Extending Programming Languages: the case of Java

Mirko Viroli
DEIS, Alma Mater Studiorum - Università di Bologna
mirko.viroli@unibo.it

The Research on Mainstream Languages in aliCE

Mirko Viroli
DEIS, Alma Mater Studiorum - Università di Bologna
mirko.viroli@unibo.it

Outline

1. On the Research Field of Mainstream Programming Langs
2. Generics in JDK 5.0
   - A brief tutorial
   - Relationships with the research in Cesena
   - Some pitfalls
3. Towards new extensions
   - Run-time generics, and the Sun-DEIS collaboration
4. Collaborations and thesis

On Research and Programming Langs.

- Several research topics are on “new” areas
  - agent-based systems, artificial intelligence, ...
  - typically, big changes with small industrial impact
  - medium-to-long term vision
- Some research topics are on mainstream areas
  - working on extending mainstream frameworks such as Java, .NET
  - typically, small changes with great impact
  - short term vision
- A research topic in aliCE is on extending Java
  - initially with my PhD, now with some collaborators

The Role of OOP Languages

- Programming languages are still the most widely used artifact for building software
- Among the others, object-orientation is the lead paradigm for large-scale systems
- OOP Languages: C++, Java (,C#)
- OOP not only impacts implementation, but the whole development cycle:
  - design, coding, testing

Java

Since 1995 Sun Microsystems’ Java surfaced as an alternative to the widely-used C++

- Great interest from industries
  - support for heterogeneity
  - fasten the development process
  - higher-quality documentation and libraries
- Adopted as reference for research
  - cleaner and more compact semantics
  - brings new and open issues to the mainstream
    - VM approach, garbage collection, reflection
- Cloned by competitors.. (Microsoft C#)

Hot Issues in the OOP field

OOPLs seem to be at a mature stage

- still, new scenarios and applications call for improvements with a potentially high impact

- Program Organization
  - Modularity, Aspect Oriented,...
- Flexibility, reuse and safety
  - Advanced Type Systems, Concurrency,...

Generics

- "abstracting constructs from types"
  - C++ templates, ADA Generics, ML polymorphism
  - lacked in early versions of either Java and C#
- Pros, general enhancement of
  - reuse, safety, maintainability, expressiveness
- Cons, need more experience
  - more complex language ⇒ need time to learn
  - current designs maybe need to become more mature
    - ➢ e.g. how to port existing approaches in Java?

Generics for Java

- The importance of generics was recognised!
- The GJ project (1996)
  - followed a CFP by Sun Microsystems
  - collaboration academy-industry
  - published/advertised in scientific forums
  - ➢ new interest on generics for the mainstream
- Some extension/variation developed meanwhile
  - e.g. "wildcards"
- It was finally released in JDK 5.0 (2004)
  - it is the most crucial Java extension so far

Collections in Java

- Java Collection Framework (JCF)
  - classes and interfaces to represent collections/containers
  - List, ArrayList, LinkedList, Hashtable, Vector
  - mostly in package java.util
- It is one of the most used libraries
  - to represent and manipulate data structures
- It is one of the most critic libraries
  - a problem in it can have a dramatic impact
  - big attention on possible changes
  - big feedbacks on the languages and tools

The genericity idiom (in Java 1.4)

```
class List{
  Object head;
  List tail;
  List(Object h,List t){
        head=h; tail=t;
  }
  Object getHead(){ return head; }
  List getTail(){ return tail; }
  void setHead(Object o){ head=o; }
  void setTail(List l){ tail=l; }
}
```

```
...
List l=new List("1",new List("2", null));
String s=(String)l.getHead(); // "1"
...
Integer i=(Integer)l.getHead();
// ClassCastException
```

- How to deal with the fact that the type of elements in the list is not known?
  - using the most general type, that is, Object

The safety problem

- Definition of "safety":
  - technically: correct programs never lead to an error
- A program is correct if it is well-typed
  - what is a type system? It is basically a checking algorithm...
  - ➢ it associates to each expression a type
  - ➢ it verifies whether operators are used correctly with respect to the types of arguments
    - es.: 2+true, new Object().prova() are not well-typed
  - ➢ it is implemented inside the compiler "javac"
- It is a crucial property!!

Safety of Java

- Which Java instructions can lead to a run-time error?
  - NullPointerException, Object o=null; o.toString()
  - ArrayIndexOutOfBoundsException, new int[]{}[-1]
  - ClassCastException, Object o="1"; Integer i=(Integer)o;
  - ArrayStoreException, Object[] o=new Integer[1]; o[0]="1";
  - ...?
- Each error could be intercepted with the proper additional code (through instanceof operator)
  - but programmers, obviously, never add that code
- These cases have to be avoided where possible
  - if compilation succeeds we must know the program is correct
  - in Java, at least, OO operations like method invocation and field access are safe

The case of the genericity idiom

```
class List{
  ...
  List(Object h,List t){...}
  Object getHead(){...}
  List getTail(){...}
  void setHead(Object o){...}
  void setTail(List l){...}
}
```

```
List l=new List(new Float(1),
          new List(new Float(2),
            null));
int n=sum(l);
// ClassCastException
```

```
// Please: this should be a List of java.lang.Integer!!!!
void int sum(List l){
        if (l==null) return 0;
        return ((Integer)l.getHead()).intValue()
                +sum(l.getTail()));
}
```


Typing is here not sufficient!!

- The programmer:
  - must rely on comments that describe what the List should contain
- That is, the type system is not sufficient
  - the compiler does not guarantee that any object passed to sum would be elaborated without errors
  - the problem is that we are trying to use informal comments in place of actual types!!
  - (it is reported that more than 50% of errors are of this kind!!)
- What do we need?
  - the possibility of denoting and managing types for the concepts of "List of Integers" and "List of Strings",..

Parametric construct

- This problem is classically solved with a polymorphism technique called parametric polymorphism
  - polymorphism: the same code usable in more contexts!!
  - inclusive polymorphism (subtyping): the code written for Object can be used for an Integer as well!
  - parametric polymorphism (genericity): the code for List can be used for any kind of list, lists of integer, lists of floats,...
- Parametric polymorphism:
  - generic modules in ADA, polymorphic functions in ML, templates in C++
  - idea: to make a construct (module, function, class, method, type) parametric with respect to one (or more) type(s).

The generic version of List

- Using generics as available in JDK 1.5

```
class List{
  Object head;
  List tail;
  List(Object h,List t){
        head=h; tail=t;
  }
  Object getHead(){ return head; }
  List getTail(){ return tail; }
  void setHead(Object o){ head=o; }
  void setTail(List l){ tail=l; }
}
```

```
class List<X>{
  X head;
  List<X> tail;
  List(X h,List<X> t){
    head=h;tail=t;
  }
  X getHead(){ return head; }
  List<X> getTail(){ return tail; }
  void setHead(X o){ head=o; }
  void setTail(List<X> l){ tail=l; }
}
```


Safe access

```
List l=new List("1",new List("2", null));
String s=(String)l.getHead(); // "1"
...
Integer i=(Integer) l.getHead();
// ClassCastException
```

```
List<String> l=new List<String>("1", new List<String>("2", null));
String s=l.getHead(); // "1"
...
Integer i=l.getHead(); // The type system here issues an error
```

- With this construct it is impossible to misinterpret the elements of a List
  - the resulting language is more solid, expressive, maintainable,..
  - it is however also more complicated!!!

Generic classes

- Declaring a generic class
  - it is a class that abstracts from the actual instantiation of one or more types, which are then treated as (type) parameters
  - if a class represents a collection, parameters represent the types of elements in the collection
- Using a generic class
  - for instance when creating an object: new List<String>(...)
  - the actual instantation of the parameter must be specified
  - then, List<String> is a type similar to standard types in Java

More type parameters

```
class Hashtable<K, V>{
  ...
  Hashtable(){...}
  void put(K k,V v){...}
  V get(K k){...}
}
...
Hashtable<Integer,String> h=new Hashtable<Integer,String>();
h.put(new Integer(1350),"one");
h.put(new Integer(1211),"two");
h.put(new Integer(76),"three");
String s =h.get(new Integer(1211));
```


Another example

```
class Pair<X,Y>{
  X fst;
  Y snd;
  Pair(X fst,Y snd) { this.fst=fst; this.snd=snd; }
  X getFst(){ return fst; }
  Y getSnd(){ return snd; }
}

Pair<String,String> p= new Pair<String,String>("1","2");
Pair<String,Integer> p2= new Pair<String,Integer>("1",new Integer("2"));
String s=p.getFst()+p.getSnd(); // "12"
```


Generic Methods

- For the same reasons why a class could be generic, it might be interesting to make a method generic!
- E.g.:
  - when the utility of a library is a static method

```
class Utility{
   <X> static void transfer(List<X> from,List<X> to){...}
}

List<String> l1=...;
List<String> l2=...;
Utility.<String>transfer(l1,l2);
```

transfer guarantees the correspondence of “from” and “to” lists

Generic Methods

- For the same reasons why a class could be generic, it might be interesting to make a method generic!
- E.g.:
  - when the utility of a library is a static method

```
class Pair<X,Y>
   ...
   <Z> Pair<X,Z> chgSnd (Z newsnd){
         return new Pair<X,Z>(getFst(),newsnd);
   }
}
Pair<String,String> p=...;
Pair<String,Integer> p2=p.<Integer>chgSnd(new Integer(1));
```


Inference in Method Calls

- In a generic method call, there could be cases where specifying the instantiation of the type parameter is useless
  - it can be inferred automatically by the compiler without ambiguity!
- In JDK 1.5:
  - Specifying method type parameters is optional: if omitted it is just inferred!

```
class Pair<X,Y>
     <Z> Pair<X,Z> chgSnd (Z newsnd){...}
}
Pair<String,String> p=...;
Pair<String,Integer> p2=p.<Integer>chgSnd(new Integer(1));
...
Pair<String,Float> p3=p.chgSnd(new Float(1.2));
// The compiler infers Float for Z
```

Complications

```
<X> List<X> nil(){ return new List<X>(null,null);}
<X> List<X> cons(X x,List<X> l){ return new List<X>(x,l); }
```

```
List<String> l=cons("1",new List<String>("2",null)));
// Inferring String for X!!
```

```
List<String> l=cons(new Integer(1),new List<String>("2",null)));
// ambiguous!
```

```
List<Object> l=nil(); // Without information, Object is inferred!
```

```
List<String> l=cons("1",null); // OK, inferring String
```


Why inference?

- It is generally considered as a nice way of reducing unnecessary syntax
- It is a mechanism invented in functional languages (ML)
  - it does not work so well in OO languages...
- History of Java's type inference algorithm:
  - first version: not sound!
  - second version: correct! but optional
  - third version: new difficulties (related to wildcards)!
  - .. what next? (It seems (to me) it complicates any new addition..)
- My opinion:
  - SAY NO TO INFERENCE IN METHOD CALLS!!!!

Bounded Polymorphism

- It is possible to constrain the polymorphism of a generic class or method
  - saying a type parameter must extend a class and/or implement some interfaces
  - if you do not specify any constraint, Object is assumed as upperbound!

```
interface Initializable{
        void init();
}
class Pair<X implements Initializable,Y implements Initializable>{
   ...
   void initBoth(){
        getFst().init();  // Invoking init to an object of type X
        getSnd().init(); // Invoking init to an object of type Y
   }
}
```


F-Bounded Polymorphism

- The bound of a type variable can be put in a recursive way
  - the resulting language is somehow complicated

```
interface Comparable<T>{
        boolean isGreaterThan(T t);
}
class MyElement implements Comparable<MyElement>{
        boolean isGreaterThan(MyElement e){...}
}
...
<T implements Comparable<T>> void sort(List<T> l){...}
```

The JDK 5.0

- The language described so far, that is, GJ proposal
- Plus the wildcard mechanisms developed in 2002/2003
  - originated from the following work at DEIS
  - Atsushi Igarashi, Mirko Viroli: On Variance-Based Subtyping for Parametric Types. ECOOP 2002 conference.
- Some background
  - invented with the name “variant parametric types”
  - first applied with some small variation in Tiger beta release (May 2003)
  - syntax and name changed, as “wildcards”..

Generics vs. subtyping

Two kinds of polymorphism

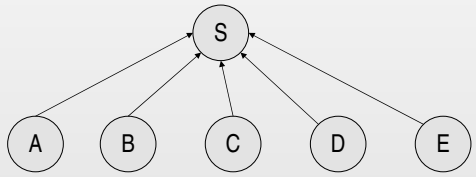
- inclusive (subtyping):
  - it creates a hierarchy of types
  - a subtype can be passed where a supertype is expected
  - e.g.: a functionality working on a Number can actually accept also an Integer
- parametric (generics):
  - a construct can be defined as parametric
  - the parameter is specified when the type is used
  - e.g.: List<X> instead of List, an example of use is List<String>

➢ each has its own benefits, flexibility, applications

- can they be integrated each other?

Factorisation and subtyping

- Factoring types in a hierarchy
  - if you have a number of types
  - if some of them have common properties (fields / methods)
  - you factorise these properties into a common supertype S
  - if in a context you need only those common properties, you can safely expect a type S
- In Java, this approach is realised with inheritance!



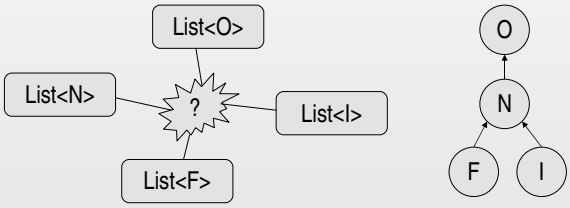
Generic types..

```
class List<X extends Object>{
  X head;
  List<X> tail;
  List(X h,List<X> t){
    head=h;tail=t;
  }
  X getHead(){ return head; }
  List<X> getTail(){ return tail; }
  void setHead(X o){ head=o; }
  void setTail(List<X> l){ tail=l; }
}
```

- From List<X> l can generate (use) types:
  - List<Object>, List<Number>,List<Integer>,List<Float>

Factoring generic types??

- Can we factorise generics?
  - considering types List<Object>, List<Number>, List<Integer>, List<Float>
  - do they have some common supertype (other than Object)?
  - does it make sense to write functions accepting **any** list?
  - for instance: is List<Object> more general than the others??



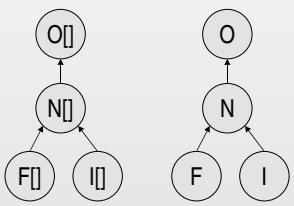
Factoring properties

- What do these types have in common:
  - List<Object>,List<Integer>,List<Number>,List<Float>
  - if I get their head I obtain a Object,Integer,Number, or Float
  - but they are all of type Object
- Hence:
  - a type that factors them can define a method: Object getHead()
- Viceversa:
  - the method setHead(Object o) cannot be factorized
  - because e.g. I can't put an Object into a List<Float>
- If a type is more general, it provides less operations!!
- Note that a wrong decision here might lead to an unsafe language!

Arrays in Java

- One can note that (already in JDK 1.4) Java arrays are sorts of generic types:
  - Object[], Number[], Integer[], Float[]
  - are similar to (Array<Object>,Array<Number>,Array<Integer>,Array<Float>)
- Which factoring do they admit?
  - in Java the so-called covariance is used
  - if X <: Y, then X[] <: Y[], that is:

```
Object[] o=new String[]{"1","2","3"};
// This compiles OK!!
```



Unsafety of arrays

- But, is it right to factorise in O[] the operation of writing a new element in N[]?
  - answer: NO!, in fact:

```
String[] s=new String[]{"1","2"};
Object[] o=s;  // OK for array covariance
o[0]=new Integer(1);
// Statically correct, but raises an ArrayStoreException
```

- This issue is not just a thing for theoreticians!
  - each array store operation can possibly fail!!!
  - the JVM must check that writing is correct, and this results in a serious performance overhead!
  - factorisation of generics must be designed with great care!!

Two approaches

- Declaration-site variance
  - trying to enforce a direct subtyping between different generic classes (for instance List<Integer> <: List<Object> since Integer <: Object)
  - It is the classical approach, but never been really used
  - Like for arrays, it leads to run-time errors
- Use-site variance
  - introducing NEW types that factorise many generic types, and define only the operations that can be safely used
  - invented in our ECOOP 2002 paper
  - implemented by Sun Microsystems in Java Wildcards
  - such new types called “wildcard types”

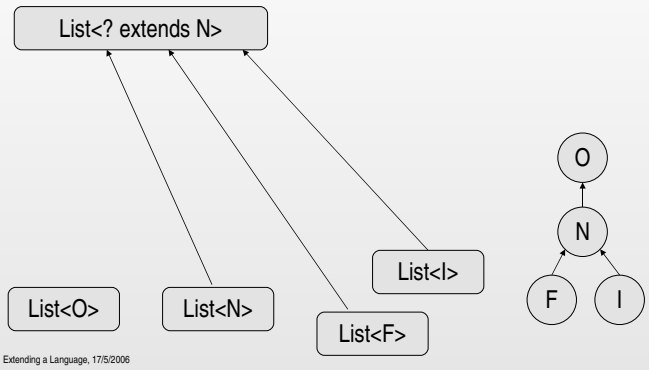
Syntax of some of these new types

After a class List<X> is defined, you can use the following types:

- standard generic types, List<T>
  - List<Object>, List<Integer>,...
  - these are used to create objects, as usual
- covariant types, List<? extends T>
  - List<? extends Number> is a supertype of all List<S> where S extends Number
  - that is, where S <: Number
- where are covariant types useful?
  - where a List<? extends Number> is expected, you can pass a List<Integer>, List<Float>, and so on.

Hierarchy

List<? extends Number> is like an interface
The type of all lists of all numbers



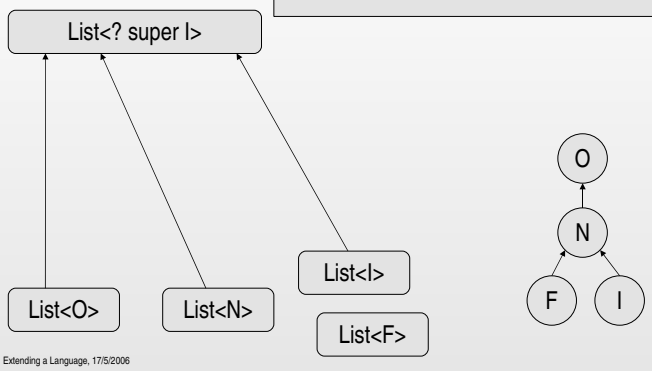
More new types

After a class List<X> is defined, you can use the following types:

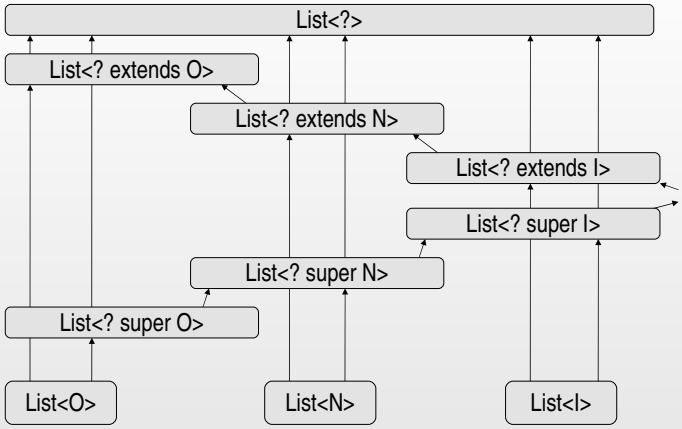
- covariant types, List<? extends T>
  - List<? extends Number> is a supertype of all List<S> where S extends Number
  - that is, where S <: Number
- contravariant types, List<? super T>
  - List<? super Number> is a supertype of all List<S> where Number extends from S
  - that is, where Number <: S
- bivariant types, List<?>
  - List<?> is a supertype of any List<S>

Hierarchy

List<? super Integer> is like an interface
The type of all lists of something more general than integers



Subtyping variance-based



The interval metaphor

- Each wildcard type
  - C<T>, C<? extends T>, C<? super T>, C<?> induces a sort of interval
  - this interval defines all the types S such that C<S> is a subtype..
- List<Number>: [Number,Number]
- List<? extends Number>: [NullType,Number]
- List<? super Number>: [Number,Object]
- List<?>: [NullType,Object]
- A wildcard W is a subtype of another V if the interval of W includes the interval of V

An interpretation: Production/Consumption

Which operations are allowed by these types?

- List<T>
  - all operations defined in class List can be invoked
- List<? extends T> (factors List<R>, R<:T)
  - only getHead and getTail (setTail and setHead can be shown unsafe)
  - it represents the lists which can just produce elements of type T
- List<? super T> (factors List<R>, R:>T)
  - only setHead and setTail
  - it represents the lists which can just consume elements of type T
- List<?> (factors List<R> for each R)
  - no methods! You could invoke a method "int size(){..}"
  - represents the lists that can produce and consume no elements of type T

Example 1

```
class List<X>{
  ...
  X getHead(){ ... }
  List<X> getTail(){ ... }
  void setHead(X o){ ... }
  void setTail(List<X> l){ ... }
  void importHead(List<X> l){
        setHead(l.getHead());
} }
```

```
List<String> ls=
  new List<String>("1",
   new List<String>("2",null));
List<String> ls2=...; //["0","1"]
ls.importHead(ls2);
/* ls from ["1","2"]  to ["0","2"] */
```

- In importHead
  - argument l is used only to invoke getHead
  - do I really need a List<X>, or something more general could be useful?
  - idea: using List<? extends X>

Example 1 with wildcards

```
class List<X>{
  ...
  X getHead(){ ... }
  List<X> getTail(){ ... }
  void setHead(X o){ ... }
  void setTail(List<X> l){ ... }
  void importHead(List<? extends X> l){
        setHead(l.getHead());
} }
```

```
List<Number> ln=...;
List<Integer> li=...;
List<Object> lo=...;
ln.importHead(li); // OK!!
ln.importHead(lo);// NO!!!
```

It is sufficient that the argument is a list of elements which are more specific than X

Example 1 with generic methods

```
class List<X>{
  ...
  X getHead(){ ... }
  List<X> getTail(){ ... }
  void setHead(X o){ ... }
  void setTail(List<X> l){ ... }
  <Y extends X> void importHead(List<Y> l){
        setHead(l.getHead());
} }
```

```
List<Number> ln=...;
List<Integer> li=...;
List<Object> lo=...;
ln.<Integer>importHead(li); // OK!!
ln.<Object>importHead(lo);// NO!!!
```

Must specify Y

Which is better?

```
class List<X>{
 <Y extends X> void importHead(List<Y> l){
        setHead(l.getHead());
 }
 ...
 void importHead(List<? extends X> l){
        setHead(l.getHead());
 }
}
```

- Almost identical..
  - The second is maybe better because it involves no other explicit type Y..

Example 2, contravariance

```
class List<X>{
  ...
  X getHead(){ ... }
  List<X> getTail(){ ... }
  void setHead(X o){ ... }
  void setTail(List<X> l){ ... }
  void exportHead(List<X> l){
        l.setHead(getHead());
} }
```

```
List<String> ls=
  new List<String>("1",
   new List<String>("2",null));
List<String> ls2=...; //["0","1"]
ls.exportHead(ls2);
/* ls2 from ["0","1"] to ["1","1"]*/
```

- In exportHead
  - the argument l is used only to invoke setHead
  - do I really need a List<X>, or something more general could be useful?
  - idea: using List<? super X>

Example 2 with wildcards

```
class List<X>{
  ...
  X getHead(){ ... }
  List<X> getTail(){ ... }
  void setHead(X o){ ... }
  void setTail(List<X> l){ ... }
  void importHead(List<? super X> l){
        l.setHead(getHead());
} }
```

```
List<Number> ln=...;
List<Integer> li=...;
List<Object> lo=...;
ln.exportHead(li); // NO!!
ln.exportHead(lo);// OK!!!
```

It is sufficient that the argument is a list of elements which are more general than X

Example 2 with generic methods

```
class List<X>{
  ...
  X getHead(){ ... }
  List<X> getTail(){ ... }
  void setHead(X o){ ... }
  void setTail(List<X> l){ ... }
  <Y super X> void importHead(List<Y> l){
        setHead(l.getHead());
} }
```

Differently from wildcards, bounds of a method or class parameter cannot be of the super kind!!!
THE ABOVE EXAMPLE DOES NOT COMPILE!!!

Example 3

```
class List<X>{
  ...
  boolean sameSize(List<X> l){ ... }
} }
```

```
class List<X>{
  ...
  boolean sameSize(List<?> l){ ... }
} }
```

```
class List<X>{
  ...
  <Y> boolean sameSize(List<Y> l){ ... }
} }
```


Example 4

```
class List<X>{
  ...
  void importFirst(List<Pair<X,X>> l){
        setHead(l.getHead().getFst());
} }
```

```
  ...
  void importFirst(List<? extends Pair<? extends X,?>> l){
        setHead(l.getHead().getFst());
  }
```

```
  ...
  <Y extends X,Z,W extends Pair<Y,Z>>
     void importFirst(List<W> l){
        setHead(l.getHead().getFst());
  }
```


Examples from Java Collections Framework

Class Collections provides utilities for handling collections, as static methods

```
interface Comparable<T>{ boolean isGreaterThan(T t); }
interface Comparator<T>{ int compare(T t1,T t2); ...}
...
<T>void fill(List<? super T> list, T obj)
<T>void copy(List<? super T> dest, List<? extends T> src)

<T extends Comparable<? super T>> void sort(List<T> list)
<T>void sort(List<T> list, Comparator<? super T> c)
```

Is there more on that?

- Yes. Very strange and not fully documented things can happen, that require the programmer to be particularly skilled.

```
class C<X>{
    C<? super X> checkAndReturn(C<? super X> l){
        return l;
    }
    public static void main(String[] s){
        C<? extends Number> l=null;
        C<? super Number> l2=null;
        l2=l.checkAndReturn(l2); // Is this call correct???
    }
}
```


The compiler reports this error!!!

```
class C<X>{
    C<? super X> checkAndReturn(C<? super X> l){
        return l;
    }
    public static void main(String[] s){
        C<? extends Number> l=null;
        C<? super Number> l2=null;
        l2=l.checkAndReturn(l2); // Is this call correct???
}   }
```

```
C.java:10: incompatible types
found   : C<capture of ? super capture of ? extends java.lang.Number>
required: C<? super java.lang.Number>
                        l2=l.checkAndReturn(l2);
                                            ^
1 error
```


Why?

- The receiver of the invocation is to be captured:
  - C<? extends Number> --> C<Z> where Z in [Number,Object]
  - what the compiler calls C<capture of ? extends Number>
- The return type is computed based on this type and is captured
  - C<? super X>, where X is “Z in [Number,Object]”
  - that is, C<? super Z> where Z in [Number,Object]
  - by capturing: C<W> where W in [NullType,Z] and Z in [Number,Object]
- Is the assignment correct?
  - “C<W>, W in [NullType,Z], Z in [Number,Object]” <: C<? super Number>
  - By looking at intervals, it is not sure if W is greater than Number!!
  - Hence the assignment is not correct!

Java subtyping is undecidable!!!

```
class D<X extends C<C<? super X>>> {
  D(X x) {
    C<? super X> f = x;
  }
}
```

```
The system is out of resources.
Consult the following stack trace for details.
java.lang.StackOverflowError
    at com.sun.tools.javac.code.Types$ContainsTypeFcn.containsType(..
    at com.sun.tools.javac.code.Types.containsType(Types.java:841)
    at com.sun.tools.javac.code.Types.containsType(Types.java:811)
    at com.sun.tools.javac.code.Types$IsSubTypeFcn.visitClassType(..
```

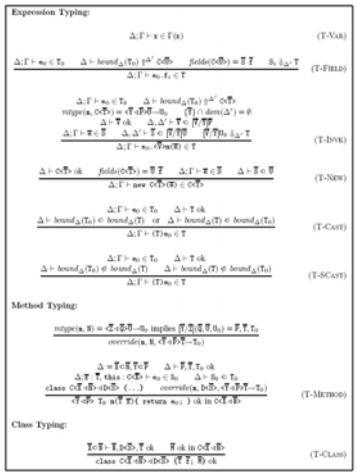

About extending languages...

- Java was conceived as a very simple language
  - no multiple inheritance
  - no nested classes
  - no generics
  - ...
- Now it is a fairly more complicated language
  - there is some code whose typing is very obscure
- However, this is not likely affecting typical Java users, but rather designers of generic libraries...

What is needed to extend a language

- If we are talking about a mainstream language, the chance of winning is VERY low
- You need:
  - a good idea, simple yet powerful
  - a formal model stating it is correct (that is, safe)
  - a good implementation support
  - to be lucky

The Type System



Outline

1. On the Research Field of Mainstream Programming Langs
2. Generics in JDK 5.0
   - A brief tutorial
   - Relationships with the research in Cesena
   - Some pitfalls
3. Towards new extensions
   - Run-time generics, and the Sun-DEIS collaboration
   - Family Polimorphism
4. Collaborations and thesis

Implementation of JDK 5.0

- How to implement generics?
  - do we need a completely new compiler and JVM?
- Sun Microsystems called for proposals
  - Java Specification Request JSR-0000014
  - some requirement on performance: overhead < 10 %
  - some requirement on compatibility
- Various Proposals
  - GJ, NextGen, PolyJ, EGO, Reflective, LoadTime
- Who won?
  - for JDK 5.0, GJ is the solution adopted
  - by Bracha, Odersky, Stoutamire, Wadler

Type erasure

- Idea, translating generic code into the corresponding non-generic one, at compile-time

```
class List{
  Object head;
  List tail;
  List(Object h,List t){
    this.head=h;this.tail=t;
  }
}
...
List l=new List("1",null);
String s=(String)l.getHead();
...
Integer i=(Integer)l.getHead();
// Run-time exception
```

```
class List<X extends Object>{
  X head;
  List<X> tail;
  List(X h,List<X> t){
    this.head=h;this.tail=t;
  }
}
...
List<String> l=
   new List<String>("1",null);
String s=l.getHead();
...
Integer i=(Integer)l.getHead();
// Intercepted at compile-time
```

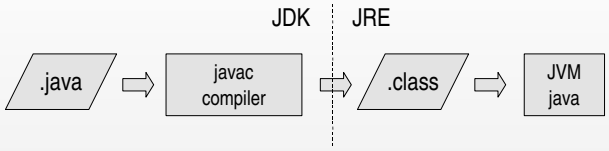

The legacy problem

GJ is a good solution because

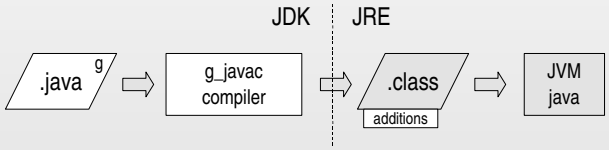
- Is upward compatibile
  - old code can be read by the new compiler
- Is downward compatibile
  - you do not need to change your JVM
  - without generics → you create the same .class
  - with generics → you create .class files readable from legacy JVMs
- Performance
  - almost 0% in space and time!

Implementation schema

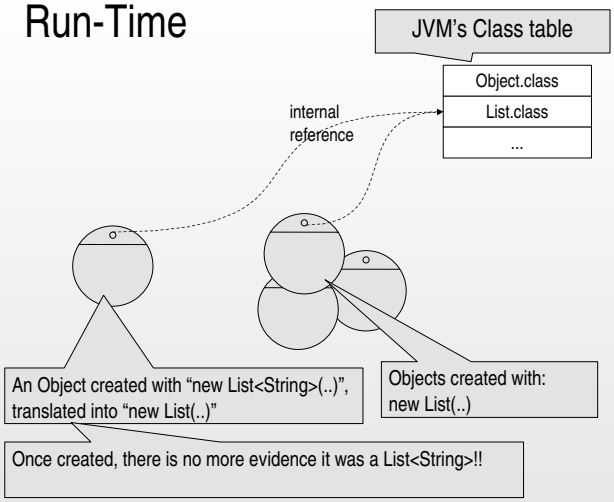
- Java standard



- GJ extension with generics



Run-Time



The run-time problem

```
List<String> l=new List<String>(...);
Object o=l;
...
List<String> l2=(List<String>)o; // OK!
List<Integer> l3=(List<Integer>)o; // Run-time error
```

```
List l=new List(...);
Object o=l;
...
List l2=(List)o; // OK!
List l3=(List)o; // OK!!!!!, but this is actually wrong
```

- GJ and JDK 5.0 implementation does not support run-time generics!
  - it does not integrate well with Reflection, Persistence, operators such as casts and instanceof
  - Operations like those above issue some warnings!!

Which impact?

- This is a big compromise
  - Sun Microsystems basically released an incomplete version of generics!!
  - The reason is that no adequate solution to the problem existed at that time!
- Which effect on programmers?
  - not easy to predict...
  - potentially relevant, maybe limited
- Note that C# already has runtime generics!!

Java extensions with runtime generics

- Extending both the compiler and the JVM
  - better performance, no legacy support
    - PolyJ (+ some work here in Cesena)
- Estending the compiler + a new class loader
  - good performance, partial legacy support
    - load-time approaches
- Estending only the compiler
  - worse performance, legacy support
    - NextGen by Sun Microsystems & Rice Univ.
    - EGO by Mirko Viroli + Maurizio Cimadamore (DEIS + Sun Microsystems)

EGO (Exact Generics On-demand)

- Conceived by Mirko Viroli
  - during the PhD
  - Completed in collaboration with Maurizio Cimadamore at Sun Labs, Palo Alto, and now developing support to wildcards
  - Project in collaboration with Sun Microsystems
- Idea
  - adding an expansion phase to GJ compiler, where the necessary code is added so that information on type parameters is passed to objects and properly stored for fast retrieval (type-passing style)
  - where is the news? we do that efficiently!!

Type-Passing (GJ → GJ)

```
class List<X>{
  X head;
  List<X> tail;
  List(X h,List<X> t){
    this.head=h;this.tail=t;
  }
}
...
Object o=new List<String>("1",null);
...
if (o instanceof List<Integer>) ...
// Not supported in GJ
```

```
class List<X>{
  $D $d;
  X head;
  List<X> tail;
  List($D $d,X h,List<X> t){
    this.$d=$d;
    this.head=h;this.tail=t;
  }
}
```

descriptor for List<String>

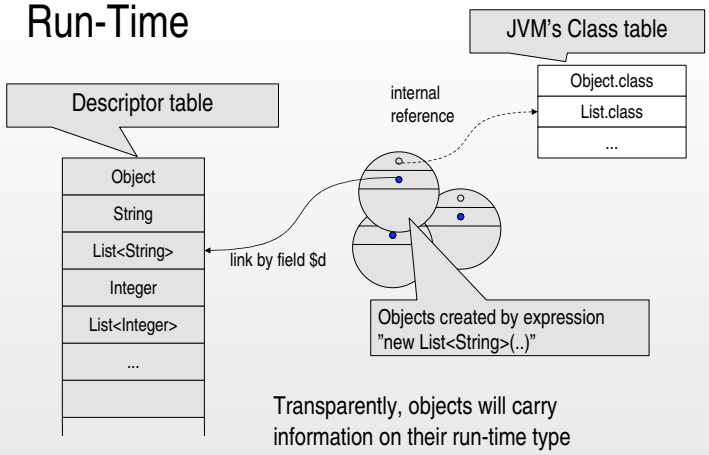
```
Object o=new List<String>($dls,"1",null);
...
if (o.$d.isSubType($dli)) ...
```

Access On-demand

```
class Client{
  void m(){
    Object o=new List<String>($get(0),“1”,null);
    Object o2=new List<Integer>($get(1),new Integer(2),null);
  }
  private static $D[] $ds=new $D[2];
  private static $D get(int i){
    if ($ds[i]!=null) return $ds[i];
    switch (i){
      case 0: return $ds[i]=$Dtab.register(/*List<String>*/);
      case 1: return $ds[i]=$Dtab.register(/*List<Integer>*/);
    }
  }
}
```

The descriptor is searched in a global table, and is created only the very first time!

Run-Time



Transparently, objects will carry information on their run-time type

Will it be applied?

- EGO:
  - it fully supports generics at run-time
  - it gives similar legacy support properties than GJ
  - acceptable perfomance (<5% in time)
- A new strategy for Sun?
  - in a future release (JDK 1.6?) there will be a run-time support of generics directly inside the JVM
  - which approach?
- New DEIS-Sun collaboration
  - The “EGO inside the JVM” project

Available theses

- On wildcards
  - basically concerning providing proper tool support for them!
  - writing some Eclipse plugin that helps in dealing with the complications due to wildcards
- On run-time generics
  - starting from our prototype JVM
  - adding support to Reflection (library + core support in the JVM)
  - adding support to Persistence (library + core support in the JVM)
  - measuring performance and finding optimisations