Java and C#

Generic Types and Methods

Peter Sestoft
KVL and IT University of Copenhagen

DIKU Generative Programming

Tuesday 31 January 2006

- Java and C#, object-oriented languages with managed execution
- History of generics in programming languages
- Why generic types and methods?
- Generics in C#
  - Using and declaring generic types and methods
  - Type parameter constraints
  - Implementation
- Generics in Java
  - Example declarations and uses
  - Wildcards
  - Implementation
  - Limitations. Differences between Java 5.0 and C# generics
- The C5 comprehensive collection class library

Your teacher

MSc (1988) and PhD (1991) in computer science from DIKU.

At DTU 1992–1994; at AT&T Bell Labs USA 1994–1995; at KVL since 1995 and at ITU since 1999.

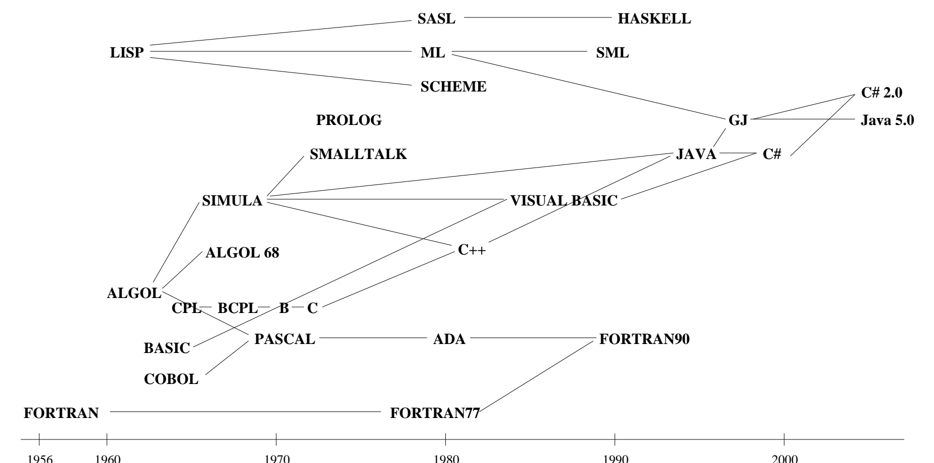
Moscow ML and Standard ML Basis Library since 1994.

Member of the Ecma standardization committees for C# and CLI (Common Language Infrastructure).

Books:

- Jones, Gomard, Sestoft: *Partial Evaluation and Automatic Program Generation* (Prentice-Hall 1993).
- Sestoft: *Java Precisely* (MIT Press 2002, second edition 2005).
- Sestoft and Hansen: *C# Precisely* (MIT Press 2004).

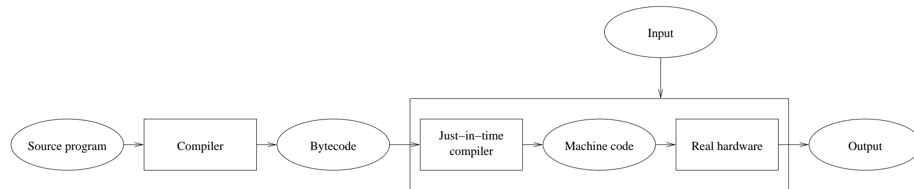
Genealogy of some programming languages



Java and C# execution model

Execution requires three steps (although 2 and 3 may be interleaved):

1. Compile source code (Java, C#) to bytecode (JVM, CIL).
2. Load and verify bytecode, and compile it to machine code (x86, Sparc, MIPS, . . .).
3. Execute machine code.



This model can give very fast code, sometimes faster than that generated by `gcc -O3`.

Comparison of Java, C#, C++ and C

Feature	Java	C#	C++	C
Automatic memory management	+	+	−	−
Exceptions	+	+	+	−
Array bounds checks	+	+	−	−
Classes	+	+	+	−
Structs (value types)	−	+	+	+
Interfaces	+	+	−	−
Inheritance	+	+	+	−
Multiple inheritance	−	−	+	−
Virtual methods	+	+	+	−
Non-virtual methods	−	+	+	−
Method overloading	+	+	+	−
Nested classes	+	+	+	−
Inner classes	+	−	−	−
Call-by-value parameters	+	+	+	+
Reference parameters	−	+	+	−
Safe variable-arity methods	5.0	+	−	−
Properties	−	+	−	−
Indexers	−	+	+	−
Looping over iterators	5.0	+	+	−
Defining iterators (`yield`)	−	2.0	−	−
User-defined operators	−	+	+	−
User-defined conversions	−	+	−?	−
Enum types	5.0	+	+	+
Autoboxing simple values	5.0	+	−	−
Nullable value types	−	2.0	−	−
Rectangular arrays	−	+	+	+
Arrays of arrays	+	+	(−)	(−)
Compile-time conditional code	−	+	+	+
Generic types and methods	5.0	2.0	(+)	−
Runtime type parameter info	−	2.0	−	−
Wildcard types (existentials)	5.0	−	−	−
Generic collection library	5.0	(−)	(+)	−
Anonymous methods	(−)	2.0	−	−
Metadata (attributes/annotations)	5.0	+	−	−

Example translation from source code to bytecode to machine code (Mono gmcs and JIT)

```
Source code (C#)             Bytecode (CIL)                Machine code (x86)
---------------------------------------------------------------------------------
for (int i=0; i<100; i++)    IL_0008: ldc.i4.0             16: xor  %esi,%esi
  arr[i] = 117*i;            IL_0009: stloc.1              18: jmp  2a <MyTest_Main+0x2a>
                             IL_000a: br        IL_001a    1a: lea  0x0(%ebp),%ebp
                             IL_000f: ldloc.0              20: lea  0x10(%edi,%esi,4),%eax
                             IL_0010: ldloc.1              24: imul $0x75,%esi,%ecx
                             IL_0011: ldc.i4.s 117         27: mov  %ecx,(%eax)
                             IL_0013: ldloc.1              29: inc  %esi
                             IL_0014: mul                  2a: cmp  $0x64,%esi
                             IL_0015: stelem.i4            2d: jl   20 <MyTest_Main+0x20>
                             IL_0016: ldloc.1
                             IL_0017: ldc.i4.1
                             IL_0018: add
                             IL_0019: stloc.1
                             IL_001a: ldloc.1
                             IL_001b: ldc.i4.s 100
                             IL_001d: blt       IL_000f
```

Variables and intermediate results

Source code	Bytecode	Machine code
`i`	local 1	`%esi`
`arr`	local 0	`%edi`
`arr[i]` address	(stack)	`%eax`

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 in C#

A generic type takes one or more type parameters:

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

In C#, 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.

Polymorphic types are well known from Standard ML

```
- val cities = ["Oslo", "Seattle"];
> val cities = ["Oslo", "Seattle"] : string list
- List.nth(cities, 1);
> val it = "Seattle" : string
- val p = ("Carsten", 1964);
> val p = ("Carsten", 1964) : string * int
- val year = #2 p;
> val year = 1964 : int
```


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);
}
```

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);
}
```

Example use: 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;
  }
}
```

A generic sort routine can sort any array with elements of type `T`, when `T` implements `IComparable<T>`.

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

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
}
```

Generic types are invariant in the type parameters (in C# and in Java).

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

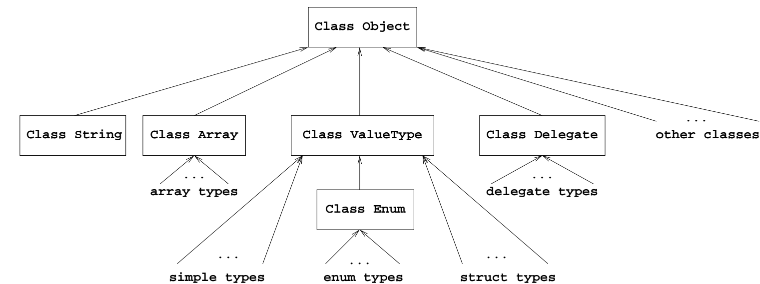
Only the declared subtype relations hold: `IMyList<T>` is a subtype of `IEnumerable<T>`, and `LinkedList<T>` is a subtype of `IMyList<T>`.

C# value types and reference types

A type is either a *reference type* (class, interface, array type) or a *value type* (`int`, `double`, ...).

- A value (object) of reference type is always stored in the managed (garbage-collected) heap.
  Assignment to a variable of reference type copies only the reference.
- A value of value type is stored in a local variable or parameter, or inline in an array or object or struct value.
  Assignment to a variable of value type copies the entire value.

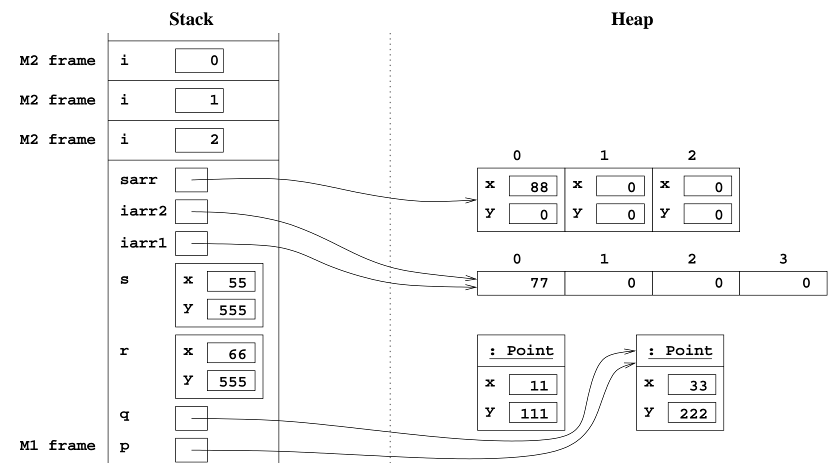
Just as in Java. But in C#, there are also user defined value types, called struct types (as in C/C++):



The machine model

Class instances (objects) are individuals in the heap.

Struct instances are in the stack, or inline in other structs, objects or arrays.



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;
```

One can use a generic type instance just like any other type, for instance as array element type.

Also, 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.

Think of this as a strange way to write this Standard ML type declaration:

```
type ('a, 'r) Mapper = 'a -> 'r
```

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);
Mapper<int,int> triple = delegate(int x) { return 3*x; };
```

These delegate bindings can be written like this in Standard ML:

```
- val sign = Real.sign;
> val sign = fn : real -> int
- val triple = fn x => 3*x;
> val triple = fn : int -> int
```


Declaring a generic method

In C# (as in Standard ML) 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

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); }
```

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);
  }
  ...
}
```


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 can be null only 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 C#, 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.

One can construct the nullable type T? only if T has the struct constraint.

Implementation of C# generics

C# generics have very different static and dynamic semantics than C++ templates:

- In C++, a template class or function is typechecked only at type instantiation.

  Type instances of a templates are created at compile-time, one copy of the template for each type instance.

- In C#, a generic class or method is typechecked at declaration.

  The run-time system (CLI) directly supports generics, and creates a new type for each type instantiation.

  Field layouts and code are shared as far as possible between type instances.

  Thus, List<String> and List<Object> would share layout and code, but not with List<int>.

  > "When the runtime requires a particular instantiation of a parameterized class, the loader checks to see if the instantiation is compatible with any that it has seen before; if not, then a field layout is determined and new vtable is created, to be shared between all compatible instantiations. The items in this vtable are entry stubs for the methods of the class. When these stubs are later invoked, they will generate code to be shared for all compatible instantiations."
  >
  > Kennedy and Syme: Design and Implementation of Generics for .Net ... (2001)

- C# generics do not have the speed advantages of C++ templates, but better typesafety and less code bloat.

Polymorphic recursion

In Standard ML, a function can call itself only with the same type of argument that it was called with:

```
fun f xs =
  if length(xs) > 20 then
    f [xs]            <== ILLEGAL; xs : ’a list, but [xs] : ’a list list
  else
    117
```

C# does not have this restriction. But then the number of type instances cannot be determined at compile-time.

Let S be this generic struct type:

```
struct S<T> {
  public int i;
  public T x;
}
```

Calling M<double>(i, 3.14) creates i type instances of S<T>:

```
static void M<U>(int i, U s) {
  if (i > 0)
    M<S<U>>(i-1, new S<U>());
}
```

Namely, S<double>, S<S<double>>, S<S<S<double>>>, ...

Some hairy (functional) uses of C# generics

Two C# delegate type definitions and the correspond Standard ML types:

```
public delegate R Fun<A1,R>(A1 x);                  // ’a1 -> ’r
public delegate R Fun<A1,A2,R>(A1 x1, A2 x2);       // ’a1 * ’a2 -> ’r
```

The function compose : (’a -> ’b) * (’c -> ’a) -> ’c -> ’b from Standard ML:

```
fun compose (f, g) x = f (g x);
```

can be written like this in C#:

```
public static Fun<A,R> Compose<A,I,R>(Fun<I,R> f, Fun<A,I> g) {
  return delegate(A x) { return f(g(x)); };
}
```

The function curry : (’a * ’b -> ’c) -> ’a -> ’b -> ’c from Standard ML:

```
fun curry f x y = f (x, y);
```

can be written like this in C#:

```
public static Fun<A,Fun<B,C>> Curry<A,B,C>(Fun<A,B,C> f) {
  return delegate(A x) {
    return delegate(B y) {
      return f(x, y);
    };
  };
}
```


Another useful function

And here’s a call-by-value fixed-point combinator of type ((A -> R) -> (A -> R)) -> (A -> R):

```
public static Fun<A,R> Fix<A,R>(Fun<Fun<A,R>, Fun<A,R>> f) {
  Fun<A,R> res = null;
  res = delegate(A x) { return f(res)(x); };
  return res;
}
```

The explicit types (in C# and Java) can become rather verbose

Method declaration in an algorithm to convert NFAs to DFAs:

```
static IDictionary<int, IDictionary<String, int>>
   Rename(IDictionary<Set<int>, int> renamer,
          IDictionary<Set<int>, IDictionary<String, Set<int>>> trans)
{ ... }
```

More examples of C# generics at http://www.dina.kvl.dk/~sestoft/gcsharp/.

Generic types and methods in Java

Most concepts are the same as in C#, but:

- Small syntactic differences:

```
class C<T extends B> {                        // "extends" not ":"
  public void <T> m(T x) { ... }              // type parameter position
}
```

- Java permits so-called wildcard type arguments.
- In Java, a type argument must be a reference type (Integer, String) not primitive (int, double).
- In Java, all type instances of a generic type share the same static fields and methods.
- In Java, there are many restrictions on the use of type parameters and type instances.

Generic types (in Java and C#) are invariant in the type parameters

Class String is a subtype of Object, **but** List<String> is not a subtype of List<Object>.

We say that type List<T> is not *covariant* in type parameter T.

But why?

Because covariance is unsound

Assume Sedan is subclass of Car is subclass of Vehicle is subclass of Object.

Consider:

```
public void InsertCar(List<Car> cars) {
  cars.InsertFirst(new Car("MX5"));
}

...
List<Sedan> list = new List<Sedan>();
InsertCar(list);
```

After this, list[0] is not a Sedan, but a Car. So passing a List<Sedan> should be (and is) forbidden.

Exercise: Find an example to show that contravariance is unsound.

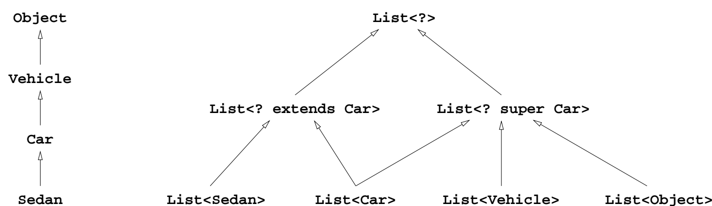
Java generics: Wildcard type arguments

Note that it would have been OK to pass a List<Vehicle> to InsertCar.

Wildcard type argument give additional flexibility.

- If a method only *reads* from a List<Car>, then it is safe to pass it a List<Sedan>.
- If a method only *writes* to a List<Car>, then it is safe to pass it a List<Vehicle>.

This can be expressed in Java using wildcard type arguments:



So in Java, method InsertCar might have had the following argument type:

```
void InsertCar(List<? super Car>} { ... }
```

This says it is OK to pass List<Car> or List<Vehicle> but not List<Sedan> to InsertCar.

A wildcard example from the Java class library

Class Collections in the Java class library has this fancy method:

```
<T extends Comparable<? super T>> int binarySearch(List<? extends T> xs, T x) { ... }
```

What does this mean?

- Type parameter T must be a subtype of Comparable<? super T>.

  So T or a supertype of T must have a compareTo whose argument type must be T or a supertype of T.
- xs may be any List whose element type is a subtype of T.

Assume that class Vehicle implements compareTo(Vehicle).

Then binarySearch could be applied to a List<Sedan> and a Car.

Namely, T would be Car, which has a supertype (Vehicle) satisfying the first requirement.

And, in the type of xs, Sedan is a subtype of Car, satisfying the second requirement.

(Mads Torgersen, who co-invented wildcards for Java generics, now works in the C# team at Microsoft).

Implementation of Java generics — simpler than C#/CLI

In Java, generic types exist only at compile-time: javac knows generics, JVM does not.

At run-time (in the JVM):

- All type instances of a generic type C<T> are represented by a single type, the *raw type* C.
- A generic type parameter is replaced by Object, or by its constraint bound, if any.

It follows that in Java, a generic type parameter in many respect **cannot** be used as an ordinary type:

```
class C<T> {
  void m(object o) {
    T[] arr = new T[10];              // Declaration OK, array creation not
    if (o instanceof T) {             // No instance-of test
      T t = (T)o;                     // Type casts are "unchecked"
      ...
    }
    Class ty = T.class;               // No getting the type object
  }
  void mo(T x) { ... }                // No overloading on type parameters
  void mo(MyList<T> x) { ... }        // and type instances
}
```

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 really 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

This makes some things more complicated in C#.

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

Here's how that would work for Java's Collections.binarySearch:

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

Clearly not the standard use of constraints; provoked internal compiler error in Microsoft's beta 1.

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 stable mergesort for doubly-linked lists.

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

Get C5 from `http://www.itu.dk/research/c5/`.

C5 is described in a free book, published as ITU Technical Report ITU-TR-2006-76.

Learn much more about generics in Java and C#

- *C# Language Specification*, Ecma Standard ECMA-334, 3rd edition.
  At <http://www.ecma-international.org/publications/standards/Ecma-334.htm>
- D. Syme and A. Kennedy: Design and Implementation of Generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, 2001.
  At <http://research.microsoft.com/~dsyme/papers/generics.pdf>
- J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*, Addison-Wesley, 3rd edition, 2005.
  <http://java.sun.com/docs/books/jls/>.
- G. Bracha *et al.*: Making the future safe for the past: Adding Genericity to the Java Programming Language. OOPSLA 98, Vancouver. At <http://homepages.inf.ed.ac.uk/wadler/gj/Documents/gj-oopsla.pdf>.
  This design became Java 1.5 generics in 2004.
- R. Garcia *et al.*: A comparative study of language support for generic programming. OOPSLA'03, October 2003, 115-134. Compares generic types in C++, Standard ML, Haskell, Eiffel, Java, C#.
- A. Kennedy and C. Russo: Generalized Algebraic Data Types and Object-Oriented Programming. OOPSLA, October 2005, San Diego, California. At <http://research.microsoft.com/~akenn/generics/gadtoop.pdf>.
  Exploits generic types in object-oriented languages for generic programming (typesafe `printf` and such).

The exercises: What software to use

- **Java:**

  Java 2 SE 5.0 SDK is installed on DIKU bach-2.
  Or download it from ⟨http://java.sun.com⟩
  Class library documentation at ⟨http://java.sun.com/docs/⟩

- **C#**

  Mono 1.1.12 should be installed somewhere on DIKU (kand-0?).
  Or download it from ⟨http://www.mono-project.com⟩
  This is still beta software and has some known bugs.

  On Microsoft Windows you can use .Net 2.0 Framework SDK (final release, November 2005).
  Download it from ⟨http://msdn.microsoft.com/netframework/⟩

  Class library documentation at ⟨http://msdn.microsoft.com/library/⟩