New C#4 Features – Part II

Nico Ludwig (@ersatzteilchen)

TOC

- New C#4 Features – Part II
  - New generic Types of the BCL in Detail: Lazy<T> and Tuple<T, ...>
  - Generic Variance

- Sources:
  - Jon Skeet, CSharp in Depth
  - Chamond Liu, Smalltalk, Objects, Design

New generic BCL Types

See accompanying project <NewGenericBCLTypes>
Shows the type Lazy<T>, Tuples and structural comparison with Tuples and arrays.

Lazy<T>

- *Lazy<T>* provides simple support for deferred initialization.
  - Idiomatic usage as (encapsulated) field of a type.
- Lazy initialization w/ dctors as well as w/ initializer functions is supported.
- Threadsafe initialization is supported.

Tuples – Part I

- *Tuple*s are known from functional programming as ad hoc data structure.
  - A *Tuple* is: Basically a list of values (called components) of possibly different static types.
  - They have been introduced to use F# APIs that make use of *Tuple*s.
  - .NET *Tuple*s are ordinary generic types, but can have any number of components.
    - There are generic overloads for up to seven components.
    - This limit can be overcome by cascading *Tuple*s.
- *Tuple*s vs. anonymous types:
  - Only Tuples (as static type) can be returned from methods and passed to methods.
  - Only anonymous types have semantic quality, *Tuple*s are rather data containers.
  - Both override *Equals()* and *GetHashCode()* in order to express value semantics.
  - Both are immutable in C#.

- Slightly like Lisp's concept of cons-cells.
- As the *Tuple*-types are not sealed you could derive your own implementation containing extra behavior. – But that's questionable as you can't hide possibly unwanted information (like the *ItemX*-properties accessing the components).
- *Tuple*s can't be XML serialized (this esp. includes that they don't have a dctor).

Tuples – Part II

- Pro:
  - Allow creating more composable APIs.
    - As alternative to passing back results with out/ref.
    - Also very handy as return type of anonymous functions (e.g. in TPL code).
  - Support structural comparison.
    - New in .NET 4: arrays also implement *IStructuralEquatable* and *IStructuralComparable* explicitly.
  - Allow exploiting F# APIs that deal with *Tuple*s.
- Contra:
  - As they have fixed property names, they have very limited semantics.
    - Limit their usage to return types and parameter types.
    - Understand and use them as data containers.
  - Can't be used in web service interfaces, as *Tuple*s can not be accordingly serialized.

6

- Using of *Tuple*s can be better than anonymous types, if you need to return the result of an anonymous function from an enclosing <u>named</u> function!
- *Tuple*s are fine to create fluent APIs, as they allow to get rid of ref and out parameters.

Consistency of Type Hierarchies: Covariance

- What does "consistency" mean?
  - The idea is to design type-safe type hierarchies with highly substitutable types.
  - The key to get this consistency is subtype or subclass based conformance.
  - (We'll discuss the Liskov Substitution Principle (LSP) on a high level here.)

- Example: Let's assume *Bird*s that lay *Egg*s "consistently" with static typing:
  - If the type *Bird* lays an *Egg*, wouldn't a *Bird*'s subtype lay a subtype of *Egg*?
  - E.g. the type *Duck* lays a *DuckEgg*; then the consistency is given like so:

  - Why is this consistent? Because in such an expression (no C#!):

    `Egg anEgg = aBird.Lay()`

    the reference *aBird* could be legally substituted by a *Bird* or by a *Duck* instance.
  - We say the return type is <u>covariant</u> to the type, in which *Lay()* is defined.
  - A subtype's override may return a more specialized type. => "They deliver more."

- The term sub typing is used here a little sloppy. Virtually the terms sub typing and sub classing have different meanings when discussing substitutability, but the difference is not relevant here.
- The LSP was introduced by Prof. (MIT) Barbara Liskov in 1987.
- Covariance is esp. handy on abstract factories.
- Exception specifications in C++ and Java are also covariant (as they are similar to return types).
- Covariance/contravariance are terms taken from the mathematical "category theory" that deals with the substitution principle.

Consistency of Type Hierarchies: Contravariance

- Let's assume *Piano*s that *Pianist*s can play "consistently" with static typing:
  - If a *Pianist* plays *Piano*, would she be able to play a *GrandPiano*?
  - Wouldn't rather a *Virtuoso* play a *GrandPiano*? (Be warned; there is a twist!)

  - This is inconsistent! Because in such an expression (no C#):

```
aPiano.Play(aPianist)
```

  - *aPiano* couldn't be legally substituted by a *Piano* or by a *GrandPiano* instance! A *GrandPiano* can only be played by a *Virtuoso*, *Pianist*s are too general!
  - *GrandPiano*s must be playable by *more general types*, then the play is consistent:

  - We say the parameter type is contravariant to the type, in which *Play()* is defined.
  - A subtype's override may accept a more generalized type. => "They require less."

- The described conformance (covariant return types/contravariant parameter types) is the theoretical ideal (supported by the languages Emerald and POOL-1). Some oop languages (e.g. Eiffel) decided to apply another type of consistency, esp. also covariant parameter types, because it better describes the reality than the theoretical ideal.
- In statically typed languages the desired consistency must often be achieved by application of design patterns like "double dispatching" and "visitor". Other languages provide so-called "multiple dispatch" (e.g. CLOS) or get the desired effect by using dynamic typing.

Problems with covariant Arrays in C#

See accompanying project <IntrinsicCSharpCovarianceWArrays>
Shows the problem with covariance of arrays in C#.

Put the Consistency Theory into Practice with C#4

- Variance expresses the consistency of type-safe type substitution.
- C# does already provide covariance on arrays.
  - Yes, it's not type-safe, but can we do anything more to reach better consistency?
- Yes, via generic variance available in C#4!
  - (But C#4's consistency is still based on invariant return and parameter types!)
- So, you can enable variances on generic delegate and interface types.
  - This feature was present since .NET 2, it was enabled for C# in C#4.
  - The idea is to define the variance of return and parameter types individually.
  - Different variances can be mixed in a specific delegate or interface.
  - With variance, the compiler will treat in- and out-types differently, so that variant generic types act as if they were related.

- Covariance on arrays works only w/ reference conversions: inheritance and interface (of implicitly or explicitly implemented interfaces) conversions. (No boxing, value type or user conversions are allowed!)
  - Try to limit the usage of array parameters to param-arrays, as they are much safer.
- In C#4 no covariant return types are available (in opposite to Java and C++).
- In C# methods with out and ref parameters are also not variant on that parameters.
- Variance can only be declared on generic interfaces and delegates, not on classes or structs. => It does not work without explicitly refactoring type hierarchies.

Generic Interface Variance in Action

See accompanying project <CSharpCovariance>
This example shows generic interface variance in C#.

- No example dealing with generic variance on delegates will be presented here. But you can find some examples in the source code accompanying this presentation.

Summary: generic Variance

- What is generic interface/delegate variance?
  - It allows "reasonable" implicit conversions between unrelated types to occur.
    - So the main features variance enables are co- and contravariant conversions.
  - It reduces the amount of compile time errors.
- But only reference conversions of return and parameter types can be used!
  - (As with array covariance: no boxing, value type or user conversions are allowed!)
- Possibly you'll never code own variant types, but you'll exploit existing ones.
  - Some .NET types have been made variant: *IEnumerable<out T>/IEnumerator<out T>*, *IComparable<in T>*, *Action<in T, ...>*, *Func<out T, in R, ..>*, *Converter<in T, out R>*.
- Pitfalls with variant types:
  - New implicit conversions are in avail: type checks and overloads behave differently.
  - Generic variance on delegates does not work with multicasting.

- Example application: return types of factory interfaces.
- Reference conversions: inheritance and interface (of implicitly or explicitly implemented interfaces) conversions.
- .NET collection types must stay invariant as they use the same parametrized type for return and parameter values.
- The problem with multicasting might be fixed with a future .NET version.
- Before generic variance was enabled with C#4 there was a way to simulate covariance (see: http://blogs.msdn.com/b/kcwalina/archive/2008/04/02/simulatedcovariance.aspx).

Variance "Mechanics"

- Expressing generic variance on interfaces:

```
interface IFactory<out T>
{ // Covariant on T.
      T Produce();
}
```

```
interface IAcceptor<in T>
{ // Contravariant on T.
      void Accept(T t);
}
```

- Expressing generic variance on delegates:

```
// Covariant on T.
delegate T Func<out T>();
```

```
// Contravariant on T.
delegate void Action<in T>(T t);
```

- Restrictions on expressing generic variance:
  - Only interface and delegate variance is supported.
  - Due to guaranteed static type-safety, return types can only be covariant and parameter types can only be contravariant!
    - You have to declare the variance on the type parameters explicitly!
    - You can not express your own consistency, esp. no consistency can be expressed on exceptions.
  - If T is a return type and a parameter type that type must be invariant.
  - out method parameters are always invariant, as they are CLR ref parameters!

- The C# compiler could, at least in most cases, infer the right variances; but you have to declare variances explicitly, because it might lead to breaking code.
- Since exceptions are not part of signatures in C#, no consistency can be declared on them.
- Because out method parameters can also be used to pass data to a method they can not be safely covariant as return types. The CLR does not make a difference between out and ref parameters.
- In Java, variances are expressed on the caller side of the (generic) type, i.e. nothing must be declared on the type itself to act variant. – C# uses statically checked definition-site variance (so does Scala).
- In dynamic languages the consistency is often present intrinsically, because substitutability is not expressed by static types, but by common interfaces (read: couple of methods). – Subtyping instead of subclassing is used here: more on this in following lectures.
- The IL syntax uses the symbols +/-T to denote the type of variance (the Scala syntax is similar).

Thank you!