The Role of Wildcard Types in Java Generics

Matthew Justin

Computer Science Discipline
University of Minnesota, Morris

Senior Seminar Conference 2006

Outline

Motivation for Generics

- A list is a commonly used data type in programming
- A given list can represent:
  - List of integers
  - List of strings
  - Etc.
- Underlying logic remains the same, regardless of element type
- Would like to use the same class for all types of lists
  - Prevents code duplication

Definition of Generics

- *Generic types* (*Generics*) allow abstraction over element type
- The element type is known as a *type parameter*
- In the list example:
  - *List* is a generic class
  - The type of the list (string, integer, etc.) is the type parameter

Generics in Java

- Added to Java in version 5.0
- Previously featured in other languages
  - In C++, known as *templates*
  - Very commonly used in C++
- Java-specific constraints complicated addition of generics
  - Static typing enforced by compiler
  - Rich type hierarchy
- Long academic debate on how best to include generics
- A number of proposals were created
- *Wildcards* were developed to meet these needs

Type Safety

- All operations on a value are defined at compile time
- Program won't compile if there are type errors such as:
  - Assigning value to a variable that doesn't match its type
  - Calling a method with a value of a wrong type
- This makes it easier to discover type errors
  - Easier to discover programming errors at compile time than at run time

Type Hierarchy

- If B is a *subtype* of A, then B can be substituted wherever a variable of type A is required
  - Can assign a B to a variable of type A
  - Can pass in a B as a parameter to a method requiring an A
- Integer is a subtype of Number

```
Number n = new Integer(5);
```

- In Java:
  - Subtyping is implemented through inheritance
  - Object is the root of the type hierarchy

Typecasting

- Given a variable of a type, can *typecast* to a subtype
- If not a variable of that type, will fail at runtime

```
Object o = new Integer(5);
Integer i = (Integer) o;
String s = (String) o; //Fails at runtime
```

The "generic idiom"

- A programming style that simulates generic functionality
- Replace type of variables by common supertype
- Use typecasts when accessing variables

```
interface List {
    boolean add(Object o);
    Object get(int index);
}
```

The "generic idiom"

- A programming style that simulates generic functionality
- Replace type of variables by common supertype
- Use typecasts when accessing variables

```
void toLowerCase(List list) {
    for (int i = 0; i < list.size(); i++) {
        Object o = list.get(i);
        String s = (String) o;
        s = s.toLowerCase();
        list.set(i, s);
    }
}

List stringList = ... //A list containing Strings
toLowerCase(stringList);
```

The "generic idiom"

- A programming style that simulates generic functionality
- Replace type of variables by common supertype
- Use typecasts when accessing variables

```
void toLowerCase(List list) {
    for (int i = 0; i < list.size(); i++) {
        Object o = list.get(i);
        String s = (String) o;
        s = s.toLowerCase();
        list.set(i, s);
    }
}

List stringList = ... //A list containing Strings
toLowerCase(stringList);
List integerList = ... //A list containing Integers
toLowerCase(integerList); //Fails at runtime
```

Problems with the "generic idiom"

- No type safety guarantees
  - Cannot enforce the type of elements
  - Fails at runtime
- More difficult to read
  - Typecasting clutters code
  - Extralinguistic method needed to denote element type
- Generics solve these problems

With generics

- Element type in code
- Returns actual element type
  - No typecasting required

```
void toLowerCase(List<String> list) {
    for (int i = 0; i < list.size(); i++) {
        String s = list.get(i);
        s = s.toLowerCase();
        list.set(i, s);
    }
}

List<String> stringList = ...
toLowerCase(stringList);
List<Integer> integerList = ...
toLowerCase(integerList); //Will not compile
```

Making a class generic

- Type parameter surrounded by angle brackets
  - May be of any object type

```
interface List<E> {
    boolean add(E o);
    E get(int index);
}
```

Making a class generic

- Type parameter surrounded by angle brackets
  - May be of any object type

```
interface List<E> {
    boolean add(E o);
    E get(int index);
}

interface List<Integer> {
    boolean add(Integer o);
    Integer get(int index);
}
```

Generics and subtyping

- String is a subtype of Object

Generics and subtyping

- String is a subtype of Object
- However, List<String> is *not* a subtype of List<Object>

Generics and subtyping

- String is a subtype of Object
- However, List<String> is *not* a subtype of List<Object>
- If it were, the following would be possible:

```
ArrayList<String> ls = new ArrayList<String>();
ArrayList<Object> lo = ls;
Object o = new Integer(5);
lo.add(o);
String s = ls.get(0);  //Error -- Illegal assignment
```

Motivating Example

```
public void printList(List<Object> list) {
    for (int i = 0; i < list.size(); i++) {
        Object next = list.get(i);
        System.out.println(next);
    }
}
```

Motivating Example

```
public void printList(List<Object> list) {
    for (int i = 0; i < list.size(); i++) {
        Object next = list.get(i);
        System.out.println(next);
    }
}
```

- We want to be able to call this with a List<String>

```
List<String> stringList = ...
printList(stringList) //Not legal
```

- We need a type which represents any type of List

Wildcards

- Type argument of the form <?>
- Ranges over all possible specific type arguments
  - List<?> is the type of all lists, regardless of parameter type

Wildcards

- Type argument of the form <?>
- Ranges over all possible specific type arguments
  - List<?> is the type of all lists, regardless of parameter type

```
public void printList(List<?> list) {
    for (int i = 0; i < list.size(); i++) {
        Object next = list.get(i);
        System.out.println(next);
    }
}
```

Wildcards

- Type argument of the form <?>
- Ranges over all possible specific type arguments
  - List<?> is the type of all lists, regardless of parameter type

```
public void printList(List<?> list) {
    for (int i = 0; i < list.size(); i++) {
        Object next = list.get(i);
        System.out.println(next);
    }
}
```

- Useful when operations do not depend on the parameter type
- Restrictions for maintaining type safety
  - Cannot insert non-null elements
  - Can read Objects

Bounded Wildcards

- Wildcard of the form C<? extends T> or C<? super T>
- Useful when wildcard required to conform to some bound.

Example

```
abstract class Shape {
    abstract double area();
}
class Square extends Shape {
    public int w;
    public double area() {return w * w;}
}
class Circle extends Shape {
    public int r;
    public double area() {return PI * r * r;}
}
```

Example

```
double sumAreas(List<Shape> list) {
    double result = 0;
    for (int i = 0; i < list.size(); i++) {
        Shape shape = list.get(i);
        result = result + shape.area();
    }
    return result;
}
```

Example

```
double sumAreas(List<Shape> list) {
    double result = 0;
    for (int i = 0; i < list.size(); i++) {
        Shape shape = list.get(i);
        result = result + shape.area();
    }
    return result;
}
```

- We want to be able to pass in a List<Circle>

Example

```
double sumAreas(List<? extends Shape> list) {
    double result = 0;
    for (int i = 0; i < list.size(); i++) {
        Shape shape = list.get(i);
        result = result + shape.area();
    }
    return result;
}
```

Extends-bounded wildcard

- Wildcard of the form C<? extends T>
- Indicates that the parameter is a subtype of T
- Cannot insert non-null elements
- Can read elements of type T

Super Bounds

Super-bounded wildcard

- Wildcard of the form C<? super T>
- Indicates that the parameter is a supertype of T
- Can insert elements of type T
- Can read Objects

Comparator Example

```
interface Comparator<T> {
    int compareTo(T first, T second);
}
```

- compareTo(a, b) $<$ 0 if a $<$ b
- compareTo(a, b) $>$ 0 if a $>$ b
- compareTo(a, b) $=$ 0 if a $=$ b

Comparator<String> comparator = ...
comparator.compareTo("apple", "banana") $<$ 0
comparator.compareTo("banana", "apple") $>$ 0

Comparator Example

- A TreeSet stores ordered data
- Can use a Comparator to define the ordering

```
class TreeSet<Circle> {
    public TreeSet(Comparator<...> c)
}
```

- Comparator must be able to compare circles

Comparator Example

- A TreeSet stores ordered data
- Can use a Comparator to define the ordering

```
class TreeSet<Circle> {
    public TreeSet(Comparator<...> c)
}
```

- Comparator must be able to compare circles

```
class TreeSet<Circle> {
    public TreeSet(Comparator<Circle> c)
}
```

Comparator Example

- A TreeSet stores ordered data
- Can use a Comparator to define the ordering

```
class TreeSet<Circle> {
    public TreeSet(Comparator<...> c)
}
```

- Comparator must be able to compare circles

```
class TreeSet<Circle> {
    public TreeSet(Comparator<Circle> c)
}
```

- A comparator that can compare shapes should work as well

Comparator Example

- A TreeSet stores ordered data
- Can use a Comparator to define the ordering

```
class TreeSet<Circle> {
    public TreeSet(Comparator<...> c)
}
```

- Comparator must be able to compare circles

```
class TreeSet<Circle> {
    public TreeSet(Comparator<Circle> c)
}
```

- A comparator that can compare shapes should work as well

```
class TreeSet<Circle> {
    public TreeSet(Comparator<? super Circle> c)
}
```

Conclusions

- Generics are a useful language feature
- Type safety and type hierarchy complicated addition of generics to Java
- Wildcards were developed to fit generics into the type hierarchy
  - Places restrictions on the use of generic types
  - Can be either bounded or unbounded

References I


G. Bracha. *Generics in the Java Programming Language.* http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2004.


G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. *Making the future safe for the past: Adding genericity to the Java programming language.* OOPSLA 1998.


D. Flanagan. *Java In A Nutshell, 5th Edition.* OReilly Media, 2005.


J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition.* The Java Series, 2005.


A. Igarashi and M. Viroli. *Variant parametric types: A flexible subtyping scheme for generics.* ACM Trans. Program. Lang. Syst., 2006.

References II


G. Rimassa and M. Viroli. *Understanding access restriction of variant parametric types and java wildcards.* In SAC 05: Proceedings of the 2005 ACM symposium on Applied computing, 2005.


Sun Microsystems, Inc. *JSR 14* http://jcp.org/en/jsr/detail?id=14, 1999.


Sun Microsystems, Inc. *Java 2 Platform Standard Edition 5.0 API Specification.* http://java.sun.com/j2se/1.5.0/docs/api/index.html, 2004.


K. K. Thorup and M. Torgersen. *Unifying genericity - combining the benefits of virtual types and parameterized classes.* ECOOP 99, 1999.

References III


M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. *Adding wildcards to the Java programming language.* SAC 04, 2004.