Adding Generics to the Java™ Programming Language

Gilad Bracha
Computational Theologist
Sun Microsystems

Goals of This Talk

Familiarize you with the proposed generics extension as it affects working programmers

- Basic features and usage
- Migration of pre-existing code
- Current status

Speaker's Qualifications

- Gilad Bracha is:
  - Computational Theologist at Sun Microsystems
  - Co-author and maintainer of the Java™ Language Specification
  - Specification lead for JSR-14, "Adding Generics to the Java™ Programming Language"
  - Well-known researcher in the field of object-oriented programming languages

What Are Generics?

- Generics abstract over Types
- Classes, Interfaces and Methods can be Parameterized by Types
- Generics provide increased readability and type safety

Example

```
interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}

interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

What Generics Are Not

- Generics are not templates
- Unlike C++, generic declarations are typechecked
- Generics are compiled once and for all
- Generic source code not exposed to user
- No bloat required

How to Use Generics

```
List<Integer> xs = new LinkedList<Integer>();
xs.add(new Integer(0));
Integer x = xs.iterator.next();
```

Compare with:

```
List xs = new LinkedList();
xs.add(new Integer(0));

Integer x = (Integer)xs.iterator.next();
```

List Usage: Without Generics

```
List ys = new LinkedList();
ys.add("zero");
List yss;
yss = new LinkedList();
yss.add(ys);
String y = (String)
  ((List)yss.iterator().next()).iterator().next();
Integer z = (Integer)ys.iterator().next();
// run-time error
```

List Usage: With Generics

```
List<String> ys = new LinkedList<String>();
ys.add("zero");
List<List<String>> yss;
yss = new LinkedList<List<String>>();
yss.add(ys);
String y =
  yss.iterator().next().iterator().next();
Integer z = ys.iterator().next();
// compile-time error
```

List Implementation Without Generics

```
class LinkedList implements List {
  protected class Node {
    Object elt;
    Node next;
    Node(Object elt){elt = e; next = null;}
  }

  protected Node h, t;
  public LinkedList() {h = new Node(null); t = h;}
  public void add(Object elt){
    t.next = new Node(elt);
    t = t.next;
  }
```

List Implementation Without Generics

```
public Iterator iterator(){
  return new Iterator(){
     protected Node p = h.next;
     public boolean hasNext(){return p != null;}
     public Object next(){
      Object e = p.elt;
      p = p.next;
      return e;
  }}}}
```

List Implementation With Generics

```
class LinkedList<E> implements List<E>
  protected class Node {
    E elt;
    Node next;
    Node(E elt){elt = e; next = null;}
  }
  protected Node h, t;
  public LinkedList() {h = new Node(null); t = h;}
  public void add(E elt){
    t.next = new Node(elt);
    t = t.next;
  }
```

List Implementation With Generics

```
public Iterator<E> iterator(){
  return new Iterator<E>(){
      protected Node p = h.next;
      public boolean hasNext(){return p != null;
      public E next(){
        E e = p.elt;
        p = p.next;
        return e;}}}}
```

Generic Methods

```
class Collections {
  public static <S,T extends S> void
   copy(List<S> dest, List<T> src){...}
}

class Collection<E> {

  public <T> boolean
    containsAll(Collection<T> c) {...}

  public <T extends E> boolean
    addAll(Collection<T> c) {...}

}
```

Experimental: Wildcards

```
class Collections {
  public static <S> void
   copy(List<S> dest,
        List<? extends S> src){...}
}


class Collection<E> {

  public boolean
   containsAll(Collection<?> c) {...}

  public boolean
   addAll(Collection<? extends E> c) {...}
}
```

How Do Generics Affect My Code?

- Once in a million lines (literally), you might notice a difference
- If you think that is too much—use source 1.4, which is totally compatible
- Painless migration—You can make your code API generic without waiting for anyone else

Migration

Distinguish among several levels of compatibility:

- Language compatibility
  - All programs in existing language remain valid
- Platform compatibility
  - All programs that run on existing platform run on new platform
- Migration compatibility
  - Existing source code can be migrated to utilize new features

Why Language Compatibility Is Inadequate

- All it guarantees is that old programs mean the same thing as they used to
- Real programs use platform libraries
- If platform libraries have changed, guarantee is useless in practice
- In itself, language compatibility is a theoretical notion, but...
  - It is a prerequisite for more useful forms of compatibility

Why Language Compatibility Is Inadequate

All programs continue to work, but the guarantees are weak. One way to support platform compatibility is to ship both old and new libraries.

- Duplication/bloat
- Migration may be tough

Platform Compatibility and Migration

```
package com.vendor1;

class  Inventory{

public static void addAssembly(String name,
   Collection parts) {

   Object o = parts;

   (Collection) o;

}

public static Assembly getAssembly(String name) {...}

}

class Assembly {

   public Collection getParts(){...}

}
```

Platform Compatibility and Migration

```
package com.vendor2;
import com.vendor1.*;
...
Collection c = new Collection();
c.add(...) ; ...
Inventory.addAssembly("thingee", c);
Collection k =
   Inventory.getAssembly("thingee").getClass();
Object ok = k;
k = (Collection) ok;
```

Platform Compatibility and Migration

```
package com.vendor1;

class  Inventory{

public static void addAssembly(String name,
     Collection<Part> parts) {

      Object o = parts;

    (Collection<Part>) o;

}

public static Assembly getAssembly(String name) {...}

}

class Assembly {

      public Collection<Part> getParts(){...}

}
```

Platform Compatibility and Migration

```
package com.vendor2;
import com.vendor1.*;
...
Collection c = new Collection();
c.add(...) ; ...
Inventory.addAssembly("thingee", c); // error
Collection k =
    Inventory.getAssembly("thingee").getClass();
// error
Object ok = k;
k = (Collection) ok;
```

Why Platform Compatibility Is Inadequate

- Any vendor who wants to migrate to generics would be forced to duplicate their library
- Cannot even do this unless all libraries I depend on have migrated
- At best delays, duplication, maintenance headaches
- Cyclic dependencies force everyone to coordinate migration

Migration Compatibility

- No duplication required
- No coordination required
- Everyone migrates when they want to
- This constrains the design a great deal

Raw Types

Allow new, generic definitions to be used by old, non-generic code

```
interface List<E> { ... }
interface Iterator<E>{...}
class LinkedList<E> implements List<E> {...}
// All definitions fully generic, as before
// usage can still be non-generic
List xs = new LinkedList();
xs.add(new Integer(0));
Integer x = (Integer) xs.iterator().next();
```

Unchecked Warnings

```
public String loophole(Integer x) {
  List<String> ys = new LinkedList<String>;
  List xs = ys;
  xs.add(x); // compile-time unchecked warning
  return ys.iterator().next();
}
```

Unchecked Warnings

```
public String loophole(Integer x) {
  List ys = new LinkedList;
  List xs = ys;
  xs.add(x);
  return (String) ys.iterator().next();
// run-time error
}
```

Migration Compatibility and Reification

```
Object o = ...
(Collection<String>) o;
```

How can the run time system check this?

Requires type parameters to be ***reified***

However, reification and migration conflict!

Migration Compatibility and Reification

```
package com.vendor2;
import com.vendor1.*;
...
Collection c = new Collection();
c.add(...) ; ...
Inventory.addAssembly("thingee", c);
Collection k =
   Inventory.getAssembly("thingee").getClass();
Object ok = k;
k = (Collection) ok;
// choose between failure and unsoundness
```

Migration Compatibility and Reification

```
package com.vendor1;

class  Inventory{

public static void addAssembly(String name,
   Collection<Part> parts) {

   Object o = parts;

  (Collection<Part>) o; // fails with reification

}

public static Assembly getAssembly(String name) {...}

}

class Assembly {

   public Collection<Part> getParts()

}
```

Migration Compatibility and Reification

- Huge language design space with many variations on several orthogonal design decisions
- Have not found a combination that is sound, compatible and reified
- Not much point to reification without dynamic soundness

When Can I Start Using Generics?

- Will ship in Tiger
- Early adopters can start now!
- Prototype implementation available
- Provides drop-in compatibility with JDK™ software

How Can I Start Using Generics?

- Download prototype implementation from: http://java.sun.com/people/gbracha/generics-update.html
- Use the compiler as a drop in replacement for **`javac`**

Summary of Generics in Java™ Technology

- A good way to catch type errors up front
- Make your code more readable
- None of the C++ template drawbacks
- Easy migration path, at your own pace
- Compatible with current Java™ technology
- “Early access” available now; should ship with JDK™ software in Tiger

Credits

- Expert group membership:
  - Gilad Bracha, Sun Microsystems (chair)
  - Norman Cohen, IBM
  - Christian Kemper, Borland
  - Martin Odersky, EPFL
  - Kresten Thorup, Trifork
  - Philip Wadler, Avaya Labs

More Credits

- The javac compiler team, past and present
  - David Stoutamire
  - Neal Gafter
  - Iris Garcia
  - Bill Maddox

More Credits

Researchers from Denmark, Italy and Japan

- Mads Torgersen
- Erik Ernst
- Peter Von der Ahe
- Christian Plesner Hansen
- Mirko Viroli
- Atsushi Igarashi

Useful URLs

http://java.sun.com/docs/books/jls

http://java.sun.com/docs/books/vmspec

http://java.sun.com/people/gbracha

gilad.bracha@sun.com

jsr-14-comments@jcp.org

jsr-14-prototype-comments@Sun.com

JavaOne℠

Sun's 2003 Worldwide Java Developer Conference

Java™

java.sun.com/javaone/sf