# C# 2.0 Generic Types and Methods

Peter Sestoft
KVL and IT University of Copenhagen

**C#/.Net project cluster**
**Tuesday 3 May 2005**

## History of generics in programming languages

The theory of generic types (parametric polymorphism) is by Hindley (1968) and Milner (1977).

First programming language with parametric polymorphism is ML (1979); then Miranda, Haskell, Clean, . . .

First object-oriented language with generics is Eiffel (1991).

**Generics in Java**

- PolyJ (Myers, Bank, Liskov; 1997):
  Type parameters can be instantiated by reference types and primitive types; requires an extended JVM.
- Generic Java (Bracha, Odersky, Stoutamire, Wadler 1998):
  Became Java 5.0 generics (plus wildcards, due to researchers at Aarhus University); runs on standard JVM.
- NextGen (Cartwright, Steele; 1998):
  Type parameters can be instantiated by reference types, not primitive types; runs on standard JVM.

**Generics in C#**

- Generic C# and new Generic Common Language Runtime (Kennedy and Syme, Microsoft Research Cambridge UK, 2001).
- In November 2002, Microsoft announced generics for next version of C#; Redmond had been convinced . . .
- In August 2003, first alpha version of .Net Common Language Infrastructure with generics released.

## Why generic types and methods?

Because the old collection classes are dynamically typed: Code may compile OK, then fail at run-time:

```
ArrayList cool = new ArrayList();
cool.Add(new Person("Kristen"));
cool.Add(new Person("Bjarne"));
cool.Add(new Exception("Larry"));      // Wrong, but no compile-time check
cool.Add(new Person("Anders"));
Person p = (Person)cool[2];            // Compiles OK, but fails at run-time
```

With generic types, collections can be statically typed: errors are detected at compile-time:

```
List<Person> cool = new List<Person>();
cool.Add(new Person("Kristen"));
cool.Add(new Person("Bjarne"));
cool.Add(new Exception("Larry"));      // Wrong, detected at compile-time
cool.Add(new Person("Anders"));
Person p = cool[2];                    // No run-time check needed
```

## Using a generic class or interface

This works just as in Java 5.0:

```
IMyList<String> cities = new LinkedList<String>("Oslo", "Seattle");
String wa = cities[1];
```

Unlike in Java, type arguments can be value types, not only reference types:

```
Pair<String, int> p = new Pair<String, int>("Carsten", 1964);
int year = p.Snd;
```

No boxing or unboxing is needed for value type arguments; hence better performance and less memory usage.

## Example: Enumerators and enumerables

A C# enumerator is similar to a Java iterator, and an enumerable is similar to a Java 5.0 iterable.

An enumerator over type T has a current element, can get the next one, and can release resources:

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

An enumerable over type T can produce an enumerator over T:

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

## Example: Comparables and comparers

An comparable for type T can compare itself to another value of type T (like a Java comparable):

```
interface IComparable<T> {
  int CompareTo(T that);
  // bool Equals(T that);
}
```

A comparer for type T can compare two values of type T (like a Java comparator):

```
interface IComparer<T> {
  int Compare(T v1, T v2);
  // bool Equals(T v1, T v2);
  // int GetHashCode(T v);
}
```

(The Microsoft design mistake of including `Equals` and `GetHashCode` has been corrected in beta 2.)

Example: A time of day `(hh, mm)` can be compared to another a time of day:

```
public class Time : IComparable<Time> {
  private readonly int hh, mm;                  // 24-hour clock
  public Time(int hh, int mm) { this.hh = hh; this.mm = mm; }
  public int CompareTo(Time that) {
    return hh != that.hh ? hh - that.hh : mm - that.mm;
  }
  public bool Equals(Time that) { return hh=that.hh && mm==that.mm; }
}
```

## Declaring a generic class

An object of class `LinkedList<T>` is a linked list with elements of type T:

```
public class LinkedList<T> : IMyList<T> {
  protected Node first, last;
  protected class Node {
    public Node prev, next;                       // Static member class
    public T item;                                // T used in static member
  }
  public LinkedList(params T[] arr) : this() {   // Variable-arity argument
    foreach (T x in arr)                          // Iterate over array arr
      Add(x);
  }
  public int Count { get { return size; } }      // Property
  public T this[int i] { ... }                   // Indexer
  public override bool Equals(Object that) {     // Equality; exact type test
    if (that != null && GetType() == that.GetType() && ...) { ... }
  }
  public IMyList<U> Map<U>(Mapper<T,U> f) { ... }
  ... more ...
}
```

Type parameters can be used also in static members. Each type instance has its own copy of the static fields.

There is a type object at run-time for every type, even for generic type instances (this is used in `GetType`).

Types are overloaded on the number of type parameters, so classes `C` and `C<T>` and `C<T,U>` can co-exist.

**Declaring a generic interface — very similar to Java**

Interface `IMyList<T>` describes lists with elements of type `T`:

```
public interface IMyList<T> : IEnumerable<T> {
  int Count { get; }                   // Number of elements
  T this[int i] { get; set; }          // Get or set element at index i
  void Add(T item);                    // Add element at end
  void Insert(int i, T item);          // Insert element at index i
  void RemoveAt(int i);                // Remove element at index i
  IMyList<U> Map<U>(Mapper<T,U> f);    // Map f over all elements
}
```

As in Java, generic types are invariant in the type parameters.

So `IMyList<String>` is not a subtype of `IMyList<Object>`,
although `String` is a subclass of `Object`.

Only the declared subtype relations hold: `IMyList<T>` is a subtype of `IEnumerable<T>`, and
`LinkedList<T>` is a subtype of `IMyList<T>`.

**Declaring a generic struct type — very similar to a generic class**

Struct type `Pair<T,U>` is the type of pairs of a `T` and a `U`:

```
public struct Pair<T,U> {
  public readonly T Fst;
  public readonly U Snd;
  public Pair(T fst, U snd) {
    this.Fst = fst;
    this.Snd = snd;
  }
}
```

**Using a generic struct type**

Declaring `appointments` to be an array of pairs of `Time` and `String`:

```
Pair<Time, String>[] appointments;
```

In contrast to Java, one can use generic type instances just like any other types.

Thus one may create an array whose element type is a generic type instance:

```
appointments = new Pair<Time, String>[100];
```

**Declaring a generic delegate type**

A delegate of generic delegate type `Mapper<A,R>` takes an argument of type `A` and returns a result of type `R`:

```
public delegate R Mapper<A,R>(A x);
```

The type parameters are given after the delegate type's name, as for classes, interfaces, structs and methods.

**Using a generic delegate type**

Method `int Sign(double)` from class `Math` can be turned into a delegate:

```
Mapper<double,int> sign = new Mapper<double,int>(Math.Sign);
```

**Declaring a generic method**

As in Java, a method can take type parameters.

Example: `Map<U>` in `LinkedList<T>` creates a new list by applying `f` to every element of the given list:

```
public class LinkedList<T> : IMyList<T> {
  public IMyList<U> Map<U>(Mapper<T,U> f) {     // Map f over all elements
    LinkedList<U> res = new LinkedList<U>();
    foreach (T x in this)
      res.Add(f(x));
    return res;
  }
  ...
}
```

**Calling a generic method**

The type parameters of a generic method may be given explicitly, but often they can be inferred automatically:

```
list.Map<int>(...);
list.Map(...);
```

**Type parameter constraints**

As in Java, the type parameters of a class (or struct type or interface or method) can be constrained.

Example: A printable linked list is a linked list whose elements are printable:

```
class PrintableLinkedList<T> : LinkedList<T>, IPrintable
  where T : IPrintable
{
  public void Print(TextWriter fs) {
    foreach (T x in this)
      x.Print(fs);
  }
}
interface IPrintable { void Print(TextWriter fs); }
```

As in Java, a type parameter constraint may involve the type parameter itself.

Example: An array of T can be sorted if T-values are comparable to T-values:

```
private static void Qsort<T>(T[] arr, int a, int b)
  where T : IComparable<T>
{ ... }
```

**Multiple type parameter constraints**

Struct type `ComparablePair<T,U>` is the type of pairs of comparable T and comparable U:

```
struct ComparablePair<T,U> : IComparable<ComparablePair<T,U>>
  where T : IComparable<T>
  where U : IComparable<U>
{
  public readonly T Fst;
  public readonly U Snd;
  public int CompareTo(ComparablePair<T,U> that) {    // Lexicographic ordering
    int firstCmp = this.Fst.CompareTo(that.Fst);
    return firstCmp != 0 ? firstCmp : this.Snd.CompareTo(that.Snd);
  }
  public bool Equals(ComparablePair<T,U> that) {
    return this.Fst.Equals(that.Fst) && this.Snd.Equals(that.Snd);
  }
  ...
}
```

**Special kinds of type parameter constraints**

C# permits several special constraints on a type parameter T:

| Constraint | Meaning |
|---|---|
| `T : t` | When t is a type: T must be subclass of (class) t or implement (interface) t |
| `T : class` | T must be a reference type |
| `T : struct` | T must be a (non-nullable) value type |
| `T : new()` | T must have an argumentless constructor; always holds for a value type |

Example: A field of type T be `null` if T is a reference type:

```
class C1<T> where T : class {
  T f = null;                           // Legal: T is a reference type
}
```

Example: One can call `new T()` only if type T has an argumentless constructor:

```
class C1<T> where T : new() {
  T f = new T();                        // Legal: T() exists
}
```

More generally, `default(t)` is `null` for a reference type t, and is `new t()` for a struct type t.

**What can type parameters be used for**

In contrast to Java, a type parameter can be used almost as an ordinary type:

```
class C<T> {
  void M(Object o) {
    T[] arr = new T[10];             // Array creation
    if (o is T) {                    // Instance-of test
      T t = (T)o;                    // Type cast
      ...
    }
    T d = default(T):                // Get default value for T
    Type ty = typeof(T);             // Get type object (reflection)
  }
  void MO(T x) { ... }               // Overloading on type parameters
  void MO(IMyList<T> x) { ... }      // and type instances
}
```

However:

One cannot call static members of a type parameter T.

One can create an instance of T using `new T()` only if T has the `new()` constraint or the `struct` constraint.

One can use `null` as a variable of type T only if T has the `class` constraint.

## Comparison of generics in Java 5.0 and C# 2.0

| Property | Java | C# |
|---|---|---|
| Can use type parameters in static member declarations | No | Yes |
| Static members are shared between type instances | Yes | No |
| Wildcard type arguments permitted | Yes | No |
| All type instances have a common supertype ('raw type') | Yes | No |
| Compiler may emit 'unchecked' (= I don't know) warnings | Yes | No |
| Type parameters can be instantiated with simple types (`int` ...) | No | Yes |
| Can overload a method on different type instances of same generic type | No | Yes |
| Exact type arguments exist at run-time | No | Yes |
| Can perform instance-of check against type parameter or type instance | No | Yes |
| Can cast to type parameter `(T)e` or type instance `(IMyList<int>)e` | No | Yes |
| Can create (`new`) object whose type is a type parameter or type instance | No | Yes |
| Can create (`new`) array whose element type is a type parameter or type instance | No | Yes |
| Can declare array variable whose element type is a type parameter | Yes | Yes |

## Why Java cannot create an array whose element type is constructed from a generic type

Java and C# array assignment requires runtime type checks:

```
static void m(Object[] arr, Object x) {
  arr[0] = x;                       // Runtime check needed
}
```

Why? Observe that `String` is a subclass of `Object`, then execute:

```
Object[] arr = new String[10];
m(arr, new Object());               // MUST fail at run-time
... otherwise arr[0] now contains an Object, not String, bad ...
```

The exact element type (`String`) of the array `arr` is needed to check the assignment in `m(...)`.

## Lack of exact runtime types (in Java 5.0) makes runtime type check impossible

This in turn makes it impossible to create an array whose element type is a constructed type:

```
Pair<String,Integer>[] heights = new Pair<String,Integer>[10];
```

This is OK in C# 2.0 because the array element type can be stored in `heights`.

It is not OK in Java 5.0, because the runtime has no presentation of `Pair<String,Integer>`.

Java workaround: Use `ArrayList<t>` instead of `t[]`. (Question: Why does this work?)

## Simulating the wildcard type parameter (`<?>`) from Java in C#

A wildcard type (`<?>`) in Java is similar to an unnamed bound type parameter, not used anywhere else.

When used in method parameter declarations it can sometimes be simulated in C# using extra type parameters `T`:

| Context | Java | C# |
|---|---|---|
| Unbounded wildcard | `tr m(C<?> x)` | `tr m<T>(C<T> x)` |
| Bounded wildcard | `tr m(C<? extends t> x)` | `tr m<T>(C<T> x) where T : t` |

Wildcards used in declarations of variables and fields can sometimes be simulated.

This makes some things more complicated in C#, but it seems possible to work around the limitations.

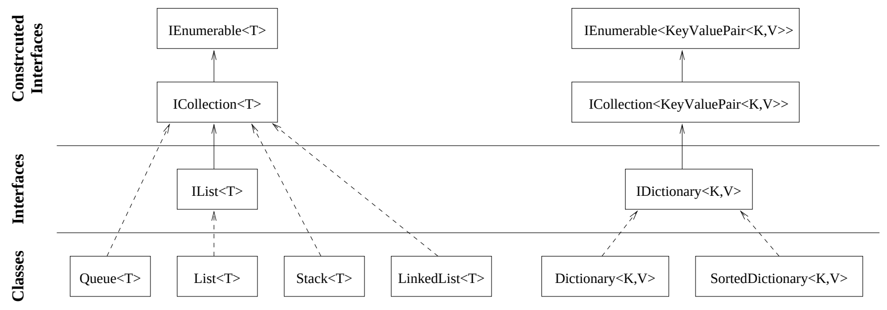
The wildcard `<? super t>` can sometimes be simulated in C#:
introduce a type parameter T for `t` and another type parameter U for ?, and then constrain T : U.

An attempt to do this for Java's `Collections.binarySearch` crashed Microsoft's beta 1 compiler:

```
public static int BinarySearch<T,U,S>(List<S> lst, T k)
  where T : U, IComparable<U>
  where S : T
{ ... }
```

## The standard generic collection classes (C# Precisely section 24)

Namespace System.Collections.Generic provides some generic collection classes:



## Critique of standard generic collection classes

- Only lists and hashtables; no (sorted) tree-based sets or dictionaries.

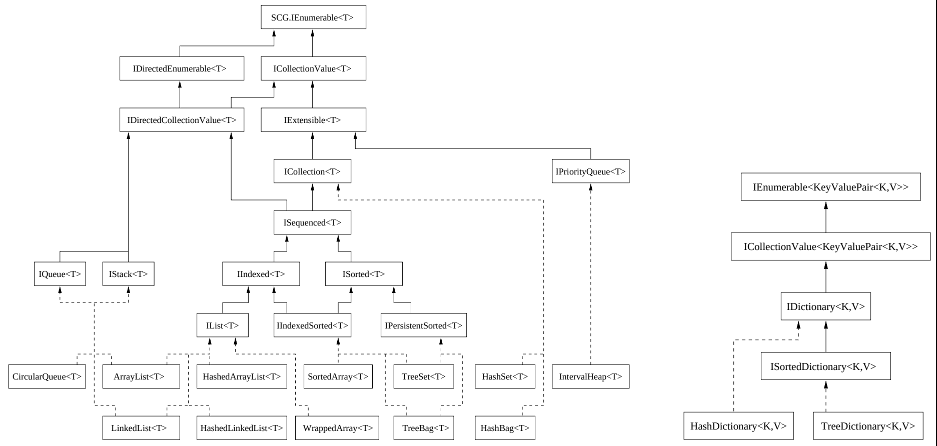
  (But red-black trees will be included later, according to a Microsoft CLI team source.)
- Proliferation of methods and lack of orthogonality: Three versions (entire list, tail of list, segment of list) of each of `CopyTo`, `FindIndex`, `FindLastIndex`, . . . ; but not so for other methods.
- Strange interfaces: `IComparer<T>` describes `Compare(T,T)` but also `GetHashCode(T)` and `Equals(T,T)` — invites 'dishonest' implementations.

  Luckily, this has been changed in beta 2: `IEqualityComparer<T>` and `IComparer<T>`.
- Array-based lists and linked lists do not have a common interface.
- Low level of abstraction: `LinkedList<T>` requires working on list nodes; invariants (e.g., every list is acyclic) must be enforced by run-time checks, cannot be checked at compile-time.
- Some methods are virtual while others are non-virtual (for efficiency); risky and confusing.
- Potential performance traps, such as array-based `SortedDictionary<K,V>`.

  Performing $n$ random insertions would take time $O(n^2)$.

Luckily, much of this was withdrawn from Ecma CLI standardization.

## C5: Copenhagen Comprehensive Collection Classes for C#



## Some highlights of C5

- Comprehensive interfaces support 'program to an interface, not an implementation'.
- Use best known data structures and algorithms, even if cumbersome to implement.
- Consider asymptotics (scalability) more important than nanosecond efficiency.
- Updatable views (sublists) of lists; ensures orthogonality of operation and range.
- Range queries by index (indexed collections) and by elements (sorted collections).
- Reversible enumeration, also of views.
- Constant-time snapshots of red-black trees (persistent trees); supports geometric algorithms.
- Supports both hash-indexes and views of a linked list.
- Introspective quicksort for arrays; worst-case running-time logarithmic.
- In-place smooth stable mergesort for doubly-linked lists.

Developed by Niels Kokholm and Peter Sestoft with support from Microsoft Research University Relations.

C5 is and will remain freely available from `http://www.itu.dk/research/c5/`

C5 is included in the Mono project implementation of C#/CLI.

Peter Golde of Wintellect, formerly Microsoft, is developing PowerCollections, another collection class library.