



C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 - May 2007

Lecture 6: Generics

Lisa (Ling) Liu

Overview



- Problems with untyped collections
- Generics
- Define generics
- Covariance

The problems with untyped collections



```
ArrayList list = new ArrayList();  
double a;  
list.Add(2);  
a = (double)list[0];
```



```
.locals init ([0] class [mscorlib]System.Collections.ArrayList list,  
             [1] float64 a)
```

```
...  
newobj instance void [mscorlib]System.Collections.ArrayList::.ctor()  
...  
box [mscorlib]System.Int32  
callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)  
...  
callvirt instance object  
[mscorlib]System.Collections.ArrayList::get_Item(int32)  
unbox.any [mscorlib]System.Double
```

Problems:

- Performance
- Type safety



- Introduce the concept of type parameter
- Defer the specification of types until declaration or instantiation
- Avoid the cost of runtime boxing operation and type casts
- Avoid the risk of runtime casts
- Allow you to define a type-safe data structure or a utility helper class without committing to the actual data types used.

Generic namespace



Strongly typed collections

- `Compare <T>`
- `Dictionary <K, T>`
- `SortedDictionary <K, T>`
- `List <T>`
- `Queue <T>`
- `Stack <T>`

```
List<int> l = new List<int>();  
double a;  
l.Add(2);  
a = list[0];
```

- No boxing and unboxing
- The compiler checks the type compatibility, if necessary, does implicit type conversion, otherwise reports errors



Define generic methods

access_modifier return_type method_name <T> (arguments)

```
public static void Swap<T> (ref T a, ref T b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
static void Main (string[] args)
{
    int x, y;
    x = 0;
    y = 1;
    GenericClass.Swap(ref x, ref y);
}
```

Define generic classes



access_modifier class_name <T> {: inheritance list}

```
public class Point<T>
{
    private T xPos;
    private T yPos;
    ...
    public Point (T xVal, T yVal)
    {
        ...
    }
    ...
}
```

default keyword in generic code



- In generic code, **default** keyword is used to set a type parameter to its default value
 - Numeric values have a default value of 0
 - Reference types have a default value of null

Creating a custom generic collection



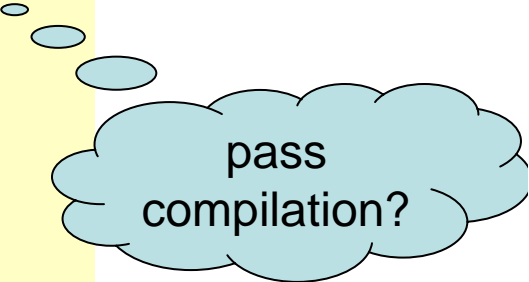
```
access_modifier class_name <T> : IEnumerable <T>
```

- **IEnumerable<>** extends the nongeneric **IEnumerable** interface
- The class that implements interface **IEnumerable<T>** should implement two versions of the **GetEnumerator()** method

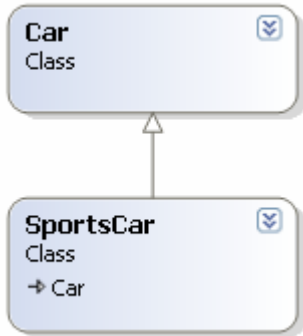
```
public interface IPerson
{
    string Name { get;}
}
```

```
public class Person : IPerson
{
    private string _name;
    public Person()
    {
    }
    public string Name
    {
        get { return this._name; }
        set { this._name = value; }
    }
}
```

```
public static class PersonFactory
{
    public static List<IPerson> CreatePeople()
    {
        List<Person> people = new List<Person>();
        return people;
    }
}
```



pass
compilation?



```
Car[] arrCar = new Car[10];
SportsCar[] arrSportsCar = new SportsCar[10];
arrCar = arrSportsCar;
```

**compiler
error**

```
List<Car> lCar = new List<Car>();
List<SportsCar> lSportsCar = new List<SportsCar>();
lCar = lSportsCar;
```

```
CarList<Car> colCar = new CarList<Car>();
CarList<SportsCar> colSportsCar = new CarList<SportsCar>();
colCar = colSportsCar;
```

**compiler
error**

Covariance & Contravariance



substitution principle:

any expression of type t can be substituted by an expression of type s if $s \leq t$

(Here, \leq denotes the subtype relationship)

- A **covariant** type operator in a type system preserves the \leq ordering of types
- A **contravariant** type operator in a type system reverses the \leq ordering of types

Subtyping relation for functional types



Contravariant rule (Cardelli [1998]):

if $T_1 \leq S_1$ and $S_2 \leq T_2$ then $S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2$

- It sounds kind of counterintuitive
- It reflects the famous advice about the implementation protocols: "be liberal in what you accept, and conservative in what you send."
- It achieves the static type safety

C# covariance



- Array of reference-types are covariant
`object[] b = new string[1];`
- Array of value-types are invariant
`int[]` is not a subtype of `double[]`
- Delegates are covariant
- Method overriding is not covariant

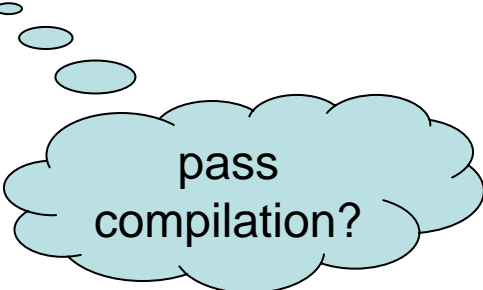
```
public delegate CarDelegate();
```

```
public static Car GetBasicCar()  
{ return new Car(); }
```

```
public static SportsCar GetSportsCar()  
{return new SportsCar();}
```

```
static void Main()  
{  
    CarDelegate targetA = new CarDelegate(GetBasicCar);  
    Car c = targetA();  
  
    CarDelegate targetB = new CarDelegate(GetSportCar);  
    SportsCar sc = (SportsCar) targetB();  
}
```

```
class CarList<T>: IEnumerable<T>
{
    private List<T> arCars = new List<T>();
    ...
    public void PrintPetName(int pos)
    {
        Console.WriteLine(arCars[pos].Name);
    }
    ...
}
```



pass
compilation?

Type parameter constraints



Unbound generic

when a type parameter is not constrained in any way, the generic type is said to be unbound

Bound generic

use the keyword `where` to set the constraints of a generic type.

the constraint list is placed after the generic type's base class and interface list.

Possible constraints for generic parameters



where T: struct

where T: class

where T: new ()

where T: NameOfBaseClass

where T: NameOfInterface

Examples



```
public class MyGenericClass<T> where T: new()  
{ ... }
```

```
public class MyGenericClass<T> where T: class, IDrawable, new()  
{ ... }
```

```
public class MyGenericClass<T> : MyBase, ISomeInterface  
    where T: struct  
  
{ ... }
```

```
public class MyGenericClass<K, T> where K: new()  
    where T: IComparable<T>
```

Inherit generic class



- Generic class can be the base class to other classes
- If a nongeneric class inherits a generic class, the derived class must specify a type parameter
- If the derived type is generic as well, the child class can (optionally) reuse the type placeholder
- Any constraints placed on the base class must be honored by the derived type

```
public class MyList<T>
{
    ...
}
```

```
public class MyStringList: MyList<string>
{
    ...
}
```

```
public class MyList<T> where T: new()
{
    ...
}
```

```
public class MySubList<T> : MyList<T> where T: new()
{
    ...
}
```

Creating generic interface



```
public interface IBinaryOperations<T>
{
    T Add(T arg1, T arg2);
    T Subtract(T arg1, T arg2);
}
```

Creating generic delegates



```
public delegate void MyGenericDelegate<T> (T arg)
```

```
...
```

```
MyGenericDelegate<string> strTarget = new  
MyGenericDelegate<string> (StringTarget);
```

Questions

