C# Generics in a nutshell

Outline

- Generics – what is it good for?
- C# generics semantics
- Generics and reflection
- Limitations
- Creative ways of overcoming the above
Why do we need generics?

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

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

And this will always work:

```csharp
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

Car

Toyota
Generics promotes type safety

```csharp
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:

```csharp
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*).
C# semantics – Side Effects

- **Downside**
  - It may cause code segment to dramatically increase in size.
  - (Partial) Solution - types are instantiated on demand (at run-time – CLR feature).
  - (Partial) Solution – All reference types share the same IL code.

- **Upside** - no type erasure in binaries, which enables:
  - Better optimization.
  - Better reflection support.
  - No need for boxing and unboxing of value types.
Reflection and Generics

- Generic parameters can be retrieved by reflection:

```csharp
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:

  ```csharp
  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 was 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- or lower- bind to classes and interfaces.
- C#: can upper-bind to classes and interfaces, and can also constrain on reference / value types and on default constructors.
Parameter Constraints - Example

- Binding a parameter to an interface or a class

```csharp
public void Print<T>( T collection )
where T: IEnumerable{
    foreach( Object o in collection )
        Console.WriteLine(o);
}
```

- Also supported in Java
● One can assign more than one constraint

```csharp
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;
}
```
Other type of constraints exists

```csharp
public bool IsSubType<T, U>(T t, U u) where T : U {
    return true;
}

public void Bar<T>(T t) where T: struct {
    // we can now assume T is a value type
}

public void Foo<T>(T t) where T: class {
    // We can now assume T is a reference type
}

public void Kuku<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.
- 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 later slides).
  - Makes coding much more simple.
- Who can guess what else? (hint – smalltalk).
Generic Delegates

- Delegates also have a generic version

```csharp
delegate T ConversionDelegate<T,U>(U u);
```

- Generic delegates were no-variant, unlike their non-generic counterparts.
The problem - generic classes are invariant to their template arguments.

Solution could be made by the usage of delegates.

- (Q) Why delegates? How do they help us?
- (A) As you remember – delegates are co-variant on their return value.
Reminder: what we couldn’t do (before C# 4.0)

- This will only accept a list of cars, not a list of Toyotas (because IList is invariant):

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

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

- Object-oriented Programming
Solution to Generics no-variant property - 1st step

- Define a delegate

```csharp
delegate Car CarDelegate();
```

- The client method now iterates the items by invoking the delegate:

```csharp
public static void CovariantCarIter(CarDelegate carDel) {
    Car current;
    while( (current = carDel()) != null ) {
        Console.WriteLine(current);
    }
}
```
Solution to Generics no-variant property – 2\textsuperscript{nd} step

- Define a wrapper class around IEnumerableView

```csharp
public class IteratorWrapper<T> where T: class {
    readonly IEnumerator<T> enumerator;

    public IteratorWrapper(IEnumerable<T> enumerable) {
        this.enumerator = enumerable.GetEnumerator();
    }

    public T GetNext(){
        if (enumerator.MoveNext())
            return enumerator.Current;
        else
            return null;
    }
}
```
Solution to Generics no-variant property – usage

- We have achieved co-variance for IEnumerable<T>:

```csharp
List<Car> cars = new List<Car>();
List<Toyota> toyotas = new List<Toyota>();

IteratorWrapper<Car> carIter = new IteratorWrapper<Car>(cars);
IteratorWrapper<Toyota> toyotaIter =
    new IteratorWrapper<Toyota>(toyotas);

CarDelegate carDel = carIter.GetNext;
CarDelegate toyotaDel = toyotaIter.GetNext;

GenericCovariance.CovariantCarIter(carDel);
GenericCovariance.CovariantCarIter(toyotaDel);
```
C# 4.0 Covariance and Contravariance in Generics

- 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.

```csharp
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```
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 contravariantly valid.
}
out (Generic Modifier) cont.

- **Given:**

  ```java
  class Cell<T> : IReadonlyCell<T>{
      private T value;
      public T get() { return value; }
      public void set(T t1) { value = t1; }
  }
  
  class A {}  // A is a class
  class C : A {}  // C inherits from A
  ```

- **Valid usage example:**

  ```java
  IReadonlyCell<A> ca = (IReadonlyCell<C>)new Cell<C>(); // ok
  A a = ca.get(); // ok.
  ```

- **Illegal usage example:**

  ```java
  ca.set(new A()); // If was possible, would cause a run-time error.
  IReadonlyCell<C> cc = (IReadonlyCell<A>)new Cell<A>(); // compilation error
  ```
in (Generic Modifier)

- Specifies that the type parameter is contravariant.
- 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.

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

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

class A {}
class C : A {}
```

Valid usage example:

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

Illegal usage example:

```java`
A a = cc.get(); // If was possible, would cause a run-time error.
IWriteOnlyCell<A> ca = (IWriteOnlyCell<C>)new Cell<C>(); // compilation error
```
Solution using covariance

```csharp
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.
}
```

236703 - Object-oriented Programming
dynamically-typed variables

- defined with the “dynamic” keyword.
- Bypass compile-time type checking.
- Operations are resolved and type checked at runtime.
- variables of type dynamic are compiled into variables of type object (type dynamic exists only at compile time, not at run time).

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

236703 - Object-oriented Programming 27
## C++ templates vs. C# generics

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++</th>
<th>C#</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instantiation</td>
<td>Compile time</td>
<td>Run time</td>
<td>~Compile time</td>
</tr>
<tr>
<td>Non-type params (int i)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Specialization</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Default type parameters</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Constraints</td>
<td>Implicit</td>
<td>Explicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Mixin</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Variance</td>
<td>No</td>
<td>Yes</td>
<td>Covariance (?)</td>
</tr>
<tr>
<td>Reflection</td>
<td>Name only (using RTTI)</td>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>Executable size overhead</td>
<td>For each instantiation</td>
<td>For each generic type</td>
<td>For each generic type</td>
</tr>
</tbody>
</table>

And there's more…
כמה מילים סיכום

כמה מילים על מה שػלדנו בקורס.

כמה מילים על המבחן

הומר: כל מה שנלמד בכרזאות, תרגולים וeteranganי הבית.
– מבנה: 5 שאלות 20 נק’ ב”א.
– תומר פיתוח.
–
תודה!
חופשה ונעים והצלחה בבחינות!

“Thats all Folks!”

תודה!
חופשה ונעימה ובצלחון בבחינות!