Generics Behind the Scenes

Bart J.F. De Smet
bartde@outlook.com

pluralsight
hardcore dev and IT training

Introducing Generic Types

- **Parameterization by type for**
  - Types: Interfaces, Classes, Structs, Delegates
  - Methods

- **Runtime feature added in CLR 2.0 (codename "Gyro")**
  - Specialized type and method layouts created at runtime
    - Unlike erase-based approaches, cf. Java
    - Full-fidelity type information in reflection
  - Support for constaints
  - Co- and contravariance

- **Language feature added in C# 2.0**
  - Simple angle-bracked syntax, e.g. List<T>
    - List(Of T) in Visual Basic
  - Support for variance annotations added in C# 4.0

Why Generics Matter

- **Static typing benefits**
  - Compile-time checking

```
class Stack
{
  object[] _items;

  public void Push(object o)
  { … }

  public object Pop()
  { … }
}
```

```
class Stack<T>
{
  T[] _items;

  public void Push(T o)
  { … }

  public T Pop()
  { … }
}
```

- **Performance improvements**
  - Reduces boxing of values
  - Less runtime-checks needed
  - No compile-time expansion (e.g. templates)

Avoiding Boxing Cost

- **Non-generic collections containing values**
  - Storage using System.Object
  - Boxing on insertion
  - Unboxing on retrieval
    - Requiring casting
    - Cost in foreach loops

```
ArrayList xs = new ArrayList();
xs.Add(/* boxing */ 2);
int two = (int)xs[0];
```

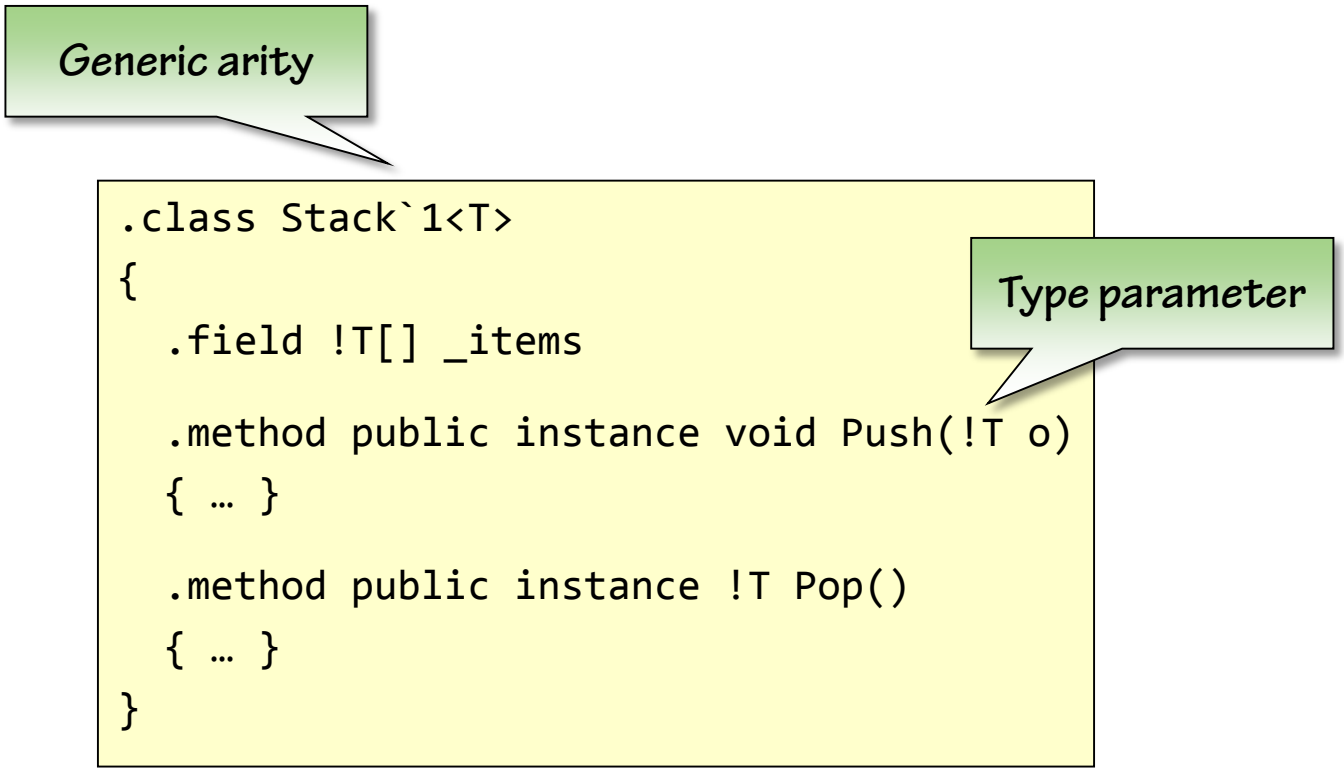
Can fail

- **Generic collections containing values**
  - Storage using specified type
  - Specialized object layouts avoid boxing
  - Static typing for the win
    - No casting that may fail

```
List<int> xs = new List<int>();
xs.Add(2);
int two = xs[0];
```

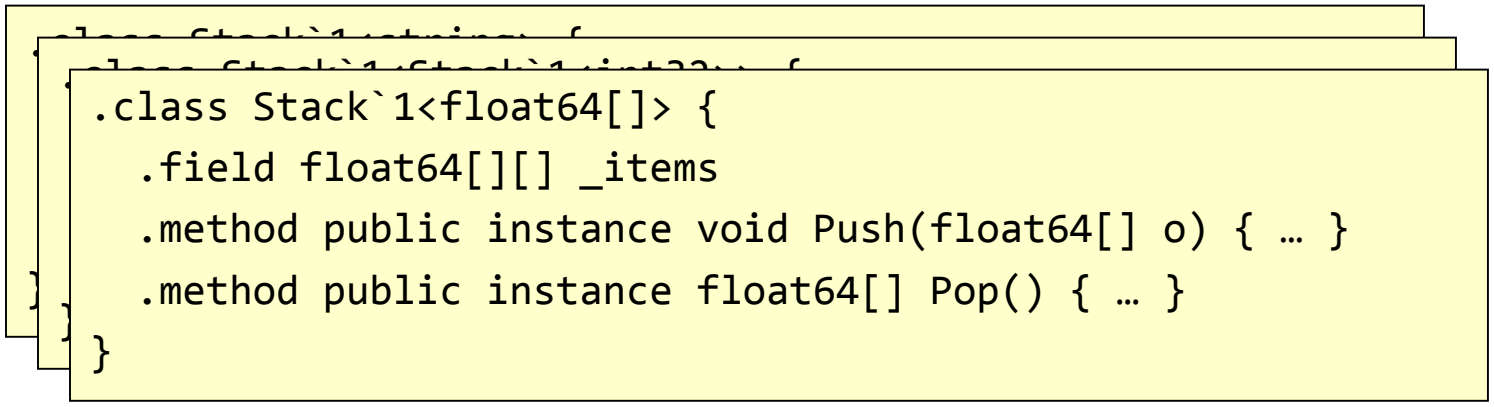
Generics Under the Hood

- **Represented in IL code**
  - Compiled by the runtime JIT



Generics Under the Hood

- **Specialized layouts at runtime**
  - Can share representation for reference type parameters
  - Specialized layouts for each struct



```
.class Stack`1<float64[]> {
  .field float64[][] _items
  .method public instance void Push(float64[] o) { … }
  .method public instance float64[] Pop() { … }
}
```

```
.class Stack`1<int32> {
  .field int32[] _items
  .method public instance void Push(int32 o) { … }
  .method public instance int32 Pop() { … }
}
```

Generics Under the Hood

- **Constrained virtual calls**
  - Virtual calls on value types require boxing
    - Access to method table
    - Typical cost on interface calls

```
static void Print<T>(T item) {
  string s = item.ToString();
}
```

Virtual method

  - Complexity for compilers
    - Decide whether type parameter is value or not?
    - New .constrained prefix for call instructions

```
.method static void Print<T>(!!T item) {
  ldarga.s item
  constrained. !!T
  callvirt string Object::ToString()
  …
```

!!T for method type parameter

Generic Constraints

- **Putting restrictions on generic type parameters**
  - Less flexibility for the caller
  - More power to the callee

```
class SortedList<T> where T : IComparable<T>
{
  …
  private int Compare(T item1, T item2)
  {
    return item1.CompareTo(item2);
  }
  …
}
```

Generic Constraints

- **Constraint types**
  - Default constructor constraint

```
where T : new()
```

Can call new T()

  - Class or struct constraint

```
where T : class
```

```
where T : struct
```

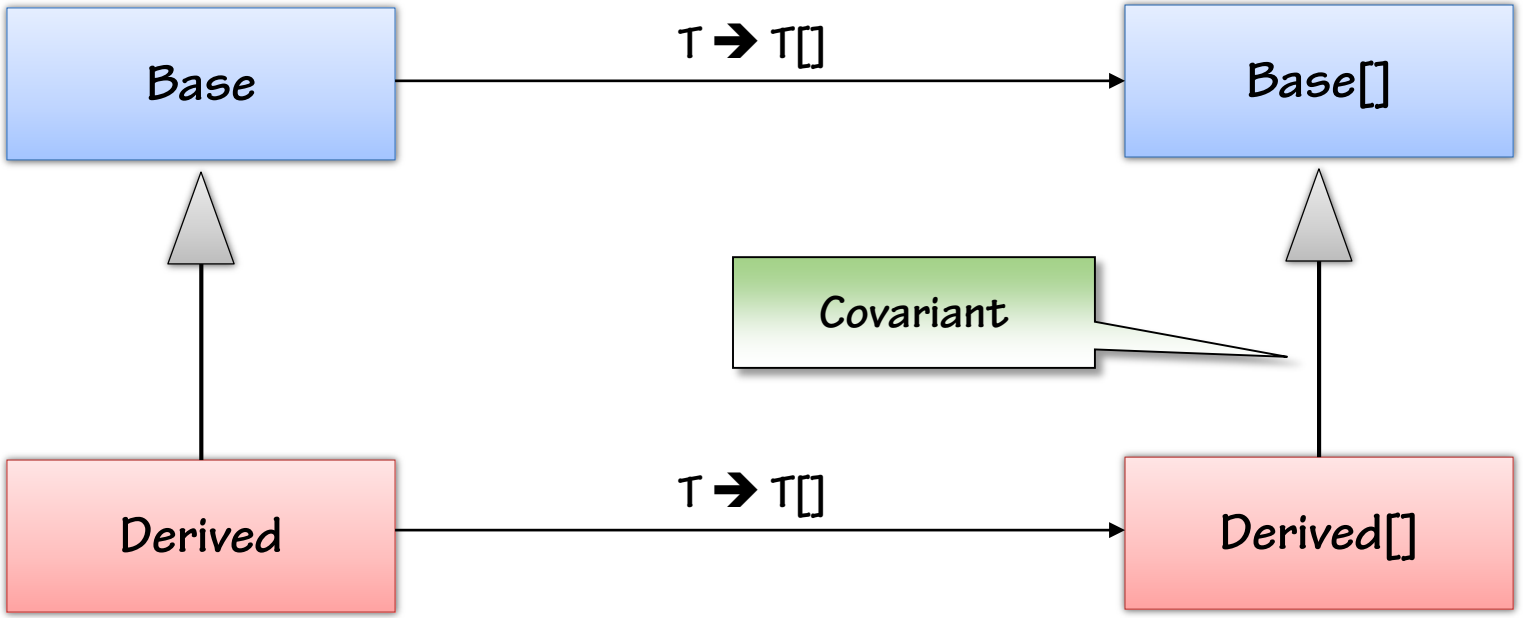
  - Base class constraint

```
where T : MyBaseObject
```

  - Interface implementation constraint

```
where T : IFoo
```

Co- and Contravariance




```
Derived d = …;
Base    b = d;
```

```
Derived[] d = …;
Base[]    b = d;
```

Type safety?

Co- and Contravariance

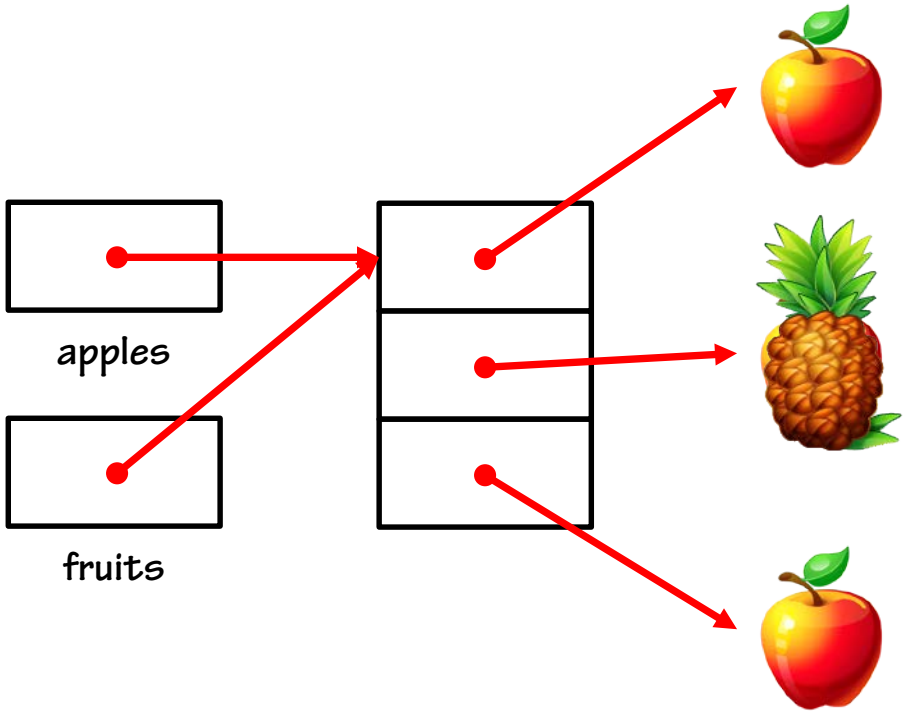
- **Broken array covariance**

```
Apple[] apples = new Apple[]
{
  new Jonagold(),
  new Gala(),
  new Fuji(),
};

Fruit[] fruits = apples;

fruit[1] = new Pineapple();

apples[1].Peel();
```



Not safe to write

Co- and Contravariance

- **For arrays**
  - Covariance is safe for reading
  - ArrayTypeMismatchException upon write

- **For generic interfaces and delegates**
  - Definition-site co- and contravariance
  - Safety via input/output restrictions

```
// Covariant
interface IReadable<out T>
{
  T Read();
}
```

```
// Contravariant
interface IWritable<in T>
{
  void Write(T input);
}
```

Co- and Contravariance

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}
```

```
interface IComparer<in T>
{
  int Compare(T t1, T t2);
}
```

```
interface IEnumerator<out T>
{
  bool MoveNext();
  T Current { get; }
}
```

```
IComparer<Fruit> fruitComp = …;
IComparer<Apple> appleComp;

appleComp = fruitComp;
```

Safe to write

```
IEnumerable<Apple> apples = …;
IEnumerable<Fruit> fruits;

fruits = apples;
```

Safe to read

Summary

- **Generic typing benefits**
  - Compile-time checking
  - Avoid runtime costs, e.g. boxing
  - Co- and contravariance

- **Behind the scenes**
  - First-class concept in IL
    - Type parameters
    - Constraints
  - Specialization of code and layouts by JIT
    - Sharing for reference types
    - System.__Canon
    - Constrained virtual calls
  - Variance checks for arrays