




C# PROGRAMMING 2007

Generics

Thomas J. Fuchs



Parametric Polymorphism (Generics)

- Benefits:
 - Code reuse
 - Faster code (no runtime casts)
 - Safer programming (static type-checking)
 - World's first cross-language generics (not just for C#, but C++ and VB and other languages running on the CLR)
 - For use in:
 - Collection classes
 - Algorithms
 - Structured types
- 

Generics

- Why generics?
 - Type checking, no boxing, no downcasts
 - Increased sharing (typed collections)
- How are C# generics implemented?
 - Instantiated at run-time, not compile-time
 - Checked at declaration, not instantiation
 - Work for both reference and value types
 - Exact run-time type information

Implementation Details

- Type-checked at declaration, not at use (unlike C++ templates)
- After type check, translated to Generic IL bytecode which supports explicit type parameters and validation of generic code



|| The Most Simple Example

```
List<T> myList = new List<T>();
```



|| The Most Simple Example

```
List<int> myList = new List<int>();
```



Placeholders for Generic Types

- T ... type
- K ... key
- V ... value



Generics

- Collection classes
- Collection interfaces
- Collection base classes
- Utility classes
- Reflection

List<T>
Dictionary<K,V>
SortedDictionary<K,V>
Stack<T> (*LIFO*)
Queue<T> (*FIFO*)

IList<T>
IDictionary<K,V>
ICollection<T>
IEnumerable<T>
IEnumerator<T>
IComparable<T>
IComparer<T>

Collection<T>
KeyedCollection<T>
ReadOnlyCollection<T>

Nullable<T>
EventHandler<T>
Comparer<T>

The default Keyword in Generic Code

```
public void ResetPoint()  
{  
    xpos = default(T);  
    ypos = default(T);  
}
```

- Numerical values have a default value of 0.
- Reference types have a default value of `null`.
- Fields of structures are set accordingly to 0 or `null`.

Constrain Type Parameters Using where

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

Unbound type parameters are assumed to have only the members of `System.Object`.

Constraints for Generic Type Parameters

- `where T : struct`
- `where T : class`
- `where T : new()`
- `where T : NameOfBaseClass`
- `where T : NameOfInterface`

Iterators

- Methods that incrementally compute and return a sequence of values

```
class Program
{
    static IEnumerable<int> Range(int start, int count) {
        for (int i = 0; i < count; i++) yield return start + i;
    }

    static IEnumerable<int> Squares(IEnumerable<int> source) {
        foreach (int x in source) yield return x * x;
    }

    static void Main() {
        foreach (int i in Squares(Range(0, 10))) Console.WriteLine(i);
    }
}
```

0
1
4
9
16
25
36
49
64
81

Notes

- Can implement more than one iterator

```
□ public IEnumerable<T> BottomToTop {  
    get {  
        for (int i = 0; i < count; i++) {  
            yield return items[i];  
        }  
    }  
}
```

```
□ public IEnumerable<T> TopToBottom {  
    get {  
        return this;  
    }  
}
```

|| Additional Functionality

- Generic Base Classes
Can implement virtual or a abstract methods.
- Generic Interfaces
- Generic Delegates





PRACTICAL EXAMPLES ...



QUESTIONS ?

Slides partly based on a presentation from:

Anders Hejlsberg

“C# 2.0 and Future Directions”

|| Resources

- Course Homepage:
- **se.inf.ethz.ch/teaching/ss2007/251-0290-00**
- Exercise Material:
- **www.inf.ethz.ch/personal/thomas.fuchs**

Generics

```
public class List<T>
{
    private T[] elements;
    private int count;

    public void Add(T element) {
        if (count == elements.Length) Resize(count * 2);
        elements[count++] = element;
    }

    public T this[int index] {
        get { return elements[index]; }
        set { elements[index] = value; }
    }

    public int Count {
        get { return count; }
    }
}
```

Generics

```
List<int> intList = new List<int>();
```

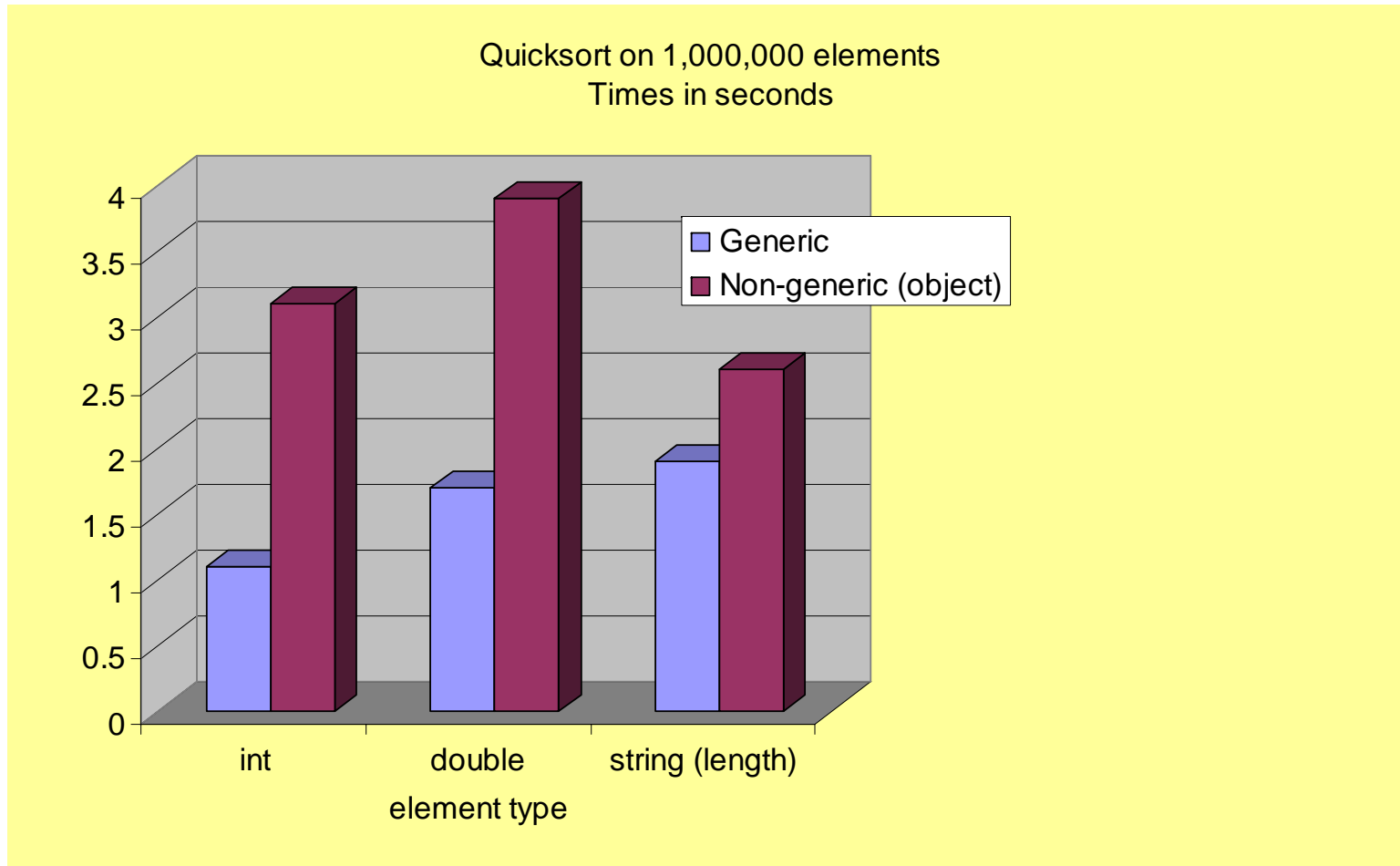
```
intList.Add(1);           // No boxing
```

```
intList.Add(2);           // No boxing
```

```
intList.Add("Three");     // Compile-time error
```

```
int i = intList[0];       // No cast required
```

Performance



Generics

```
public class List<T>
{
    private T[] elements;
    private int count;

    public void Add(T element) {
        if (count == elements.Length) Resize(count * 2);
        elements[count++] = element;
    }

    public T this[int index] {
        get { return elements[index]; }
        set { elements[index] = value; }
    }

    public int Count {
        get { return count; }
    }
}
```

```
List<int> intList = new List<int>();

intList.Add(1);           // No boxing
intList.Add(2);           // No boxing
intList.Add("Three");     // Compile-time error

int i = intList[0];       // No cast required
```