Adding Wildcards to the Java Programming Language

Mads Torgersen, University of Aarhus, Denmark
Erik Ernst, University of Aarhus, Denmark
Christian Plesner Hansen, OOVM, Aarhus, Denmark
Gilad Bracha, Sun Microsystems, Inc., California, USA
Peter von der Ahé, Sun Microsystems, Inc., California, USA
Neal Gafter, Google Inc., California, USA


Presented by Boris Kirzner
Old Java containers

- List can contain elements of any type
- Unable to ensure run-time type safety
- Requires explicit casting

```java
public List choose (List fst, List snd) {
    return (today == "Sunday") ? fst : snd;
}
```

```java
List circles = new ArrayList();
List rectangles = new ArrayList();
...
Circle circ = (Circle) circles.get(0);
circles.add(new Rectangle());
Rectangle rect = (Rectangle) circles.get(0);
List l = choose(circles, rectangles);
Shape s = (Shape) l.get(0);
```
Old Java containers

• List can contain elements of any type
• Unable to ensure run-time type safety
• Requires explicit casting

```java
public List choose (List fst, List snd) {
    return (today == "Sunday") ? fst : snd;
}
```

List circles = new ArrayList();
List rectangles = new ArrayList();

... Circle circ = (Circle) circles.get(0);
circles.add(new Rectangle());
Rectangle rect = (Rectangle) circles.get(0);
List l = choose(circles, rectangles);
Shape s = (Shape) l.get(0);

Bug
Run-time exception
What is the type of list returned?

Object → Shape → Circle → Rectangle → Square

● List can contain elements of any type
● Unable to ensure run-time type safety
● Requires explicit casting
Java generics type safety

• In GJ type “list of circles” is different from type “list of rectangles”.
• Compile-time check

```java
public interface List<T> {
    void add(T t);
    T get(int i);
}
```

```java
List<Circle> circles = new ArrayList<Circle>();
List<Rectangle> rectangles = new ArrayList<Rectangle>();
...
circles.add(new Rectangle());
Rectangle rect = circles.get(0);
```
Java generics type safety

• In GJ type “list of circles” is different from type “list of rectangles”.
• Compile-time check

```java
class Circle {
    // Circle class implementation
}
class Rectangle {
    // Rectangle class implementation
}

public interface List<T> {
    void add(T t);
    T get(int i);
}

List<Circle> circles = new ArrayList<Circle>();
List<Rectangle> rectangles = new ArrayList<Rectangle>();
...
circles.add(new Rectangle()); // Fails at compile-time
Rectangle rect = circles.get(0); // Fails at compile-time
```
Peek into Java generics implementation

- **Erasure**: all the type information between '<>' is thrown out, i.e. List<String> becomes List
- Remaining uses of type variable replaced with upper bound (usually Object)
- Cast to specific types inserted whenever needed

```java
List<String> ls = new ArrayList<String>();
ls.add("Boris");
String s = ls.get(0);
```
Peek into Java generics implementation

- **Erasure**: all the type information between '<>' is thrown out, i.e. `List<String>` becomes `List`
- Remaining uses of type variable replaced with upper bound (usually `Object`)
- Cast to specific types inserted whenever needed

```java
List<String> ls = new ArrayList<String>();
ls.add("Boris");
String s = ls.get(0);
```

- Generic code is shared by all of its instances

```java
List ls = new ArrayList();
ls.add("Boris");
String s = (String)ls.get(0);
```
GJ object-oriented abstraction weakness

• Polymorphic abstraction: common supertype for List<T> and Set<T> is Collection<T>
• Object-oriented abstraction: no common supertype for List<Circle> and List<Rectangle>
• List<Object> can not be used as a supertype because this leads to run-time type exceptions

```java
List<Circle> circles = new ArrayList<Circle>();
List<Object> col = circles;
col.add(new Rectangle());
Circle circ = circles.get(0);
```
GJ object-oriented abstraction weakness

- Polymorphic abstraction: common supertype for List<T> and Set<T> is Collection<T>
- Object-oriented abstraction: no common supertype for List<Circle> and List<Rectangle>
- List<Object> cannot be used as a supertype because this leads to run-time type exceptions

```java
List<Circle> circles = new ArrayList<Circle>();
List<Object> col = circles;
col.add(new Rectangle());
Circle circ = circles.get(0);  
```
Object-oriented abstraction

- Should be some kind of list

Type-safety

- Prevent insertion of wrong elements
- Compile-time check
List of something?

```
List<?>

List<Circle>  List<Rectangle>
```
Wildcards

- Use '?' as a placeholder for a type name
- List<?> is a superclass for all List<T> classes
- Compile-time check

```java
public List<?> choose (List<?> fst, List<?> snd) {...}

List<Circle> circles = new ArrayList<Circle>();
List<Rectangle> rectangles = new ArrayList<Rectangle>();
...
List<?> l = choose(circles, rectangles);
```

- List l is guaranteed to contain elements of the same type, also we do not know the exact type
Simple wildcards limitations

• Exact parameter type of List<??> is unknown:
  • Nothing can be inserted except null, which is a member of every type
  • All the elements extracted are of type Object

• Let's introduce some restrictions (bounds) on parameter type in order to enable:
  • Adding new elements to collection
  • Reading elements of more specific type
**Extend-bounded wildcards**

- `List<? extends T>` is a list of elements of the type that derived from `T`.
- So-called “upper bound”: can contain elements of types that are down to `T` in the class hierarchy.
- Elements of type `T` can be extracted from the list.
- Nothing can be inserted except null.

```java
public List<? extends Shape> choose (...) {...}
List<? extends Shape> l = choose(circles, rectangles);
Shape s = l.get(0);
```
Why extend-bounds do not allow insertions

List<? extends Shape> may actually be one of

- List<Shape>
- List<Rectangle>
- List<Square>
- List<Circle>

```java
List<? extends Shape> l = choose(circles, rectangles);
l.add(...);
```

Fails at compile-time
Super-bounded wildcards (lower bounds)

- `List<? super T>` is a list of elements of the type that is supertype of T
- So-called “lower bound”: can contain elements of types that are higher than T in the class hierarchy
- Elements of type T can be inserted into the list
- All the elements extracted are of type `Object`

```java
List<? super Rectangle> superRectangles = ...
superRectangles.add(new Rectangle());
Object o = superRectangles.get(0);
```
Why super-bounds allow 'get' of Object only

List<? super Rectangle> may actually be one of

- List<Rectangle>
- List<Shape>
- List<Object>

```
List<? super Rectangle> superRectangles = ...
Object o = superRectangles.get(0);
Rectangle r = superRectangles.get(0);
```

Fails at compile-time
Type inference with wildcards

- Calling polymorphic methods without explicit type parameters
- Type is inferred from the type info on the call side
- Must satisfy a bound for T
- Prefer subtype over supertype: preserves more information about arguments and return values

```java
public <T> T choose (T fst, T snd) {...}

List<Circle> circles = new ArrayList<Circle>();
Set<Rectangle> rectangles = new TreeSet<Rectangle>();
...
Object o = choose(circles, rectangles);
Collection<?> l = choose(circles, rectangles);
```
Type inference with wildcards

- Calling polymorphic methods without explicit type parameters
- Type is inferred from the type info on the call side
- Must satisfy a bound for T
- Prefer subtype over supertype: preserves more information about arguments and return values

```java
public <T> T choose (T fst, T snd) {...}
```

```java
List<Circle> circles = new ArrayList<Circle>();
Set<Rectangle> rectangles = new TreeSet<Rectangle>();
...
Object o = choose(circles, rectangles);
Collection<?> l = choose(circles, rectangles);
```
Wildcards interface subtyping

```
Collection<?>

List<?>
Collection<? extends Shape>
List<? extends Shape>
List<Circle>

Set<?>
Collection<? extends Shape>
Set<? extends Shape>
Set<Rectangle>
```
Example: type inference

```java
public <T> T choose (T fst, T snd) {...}

Set<? extends Shape> s = ...
List<Square> l = ...
```

- What is the most specific type that can be inferred in call to `choose()`?
- Remember that List and Set share common superinterface: Collection
Example: type inference

```java
public <T> T choose (T fst, T snd) {...}
```

```java
Set<? extends Shape> s = ...;
List<Square> l = ...;
```

- What is the most specific type that can be inferred in call to `choose()`?
- Remember that List and Set share common superinterface: `Collection`
Example: type inference

```java
public <T> T choose (T fst, T snd) {...}
```

```java
Set<? extends Shape> s = ... List<Square> l = ...
```

- What is the most specific type that can be inferred in call to `choose()`?
- Remember that List and Set share common superinterface: Collection

```java
Collection<? extends Shape> c = choose(s, l);
```
Example: more complex type inference

```java
public <T> T choose (T fst, T snd) {...}
Set<? extends Square> s = ...
List<? super Shape> l = ...
```

- What is the most specific type that can be inferred in call to choose()?
Example: more complex type inference

```java
public <T> T choose (T fst, T snd) {...}
Set<? extends Square> s = ...;
List<? super Shape> l = ...;

Object o = choose(s, l);
```

- What is the most specific type that can be inferred in call to `choose()`?
Dilemma:

```java
public <T> List<T> unmodifiableList(List<T> list)
```

can not be called with `List<?>` since the actual element type is unknown

```java
public List<?> unmodifiableList(List<?> list)
```

called with `List<String>` will discard some information about return type

Solution:

allow first case: even we do not know the actual type, on each call it will be `some` specific type and `any` such type would make the invocation type-safe

Actual run-time type behind `T` is “captured” in the method invocation
Proper abstraction with wildcard capture

Method that reverses list elements order

- Implementator perspective: needs a name for element type, to remove and re-insert the elements
- Caller perspective: should only know the method receives List as an argument
- Wildcard capture allows us to mediate between these two needs:
Proper abstraction with wildcard capture

Method that reverses list elements order

- Implementator perspective: needs a name for element type, to remove and re-insert the elements
- Caller perspective: should only know the method receives List as an argument
- Wildcard capture allows us to mediate between these two needs:

```java
public static void reverse(List<?> list) {
    rev(list);
}

private static <T> void rev(List<T> list) {...}
```
Java type system with wildcards

- Like generic classes, wildcards can be nested:
  List<List<String>> is type “list of lists of strings”,
  and List<List<?>> is type “list of lists”.

- Wildcard is not a name of a type: in Pair<?,?> each occurrence of '?' are not assumed to be the same type: Pair<String,Integer> vs. Pair<String, String>
Virtual types

• BETA language *pattern* – abstraction mechanism that generalizes class, procedure, function etc.
• Virtual pattern – pattern that can be redefined in subpatterns (similar to “virtual” methods)
• Virtual class in BETA is a private case of virtual pattern

• Virtual type – introduced as proposal to add BETA virtual classes to Java (K. Thorup, 1997)
Bounding of type attribute

Generic List class

T is a type attribute, which may be further specified in subclasses:

Subclass with **further bound**

```java
abstract class NumberList {
    abstract typedef T as Number;
}
```

Subclass with **final bound**

```java
abstract class IntegerList {
    final typedef T as Integer;
}
```
Type hierarchy with virtual types

Similar to Java wildcards, but not the same

- “Pure” BETA only permits single inheritance, thus `List<Integer>` cannot be a subtype of both `Collection<Integer>` and `List<<?>>`
- Virtual types extension with structural subtyping (Thorup and Torgersen, 1999) solves this problem
Use-site variance

- Thorup and Torgersen
  
  *use-site covariance:*
  
  \[ \text{List}^{+\text{Number}} \], equivalent to \[ \text{List}^{\text{? extends Number}} \]
  (writing is prohibited)

- Igarashi and Viroli
  
  *contravariant form*
  
  \[ \text{List}^{-\text{Number}} \], equivalent to \[ \text{List}^{\text{? super Number}} \]
  (reading prohibited)

  *bivariant form*
  
  \[ \text{List}^{*\text{Number}} \] or just \[ \text{List}^{*} \], equivalent to \[ \text{List}^{??} \]
  (disallows both reading and writing)
Wildcard conclusions

- Origins from BETA virtual types
- Bridges the gap between object-oriented and polymorphic abstraction
- Simpler and more precise signatures
- Better type inference
- Part of Java5
Questions