictionary<T, U>* etc.)
  - Generic Delegates (*Action<T>*, *Predicate<T>* etc.)
  - We should prefer these generic types over non-generic ones.
- The type argument of a generic type, can also be a constructed generic type:

```
List<List<string>> lls; // This is a list of string-lists.
```

- Creating new generic types: Following .NET types can be defined generically.
  - interfaces, classes, structs, delegates and methods (including type inference).
- Implementation of generic and non-generic interfaces may conflict.
  - If we need to implement *IEnumerable<T>* we also have to implement *IEnumerable*.
  - Conflicts must be fixed with explicit interface implementation and redirection.

- We can not use ref/out parameter types for generic (esp. delegate) types.

Type Constraints in Action

See accompanying project <GenericConstraints>
Coding a generic type with constraints.

Implementing Generic Types – Part I

- The open type argument can't be arbitrary, if we call methods on it.

```
// Invalid! We can only call methods of the static type. If T is unconstrained, t's static type is Object!
int Foo<T>(T t ) {
      return t.Count;
}
```

- This is a sort of tradeoff, but we have to constrain the static type.

```
 // Fine (a type constraint)!
int Foo<T>(T t) where T : ICollection {
      return t.Count;
}
```

- Type parameters can be constrained in different ways, also overloads are allowed:

```
class C<T> where T : Form {} // T must be derived from type Form (type constraint).
```

```
class C<T, U> where T : struct where U : struct {} // T and U must be value types.
```

```
class D<T> where T : class, new() {} // T must be a reference type with a dctor.
```

- For .NET's Collection implementations the static type is generally transparent.
  - They hold any types of objects, but will only call *Object*'s methods on them.
  - (Exceptions: e.g. equivalence-based associative collections, which require *IComparable* to be implemented.)

17

- C++ does also provide a way to constraint templates: static_asserts. C#'s constraints are also similar to "concepts", a feature planned for future C++ standards.
- In Java, generics can be constrained with so called "bounds".
- In C# we have four kinds of constraints:
  - Reference type constraints and value type constraints that must be the first being declared.
  - Ctor type constraints must be the last ones.
  - In between we have the conversion type constraints (identity, reference and boxing conversion, no user conversion); a type argument must then fulfill all conversion type constraints. In conversion type constraints we can only have one class (as our type can only inherit one class), but multiple interfaces a type argument needs to implement/inherit.
- We can not use type constraints for value types, sealed types or for following "special" types: *System.Object*, *System.Enum*, *System.ValueType* or *System.Delegate*.

Implementing Generic Types – Part II

- To get the default value of a type parameter use the default(*T*) expression.
  - The default value is null for reference types.
  - The default value is 0 or false for value types (or their fields).

- We can't call operators on "open" types' instances.
  - Only the interface of type arguments can be checked.
  - In .NET static methods/properties/fields/delegates don't belong to this interface.
  - Consequence: we can't use (mathematical) operators on "open" types!

18

- The operators != and == can be used if the type is constrained to be a reference type. But then only the references will be compared. If the types are further constrained to derive from a special reference type the overloads of == and != of exactly that derived type will be called. It is possible to use various operators with the advent of .NET 4/C#4 and dynamic typing.
- Also, we can not define generic ctors (C++ allows the definition of template ctors for type-conversion scenarios).

Operators can't be called on open Types' Instances

See accompanying project <GenericConstraints>
The prove: operators can't be applied.

Thank you!