C# Generics

Object Oriented Programming (236703)

Winter 2014-5

C# Generics in a nutshell

- Outline
  - Generics – what is it good for?
  - C# generics semantics
  - Generics and reflection
  - Limitations
  - Variance

Why Do We Need Generics?

Everything inherits from Object, so this list can hold any type:

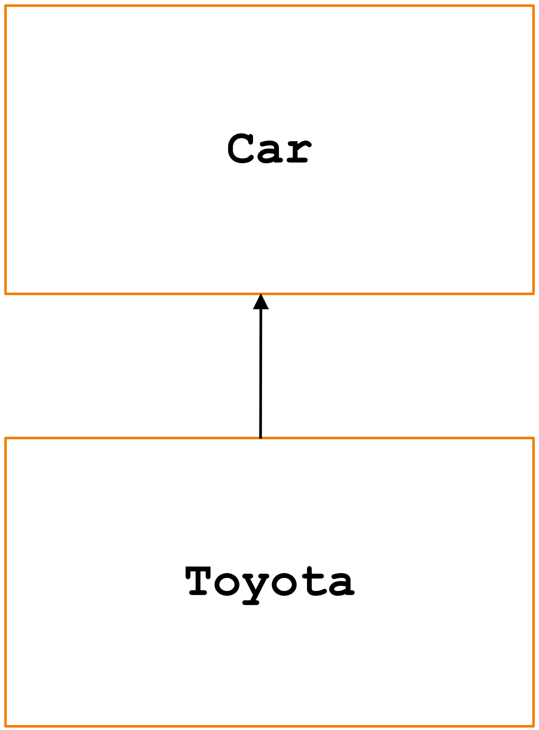
```
interface IList {
    public void Add(object o);
    public object Get(int index);
    ...
}
```

And this will always work:

```
void PrintList(IList list) {
    foreach (object o in list)
        Console.WriteLine(o);
}
```

No code duplication, which is what we wanted to avoid!

Introducing Our Use-case



Generics Promotes Type Safety

```
void PrintCars(IList cars) {
    foreach (object o in cars)
        Console.WriteLine(o);
}
```

```
void PrintCars(IList cars) {
    foreach (Car c in cars)   // illegal
        Console.WriteLine(c.RemainingFuel);
}
```

Type-safety In Generics

- With generics at hand, we can now do:

```
public void PrintVehicles(IList<Car> cars){
    foreach (Car c in cars)
        Console.WriteLine(c.RemainingFuel);
}
```

- Elements in cars are now checked for their types statically

Semantics Of Generics In C#

- Somewhere in the middle between Java and C++
- Each parametrized generic class forms a new type *(C++ semantics)*
- Constraints are not implicitly imposed by the compiler, but explicitly by the programmer *(Java semantics)*
- Generics are a language *and* CLR feature
  - Whose feature is Generics in Java?

C# Semantics – Implications

- Downside – It may cause code segment to dramatically increase in size
  - Solution #1 – types are instantiated on demand (at run-time – CLR feature)
  - Solution #2 – All reference types share the same IL code
- Upside – no type erasure in binaries, which enables:
  - Better optimization
    - E.g., no need for boxing and unboxing of value types
  - Better reflection support

Reflection And Generics

Generic parameters can be retrieved by reflection:

```
void ExploreGeneric(object o) {
    if (o.GetType().IsGenericType) {
        Type genericParameter =
            o.GetType().GetGenericArguments()[0];
        Console.WriteLine("o is parameterized " +
           "with class " + genericParameter.Name);
    }
}
```

Reflection And Generics (cont.)

- Generic types may be also created on the fly:

```
Type CreateGenericList(Type parameter){
  Type listType = typeof(System.Collections.Generic.List<>);
  return listType.MakeGenericType(parameter);
  // OR
  string typename = string.Format(
    "System.Collections.Generic.List`1[{0}]", parameter.FullName);
  return Type.GetType(typename);
}
```

- Usage:

```
CreateGenericList(typeof(int)); // creates: List<int>
```

- That enhanced reflection support could not have been achieved if there were type erasure in the binaries!

Generic Parameter Constraints

- A modification on Java semantics
- Not imposed implicitly by the compiler, but explicitly by the user (as it is in Java)
- Java: can upper-bound classes and interfaces
    - Wildcards can be bound both ways (but not together)
- C#: can upper-bound to classes and interfaces, and can also constrain on reference / value types and on default constructors.

Parameter Constraints - Example

- Bounding a parameter to an interface or a class:

```
public void Print<C, T>(C collection)
    where C : IEnumerable<T> {

    foreach (object o in collection)
        Console.WriteLine(o);

}
```

- Also supported in Java:
  - public <T, C extends Iterable<T>> void Print(C collection)

Parameter Constraints – Example (2)

- One can assign more than one constraint:

```
public int BiggerThanTwo<T, U>(T collection)
            where T: IEnumerable<U>
            where U: System.IComparable<int> {
    int ret = 0;
    foreach (U item in collection) {
        if (item.CompareTo(2) > 0)
            ret++;
    }
    return ret;
}
```

Parameter Constraints – Example (3)

Other types of constraints:

```
public bool IsSubType<T,U>(T t, U u) where T : U {
  return true;
}
public void Foo<T>(T t) where T : struct {
  // we can now assume T is a value type
}
public void Bar<T>(T t) where T : class {
  // we can now assume T is a reference type
}
public void Baz<T>(T t) where T : new() {
  // we can now assume T in a non-abstract
  // type with a public parameterless
  // constructor
}
```

not supported in Java

C# Generics Limitations

- Generics are no-variant by default
  - `List<object> lo = new List<string>();` → Error
- Although MSIL supported generic covariance, C# doesn't!
- Co-variance and Contra-variance for Generic interfaces are supported from C# 4.0 (VS2010)
  - By the use of in/out (example in a few slides)
  - Makes coding much more simple
  - C# 4.0 also introduced dynamic variables

Generic Delegates

- Delegates ≈ function pointer type (details on the tutorial)
  - `public delegate object ConversionDelegate(string d);`
    - Can hold various conversion methods that take a string and return an object
- Delegates also have a generic version

```
delegate T ConversionDelegte<T, U>(U u);
```

- Generic delegates are no-variant by default, unlike their non-generic counterparts.
  - ```
    ConversionDelegte<object, string> cd =
            new ConversionDelegte<string, object>(…);
    ```
    ➔ *Error!*
    ```

Generics Co-variance And Contra-variance

- Variant type parameters are restricted to generic interface and generic delegate types
- A generic interface or generic delegate type can have both covariant and contra-variant type parameters
- Variance applies only to reference types; if you specify a value type for a variant type parameter, that type parameter is invariant for the resulting constructed type

```
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```

The *out* Generic Modifier

- Specifies that the type parameter is covariant
- Can be used in generic interfaces and delegates
- Enables its methods to return more derived types than those specified by the type parameter

```
interface IReadOnlyCell<out T> // 'T' is  covariant
{
  T get(); // ok
  // void set(T t);  // Invalid variance: The type parameter
                     // 'T' must be contra-variantly valid.
}
```

The *out* Generic Modifier (2)

- Given:

```
class Cell<T> : IReadOnlyCell<T> {
  private T value;
  public T get() { return value; }
  public void set(T t1) { value = t1; }
}
```

```
class A {}

class B :
    A {}
```

- Valid usage example:

```
IReadOnlyCell<A> ba = (IReadOnlyCell<B>)new Cell<B>(); // ok
A a = ca.get(); // ok.
```

- Illegal usage example:

```
ba.set(new A()); // error - if ok would cause a run-time error
IReadOnlyCell<B> cc = (IReadOnlyCell<A>)new Cell<A>(); // error
```

The *in* Generic Modifier

- Specifies that the type parameter is contra-variant
- Can be used in generic interfaces and delegates
- Used only as a type of method arguments
- *ref* and *out* parameters cannot be variant
- Allows its methods to accept arguments of less derived types than those specified by the interface type parameter

```
interface IWriteOnlyCell<in T> // 'T' is contra-variant
{
  //T get(); // Invalid variance: The type parameter 'T'
             // must be covariantly valid.
  void set(T t); //ok
}
```

The *in* Generic Modifier (2)

- Given :

```
class Cell<T> : IWriteOnlyCell<T> {
  private T value;
  public T get() { return value; }
  public void set(T t1) { value = t1; }
}
```

```
class A {}

class B :
    A {}
```

- Valid usage example:

```
IWriteOnlyCell<B> ba = (IWriteOnlyCell<A>)new Cell<A>(); // ok
cc.set(new B()); // ok.
```

- Illegal usage example:

```
B b = ba.get(); // error - if ok would cause a run-time error
IWriteOnlyCell<A> ba = (IWriteOnlyCell<B>)new Cell<B>(); // error
```

Reminder: Pre-C# 4.0 Limitation

- IList is no-variant: a list of cars is not a list of Toyotas

```
public static void CarIter(IList<Car> carList) {
    foreach (Car c in carList) {
        Console.WriteLine(c.RemainingFuel);
    }
}
```

```
CarIter(new List<Car>()); // OK
CarIter(new List<Toyota>()); // does not compile
```

Using Covariance

```
public static void CarIter(IEnumerable<Car> carList)
{
  foreach (Car c in carList)
    Console.WriteLine(c.RemainingFuel);

}

public static void Main(string[] args)
{
  CarIter(new List<Car>());
  CarIter(new List<Toyota>());
  // this works because IEnumerable is declared as <out T>.
  // IList is not.
}
```

Dynamically-typed Variables

- Defined with the “dynamic” keyword
- Bypass compile-time type checking
- Operations are resolved and type checked at run time
- Variables of type dynamic are compiled into variables of type object (type dynamic exists only at compile time)

```
dynamic dyn = 1;
dyn = dyn + 1;
System.Console.WriteLine(dyn.GetType()); //System.Int32
dyn = “string”;
System.Console.WriteLine(dyn.GetType()); //System.String
```

Parametric Polymorphism

Feature	C++	C#	Java
Instantiation	Compile time	Run time	~Compile time
Non-type params (<int i>)	Yes	No	No
Specialization	Yes	No	No
Default type parameters	Yes	No	No
Constraints	Implicit	Explicit	Explicit
Mixin	Yes	No	No
Variance (of variables)	No	Yes (using in and out)	Yes (using wildcards)
Reflection	Name only (using RTTI)	Full	No
Executable size overhead	For each instantiation	For each generic type	For each generic type
And there's more…			