C# Generics

Object Oriented Programming (236703)
Winter 2015-6
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*)
- Generics are a language *and* CLR feature
  - Whose feature is Generics in Java?
C# Semantics – Implications

- **Downside** – It might have caused 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:

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

What generic information is available at run time in Java?
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:

```csharp
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 of Java semantics
- Not imposed implicitly by the compiler, but explicitly by the user (as it is in Java)
- Upper-bound classes and interfaces: both Java and C#
- Covariant and contra-variant assignment: wildcard in Java, in/out type arguments modifiers in C#
- C# has a few more constraints: reference/value types, default constructors
Parameter Constraints - Example

Bounding a parameter to an interface or a class:

```csharp
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)
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 types of constraints:

```csharp
class ParameterConstraints {
    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 Variance

- 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
Generic Delegates

- Reminder: Delegates ≈ function pointer type
  - 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
  
  ```csharp
  delegate T ConversionDelegate<T, U>(U u);
  ```

- Generic delegates are no-variant by default, unlike their non-generic counterparts:
  - `ConversionDelegate<object, string> cd = new ConversionDelegate<string, object>(...);
  - Error!
Generics Co-variance And Contra-variance

- Only generic interfaces and delegates can be variant – classes cannot
- A generic interface or delegate can have both covariant and contra-variant type parameters
- Variance applies only to reference types
  - Value type arguments make the type parameter invariant for the resulting type

```csharp
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

```csharp
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:

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

```csharp
class A {}
class B : A {}
```

### Valid usage example:

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

### Illegal usage example:

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

```csharp
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 \textit{in} Generic Modifier (2)

\begin{itemize}
  \item Given:
    \begin{verbatim}
    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 {}
    \end{verbatim}
  \item Valid usage example:
    \begin{verbatim}
    IWriteOnlyCell<B> ba = (IWriteOnlyCell<A>)new Cell<A>(); // ok
    ba.set(new B()); // ok.
    \end{verbatim}
  \item Illegal usage example:
    \begin{verbatim}
    B b = ba.get(); // error - if ok would cause a run-time error
    IWriteOnlyCell<A> ba = new Cell<B>(); // error
    \end{verbatim}
\end{itemize}
Reminder: Pre-C# 4.0 Limitation

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

```csharp
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

```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.
}
```
## Parametric Polymorphism

<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 (&lt;int N&gt;)</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 (of variables)</td>
<td>No</td>
<td>Yes (using in and out)</td>
<td>Yes (using wildcards)</td>
</tr>
<tr>
<td>Reflection</td>
<td>Name only (using RTTI)</td>
<td>Full</td>
<td>Very little</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…
The Tedious Part Is Over!

And now, let’s discuss the fun part:

The Exam!

Closed material, 5 questions –

- 3 originals
- 1 dealing with a homework task
- 1 from a previous exam (last 7 years, found on the website)
What's Next?

- This course mostly discussed OO language features
  - The need, the solution, the implementation
- Also, a bit of high-level abstract discussion
- You know the tools now, but aren’t proficient in using them
- Take the Software Design course, and complete the chain
  - It will make you a better programmer, and that’s what most of us are going to be

Good Luck!