Polymorphism in O-O Languages: 
What are the alternatives?

Kim Bruce Williams College ⇒ UC Santa Cruz ⇒ Pomona College

Outline

• History of Generics in Java
• Language Design Proposals for Generics
• Relevant Type Issues
• An Alternative Proposal
• Pros & Cons
• Java wildcard types

History

• Java 1.0 announced in 1995
• Generics considered, but omitted from final language.
• Proposals for adding generics published starting in 1997 (or earlier)
• Java-generics mailing list set up in 1997
• Generics included in Java 5 in 2004

Why Generics?

• Java collection classes defined to hold Object:
  - List, Stack, Queue, Set, SortedSet, ...
• No restrictions on what can be added
• Casts required when remove elements
  - Slows execution and obscures code.
  - Lose advantages of static type system.

Example

class Stack {
   Stack() {...}
   void push(Object newTop){...}
   Object pop() {...}
}

Stack s;
s.push("Hello");
s.push(new Point(12,23));
String greeting = (String)(s.pop());

Lost information about what kind of values are in data structure.

Proposals for Generics
F-Bounded Polymorphism

- Bounded Polymorphism introduced by Cardelli & Wegner in 1985
- F-Bounded Polymorphism introduced in 1989 by John Mitchell & Abel group at HP Labs
- Overcome limitations of bounded polymorphism
- Bounded polymorphism first used in Eiffel

Parametric Polymorphism

class Stack<T> {
   Stack() {...}
   void push(T newTop){...}
   T pop() {...}
}
Stack<String> s;
s.push("Hello");
s.push(new Point(12,23)); // now illegal
String greeting = s.pop();

Safe and more efficient!

Bounded Polymorphism

- Can restrict the kinds of values allowed:
class ShiftList<T extends Movable> {
   T hd;
   ShiftList<T> rest;
   ...
   public void shiftAll(int x, int y) {
      hd.move(2,3); // illegal w/out bound
      rest.shiftAll(2,3);
   }
}

Problems with Expressiveness

interface Comparable {
   boolean lessThan(Comparable other);
}
class SortedList<T extends Comparable> {
   List<T> aList;
   T current;
   ...
   void insert(T newElt) {
      ...
      if (newElt.lessThan(current)){...}
      else {...}
   }
}

Implementing Comparables

Calendar implements Comparable {
   int month, day, year;
   boolean calendarLessThan(Calendar other) {
      return (month < other.month || ...);
   }
   public boolean lessThan(Comparable other) {
      if (other instanceof Calendar) {
         return calendarLessThan((Calendar)other);
      } else {
         raise new BadCalComparison(...);
      }
   }
}
Should generate static type error, not exception!

Using F-bounded Polymorphism

interface FComparable<T> {
   boolean lessThan(T other);
}
class SortedList<T extends FComparable<T>> {
   List<T> aList;
   T current;
   ...
   void insert(T newElt) {
      ...
      if (newElt.lessThan(current)){...}
      else {...}
   }
}
Guarantees T has lessThan taking parameter of type T
Implementing Comparables

class Calendar implements FComparable<Calendar> {
   int month, day, year;
   ...
   boolean lessThan(Calendar other) {
      return (month < other.month || ...);
   }
}

• F-Bounded solves problems with expressiveness
• In Pizza, GJ, and NextGen among others.

PolyJ and “with” clauses

class SortedList<T> with T {boolean lessThan(T other);} {
   List<T> aList;
   ...
   public void insert(T newElt) {
      ...
      if (newElt.lessThan(current)) {...}
      else {...}
   }
}

Equivalent to F-bounded polymorphism, but uses structural subtypes

Virtual Classes

• Designed as part of Beta
• Proposed to be added to Java by Thorup, and later with Torgersen
• Idea is to allow covariant changes to types in subclasses.

Virtual Classes Example

class List {
   typedef T as Object;
   ...
   void insert(T newElt, int where) {
      ...
   }
}
interface Comparable {
   boolean lessThan(Comparable other);
}

class ComparableList extends List {
   typedef T as Comparable;
}

class SortableList {
   typedef T as Comparable;
   typedef List T as ComparableList;
   List<T> aList; private T current;
   ...
   void insert(T newElt) {
      ...
      if (newElt.lessThan(current)) {...}
      else {...}
   }
}
More Expressiveness

class Subject {
    typedef OType as Observer;
    typedef EventType as Object;
    OType[] observers;
    void notifyObservers(EventType e) {
        for (int i = 0; i < observers.length; i++)
            observers[i].notify(this, e);
    }
}

class Observer {
    typedef SType as Subject;
    typedef EventType as Object;
    void notify(SType subject, EventType e) {...}
}

Even More Expressiveness

class MenuSubject extends Subject {
    typedef OType as MenuObserver;
    typedef EventType as MenuEvent;
    String getSelectedItem() { ... }
}

class MenuObserver extends Observer {
    typedef SType as MenuSubject;
    typedef EventType as MenuEvent;
    void notify(SType subject, EventType e) {
        ... subject.getSelectedItem() ...
    }
}

Concerns About Virtual Types

• Beta (originator of Virtual Classes) appears to be type safe, but not statically type-safe.
• Emits compiler warnings where may be unsafe and inserts run-time checks
• Beta has never been given a formal semantics, so type system has never been proven correct

Language Design Interlude

Static Overloading vs Dynamic Dispatch

• Dynamic dispatch - object receiving message determines which code will be executed.
  - Determined at run-time.
• Static overloading occurs when an object supports two or more implementations of a message name.
  - Determined at compile-time.

Overloading vs Dynamic Dispatch

class C {
    boolean eq(C other) {...}  (1)
}
class SC extends C {
    boolean eq(C other) {...}  (2) override
    boolean eq(SC other) {...}  (3) overload
}

c = new C();  c' = new SC();  sc = new SC();
c.eq(c);  c.eq(c');  c.eq(sc);
c'.eq(c);  c'.eq(c');  c'.eq(sc);
sc.eq(c);  sc.eq(c');  sc.eq(sc);
Subtype Polymorphism

- If $S <: T$, a value of type $S$ can be
  - used for a parameter of type $T$.
  - assigned to a variable of type $T$.
  - Value of type $S$ can masquerade as a value of type $T$.
- Value of expression can be from a subtype of its static type.
- In Java subclass is always a subtype.

Type-Safety Issues

- What changes in subclasses are allowed without breaking type safety?
- Java allows addition of new features (fields or methods), overriding old methods.
- Before Java 5: No changes allowed in method signature when override.
- Java 5 allows specialization of return types.

Functions

If $f$ has type $S \rightarrow T$ and $s$ has type $S$ then $f(s)$ has type $T$.

When is $S' \rightarrow T' <: S \rightarrow T$?

If $f'$ has type $S' \rightarrow T'$,
then need $f'(s)$ to have type $T$.

Variables

Variables are different from values.
Variables can be *suppliers & receivers* of values.

$x := x + 1$

If $x$ is a variable of type $T$, write type as $\text{ref} T$.

When is $\text{ref} T' <: \text{ref} T$?

To replace $x$ with type $\text{ref} T$ by $x'$ of type $\text{ref} T'$ in:

- expression: $... x ...$
- assignment: $x := e$ where $e: T$.
  
  Need $T' <: T'$.

Subtyping Functions

$S <: S' \rightarrow T'$

Contravariant for parameter types.
Covariant for result types.
Variables

\[
\begin{array}{c}
\text{val} \\
\text{T} \\
\text{T}' \\
\text{T} \\
\text{T}' \\
\end{array}
\]

Supplier: covariant; Receiver: contravariant

\[\text{ref } T' <: \text{ref } T \text{ iff } T' = T\]

Exercise

Arrays:

- If \( S <: T \), is \( S \times T \)?

Java says yes, but ... not safe!

With few exceptions, for \( F : \text{Types} \rightarrow \text{Types} \),

\[ S <: T \text{ iff } F(S) <: F(T) \]

Why Restrictions on Subclasses?

- Why restrict changing types in subclasses?
  - Even if don't care if subclasses are subtypes!
- Methods are implicitly mutually recursive!

class Example {
   void m(...) {... this.n(s) ...}
   T n(S x) {...}
end class;

class SubExample extends Example {
   ...
end class;

What is relationship of new signature of \( n \) to old if want to remain type safe?

Need \( S' \rightarrow T' <: S \rightarrow T \)

Virtual Types Are Unsafe

- Allow specialization of any type in a class when creating a subclass!
- Adopting in Java would introduce safety (and efficiency) issues similar to arrays!
- Is there a type-safe way of achieving same expressiveness?

ThisType and self-reference
Type-Checking

- Can only send binary message to object if know its exact type \( @C \).
- If \( o: @C \) and \( m: U \) then \( o.m: U \)
- If \( m: @Node \) then \( m.clone# @Node \)
- If \( n: Node \) then \( n.clone# @Node \)
- \( m.setNext#m$ legal, n.setNext(m) illegal \)

Type-Safety?

- void breakit(Node node1, Node node2) {
  ... node1.setNext(node2);
}
- breakit(aNode, bNode) OK (if ...
- (if dble\_nd doubly\_linked & nd node!

Covariant changes to parameters and instance variables
break type system!

Type-Safety?

- void breakit(Node node1, Node node2) {
  ... node1.setNext(node2);
}
- breakit(aNode, bNode) OK
  if aNode & bNode both nodes.
- breakit(dble\_nd, nd) not OK
  if dble\_nd doubly\_linked & nd node!

Covariant changes to parameters and instance variables
break type system!

Type-Checking Classes

- Type-check modularly.
- Type-check methods of class \( C \) under assumptions that hold in all extensions!
  - this: ThisClass
  - ThisClass extends C

Can prove soundness of static type system.
(EOOOP '04)
F-Bounded Not Necessary!

abstract class Comparable {
    boolean lessThan(ThisClass other);
}

class SortedList<T extends Comparable> {
    ExactList<T> aList;
    @T current;
    ...
    void insert(@T newElt) {
        ...
        if (newElt.lessThan(current)) {...}
        else {...}
    }
}

Implementing Comparables

Calendar extends Comparable {
    int month, day, year;
    boolean lessThan (ThisClass other) {
        return (month < other.month || ...);
    }
}

Type check using ThisClass extends Calendar

Comparing Proposals for Generics

Comparing Solutions

• F-Bounded and “with” statements roughly equivalent.
• Compare ThisClass with F-Bounded:
  - Both statically type-safe
  - Trade off ThisClass & @-types with complexity of recursive bounds.

Disadvantage of F-bounded polymorphism

Subclasses and subinterfaces not satisfy same constraints as parent classes and interfaces.

interface IntComp extends FComparable<IntComp> {
    boolean lessThan(IntComp other);
}

interface ExtIntComp extends IntComp{...}

ExtIntComp NOT extend FComparable < ExtIntComp >.

Disadvantages of F-Bounded Polymorphism

SortedList can be instantiated with IntComp, but not ExtIntComp!
F-bounded polymorphism not consistent with extension in Java.
Comparing with Virtual Types

- Virtual types not statically type-safe
  - Variations claimed to be type-safe, but no proof.
  - Doesn't need exact types.
- Can also express mutually recursive types with generalization of ThisClass.

Summary

- After years of discussion, Java 5 supports F-bounded parametric polymorphism.
- Many alternative approaches.
- F-Bounded polymorphism not always match subclass hierarchy.
- Virtual types provide more expressiveness, but not statically type-safe.
- ThisClass can provide similar expressiveness, but provably type-safe.

Questions?