Programming with Java Generics

Angelika Langer
Training/Consulting
www.AngelikaLanger.com

jax2005

agenda

- generics overview
- refactoring legacy into generics
- building a generic abstraction

use of non-generic collections

- no homogeneous collections
  - lots of casts required
- no compile-time checks
  - late error detection at runtime

```
LinkedList list = new LinkedList();
list.add(new Integer(0));
Integer i = (Integer) list.get(0);
String  s = (String)  list.get(0);
```

fine at compile-time, but fails at runtime

casts required

use of generic collections

- collections are homogeneous
  - no casts necessary
- early compile-time checks
  - based on static type information

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(new Integer(0));
Integer i = list.get(0);
String  s = list.get(0);
```

compile-time error

definition of generic types

```
interface Collection<A>  {
  public void add (A x);
  public Iterator<A> iterator ();
}
```

```
class LinkedList<A> implements Collection<A> {
  protected class Node {
    A elt;
    Node next = null;
    Node (A elt) { this.elt = elt;  }
  }
  ...
}
```

- *type variable* = "placeholder" for an unknown type
  - similar to a type, but not really a type
  - several restrictions
    - not allowed in new expressions, cannot be derived from, no class literal, ...

type parameter bounds

```
public interface Comparable<T> { public int compareTo(T arg); }
```

```
public class TreeMap<K extends Comparable<K>,V> {
    private static class Entry<K,V> { ... }
    …
    private Entry<K,V> getEntry(K key) {
        …
        while (p != null) {
            int cmp = k.compareTo(p.key);
            …
        }
        …
    }
    …
}
```

- *bounds* = supertype of a type variable
  - purpose: make available non-static methods of a type variable
  - limitations: gives no access to constructors or static methods

using generic types

- can use generic types with or without type argument specification
  - with concrete type arguments
    - *concrete instantiation*
  - without type arguments
    - *raw type*
  - with wildcard arguments
    - *wildcard instantiation*

concrete instantiation

- type argument is a concrete type

```
void printDirectoryNames(Collection<File> files) {
  for (File f : files)
    if (f.isDirectory())
      System.out.println(f);
}
```

- more expressive type information
  - enables compile-time type checks

```
List<File> targetDir = new LinkedList<File>();
... fill list with File  objects ...
printDirectoryNames(targetDir);
```

raw type

- no type argument specified

```
void printDirectoryNames(Collection files) {
  for (Iterator it = files.iterator(); it.hasNext(); ) {
    File f = (File) it.next();
    if (f.isDirectory())
      System.out.println(f);
} }
```

- permitted for compatibility reasons
  - permits mix of non-generic (legacy) code with generic code

```
List<File> targetDir = new LinkedList<File>();
... fill list with File  objects ...
printDirectoryNames(targetDir);
```


wildcard instantiation

- type argument is a wildcard

```
void printElements(Collection<?> c) {
  for (Object e : c)
    System.out.println(e);
}
```

- a wildcard stands for a family of types
  - bounded and unbounded wildcards supported

```
Collection<File> targetDir = new LinkedList<File>();
... fill list with File  objects ...
printElements(targetDir);
```

generic methods & type inference

- defining a generic method

```
class Utilities {
  public static <A extends Comparable<A>> A max(Iterable<A> c) {
    A result;
    for (A a : c) { if (result.compareTo(a) <0) result = a; }
    return result;
  }
}
```

- invoking a generic method
  - no special invocation syntax
    - type arguments are inferred from actual arguments (*type inference*)

```
public static void main (String[ ] args) {
    LinkedList<Byte> byteList = new LinkedList<Byte>();
    ...
    Byte y = Utilities.max(byteList);
}
```


agenda

- generics overview
- refactoring legacy into generics
- building a generic abstractions

transition to generic Java

- refactoring legacy from non-generic to generic Java has two aspects

1. refactoring code that uses generified types to take advantage of generification

2. generification of non-generic types and methods

agenda

- generics overview
- refactoring legacy into generics
  - usage of generified types and methods
  - generification of types and methods
- building a generic abstractions

refactoring usage code

- refactoring code that uses generified types is relatively easy
  - IDEs have refactoring support
- example: JDK collection framework
  - `List` has been generified to `List<E>`
  - code that uses `List` must now say: "list of what"

usage of generified types

- before refactoring

```
public static Collection
removeDirectory(Collection absoluteFiles,
                String directoryToBeRemovedFromPath) {
  Collection relativeFileNames = new HashSet();
  Iterator iter = absoluteFiles.iterator();
  while (iter.hasNext()) {
    relativeFileNames.add(
      FileUtility.relativePath(((File)iter.next()).getPath(),
      directoryToBeRemovedFromPath));
  }
  return relativeFileNames;
}
```

- start providing type arguments
  - `Collection` => `Collection<String>` oder `Collection<File>`
  - leads to warning when `iterator()` method is called

usage of generified types

```
public static Collection<String>
removeDirectory(Collection<File> absoluteFiles,
                String directoryToBeRemovedFromPath) {
  Collection<String> relativeFileNames = new HashSet<String>();
  Iterator<File> iter = absoluteFiles.iterator();
  while (iter.hasNext()) {
    relativeFileNames.add(
      FileUtility.relativePath(((File)iter.next()).getPath(),
      directoryToBeRemovedFromPath));
  }
  return relativeFileNames;
}
```

can be eliminated

- challenge: elimination of no longer needed casts
  - spread over the entire program
  - no indication in form of a warning

agenda

- generics overview
- refactoring legacy into generics
  - usage of generified types and methods
  - generification of types and methods
- building a generic abstractions

generifying legacy code

- refactor non-generic abstraction
  - turn it into a generic abstraction
  - provided it is intrisically generic
- example: JDK collections
  - re-engineer `Collection` into generic `Collection<E>`
  - retain semantics of `Collection` interface
  - new generic interface must be compatible with old non-generic interface

step 1

- methods that take or return a single element
  - replace `Object` by type parameter

before:

```
interface Collection {
  boolean add     (Object o);
  boolean contains(Object o);
  boolean remove  (Object o);
} ...
```

after:

```
interface Collection<E> {
  boolean add     (E o);
  boolean contains(E o);
  boolean remove  (E o);
} ...
```

have the semantics been preserved?

before: mixed-type collections are the norm

```
Collection c = new HashSet();
c.add(new String("abc"));
c.add(new Date());
boolean b = c.contains(Long(0));
```

after: mixed-type operations do not compile

```
Collection<Date> c = new HashSet<Date>();
c.add(new String("abc"));               ← error
c.add(new Date());
boolean b = c.contains(Long(0));        ← error
```


semantics (cont.)

- consequence:
  - a mixed-type collection must be declared as such
  - e.g. as `Collection<Object>`

```
Collection<Object> c = new HashSet<Object>();
c.add(new String("abc"));
c.add(new Date());
boolean b = c.contains(Long(0));
```

JDK Collection

- actual generification in the JDK is different

JDK:

```
interface Collection<E> {
  boolean add      (E o);
  boolean contains(Object o);
  boolean remove  (Object o);
  ...
}
```


jax2005

discussion

- JDK generification is more relaxed
  - `contains(Object)` allows passing arguments through any type of reference, not just through references to type `E`

```
Collection<Long> c = ...
void f(Object ref) {
  ...
  c.contains(ref);            // JDK permits:

  c.contains((Long)ref);      // we require:
  ...
}
```


jax2005

step 2

- methods that take a collection
  - replace `Collection` by `Collection<E>`

before:

```
interface Collection {
  boolean addAll     (Collection o);
  boolean containsAll(Collection o);
  boolean removeAll  (Collection o);
  boolean retainsAll (Collection o);
} ...
```

after:

```
interface Collection<E> {
  boolean addAll     (Collection<E> o);
  boolean containsAll(Collection<E> o);
  boolean removeAll  (Collection<E> o);
  boolean retainsAll (Collection<E> o);
} ...
```


have the semantics been preserved?

before: could use any type of collection

```
List       l = ...
Set        s = ...
Collection c = ...
c.removeAll(l);
c.containsAll(s);
```

after: collection of different element type not permitted

```
List<Long>         l = ...
Set<Object>        s = ...
Collection<Number> c = ...
c.removeAll(l);      // error
c.containsAll(s);    // error
```

alternative

- allow *all* types of collections
  - replace `Collection` by `Collection<?>`

2nd try:

```
interface Collection<E> {
  boolean addAll     (Collection<?> o);
  boolean containsAll(Collection<?> o);
  boolean removeAll  (Collection<?> o);
  boolean retainsAll (Collection<?> o);
} ...
```

```
List<Long>         l = ...
Set<Object>        s = ...
Collection<Number> c = ...
c.removeAll(l);
c.containsAll(s);
c.addAll(s);
```

does it make sense?

discussion

Q: Do we want to allow addition, removal, search of arbitrary objects to a collection of number?

A: Certainly not - but who prohibits it?

two choices:

- method accepts all types of elements and checks for itself (dynamically at runtime)
- method signature excludes arbitrary types of elements and the compiler checks (at compile-time)

alternative

- allow collections with same element type or subtype thereof
  - replace `Collection` by `Collection<? extends E>`

3rd try:

```
interface Collection<E> {
  boolean addAll     (Collection<? extends E> o);
  boolean containsAll(Collection<? extends E> o);
  boolean removeAll  (Collection<? extends E> o);
  boolean retainsAll (Collection<? extends E> o);
  ...
}
```


alternative

- in a collection of numbers we can add, remove search for longs, but not for arbitrary objects

```
List<Long>         l = ...
Set<Object>        s = ...
Collection<Number> c = ...

c.removeAll(l);
c.containsAll(s);       <---- error
c.addAll(s);            <---- error

c.addAll(l);
```

alternative without wildcards

- parameterize the methods
  - has the advantage of no restrictions on use of argument
  - wildcard type restricts usage

4th try:

```
interface Collection<E> {
  <F extends E> boolean addAll     (Collection<F> o);
  <F extends E> boolean containsAll(Collection<F> o);
  <F extends E> boolean removeAll  (Collection<F> o);
  <F extends E> boolean retainsAll (Collection<F> o);
  ...
}
```


JDK Collection

- actual generification in the JDK is different

JDK:

```
interface Collection<E> {
  boolean addAll     (Collection<? extends E> o);
  boolean containsAll(Collection<?> o);
  boolean removeAll  (Collection<?> o);
  boolean retainsAll (Collection<?> o);
  ...
}
```

step 4

- methods that convert collection to array
  - replace `Object` by type parameter

before:

```
interface Collection {
  Object[] toArray();
  Object[] toArray(Object[] a);
  ...
}
```

after:

```
interface Collection<E> {
  E[] toArray();
  E[] toArray(E[] a);
  ...
}
```


problem

- `E[] toArray()` is clearly wrong
  - collections is incapable of returning an array of `E`
  - arrays of type variables are not allowed

```
class Sequence<E> implements Collection<E> {
  ...
  E[] toArray() {
     E[] arr = new E[size];      <-- does not compile
     ... fill array with collection elements ...
  }
  ...
}
```

problem (cont.)

- if `new E[]` were allowed ...
  - what would it look like after type erasure ?

```
class Sequence implements Collection {
  ...
  Object[] toArray() {
     Object[] arr = new Object[size];
     ... fill array with collection elements ...
  }
  ...
}
```

E replaced by Object

- solution:
  - retain the original signature and return `Object[]`

problem

- `E[] toArray(E[])` is too restrictive
  - return type need not be "array of element type"
  - original semantics:
    - argument type (of array supplied) determines runtime type of result
- problem:
  - cannot put elements into array of supertype
    - although it was possible in non-generic `Collection`

```
Collection<Long> longs = ...;

Number[] numbers
   = longs.toArray(new Number[0]);
```

error

solution

- solution:
  - generify the method

2nd try:

```
interface Collection<E> {
  ...
  <T> T[] toArray(T[] a);
  ...
}
```

- note
  - implementation uses reflection to create array of correct type

JDK Collection

- actual generification in the JDK is the same

JDK:

```
interface Collection<E> {
  Object[] toArray();
  <T>  T[] toArray(T[] a);
  ...
}
```

conclusion

- refactoring a non-generic abstraction into a generic one is non-trivial
  - because original semantics must be retained *and*
  - generic API must not be more restrictive that the original

- not discussed here:
  - byte code compatibility is also required
  - usually happens automatically
  - sometimes "interesting" hacks are necessary

Collections.max()

- method that finds maximum element in collection of comparable elements
  - use bound to make sure elements are comparable
  - use wildcards to permit supertype of element type as return type

before:

```
class Collections {
  static Object max(Collection coll) {...}
  ...
}
```

after:

```
class Collections {
  static <T extends Comparable<? super T>>
  T max(Collection<T> coll) {...}
  ...
}
```

JDK Collection

- actual generification in the JDK is different

JDK:

```
class Collections {
  static <T extends Object & Comparable<? super T>>
  T max(Collection<? extends T> coll)  {...}
  ...
}
```


problem

- our API is not backward compatible
  - by using bounded type parameter we change return type

before:

```
class Collections {
  static Object max(Collection coll) {...}
} ...
```

after:

```
class Collections {
  static <T extends Comparable<? super T>>
  T max(Collection<T> coll) {...}
} ...
```

type erased:

```
class Collections {
  static Comparable max(Collection coll) {...}
} ...
```

agenda

- generics overview
- refactoring legacy into generics
- building a generic abstractions

a generic Pair class

- Implement a class that holds two elements of different types.
  - Constructors
  - Getters and Setter
  - Equality and Hashing
  - Comparability
  - Cloning
  - Value Semantics

```
final class Pair<X,Y> {
  private X first;
  private Y second;
  ...
}
```

constructors - 1st naive approach

```
final class Pair<X,Y> {
  ...
  public Pair(X x, Y y) {
    first = x; second = y;
  }
  public Pair() {
    first = null; second = null;
  }
  public Pair(Pair other) {
    if (other == null) {
      first  = null;
      second = null;
    } else {
      first  = other.first;
      second = other.second;
    }
  }
}
```

Y   Object

- does not compile

error: incompatible types

constructors - tentative fix

```
final class Pair<X,Y> {
  ...
  public Pair(X x, Y y) {
    first = x; second = y;
  }
  public Pair() {
    first = null; second = null;
  }
  public Pair(Pair other) {
    if (other == null) {
      first  = null;
      second = null;
    } else {
      first  = (X)other.first;
      second = (Y)other.second;
    }
  }
}
```

Y   Y

- insert cast

warning: unchecked cast

ignoring unchecked warnings

- what happens if we ignore the warnings?

```
public static void main(String... args) {
  Pair<String,Integer> p1
     = new Pair<String,Integer>("Bobby",10);
  Pair<String,Date> p2
     = new Pair<String,Date>(p1);
  ...
  Date bobbysBirthday = p2.getSecond();
}
```

ClassCastException

- error detection at runtime
  long after debatable assigment in constructor

constructors - what's the goal?

- a constructor that takes the same type of pair?
- allow creation of one pair from another pair of a different type, but with compatible members?

same type argument

```
public Pair(Pair<X,Y> other) {
  if (other == null) {
    first  = null;
    second = null;
  }
  else {
    first  = other.first;
    second = other.second;
  }
}
```

- accepts same type pair
- rejects alien pair

```
public static void main(String... args) {
  Pair<String,Integer> p1
     = new Pair<String,Integer>("Bobby",10);
  Pair<String,Date> p2
     = new Pair<String,Date>(p1);      ← error: no matching ctor
  ...
  Date bobbysBirthday = p2.getSecond();
}
```


downside

- implementation also rejects useful cases:

```
public static void main(String... args) {
 Pair<String,Integer> p1
   = new Pair<String,Integer>("planet earth",10000);
  Pair<String,Number> p2
   = new Pair<String,Number>(p1);      ← error: no matching ctor
  Long thePlanetsAge = p2.getSecond();
}
```

compatible type argument

```
public <A extends X, B extends Y>
Pair(Pair<A,B> other) {
  if (other == null) {
    first  = null;
    second = null;
  }
  else {
    first  = other.first;
    second = other.second;
} }
```

- accepts compatible pair

```
public static void main(String... args) {
  Pair<String,Integer> p1
   = new Pair<String,Integer>("planet earth",10000);
  Pair<String,Number> p2
   = new Pair<String,Number>(p1);     // now fine
  Long thePlanetsAge = p2.getSecond();
}
```


what does "compatible" mean?

- subtyping relationship
  - all extends and implements relationships
  - covariance relationship between “array of subtype” and “array of supertype”
  - relationship between wildcard instantiations and concrete instantiations of parameterized types
- examples:
  - `Pair<Object,Object[]>` created from `Pair<String,String[]>`
  - `Pair<String,? extends Number>>` created from `Pair<String,Integer>`

equivalent implementation

```
public Pair(Pair<? extends X,? extends Y>  other) {
  if (other == null) {
    first  = null;
    second = null;
  }
  else {
    first  = other.first;
    second = other.second;
  }
}
```

- difference lies in methods that can be invoked on `other`
  - no restriction in generic method
  - no methods that take arguments of "unknown" type in method with wildcard argument
- does not matter since we do not invoke any methods

getters and setters

```
final class Pair<X,Y> {
  ...
  public X getFirst()  { return first; }
  public Y getSecond() { return second; }
  public void setFirst(X x)  { first = x; }
  public void setSecond(Y y) { second = y; }
}
```

- add setters that take the new value from another pair

comparison

```
final class Pair<X,Y> implements Comparable<Pair<X,Y>> {
…
 public int compareTo(Pair<X,Y> other) {
  ... first.compareTo(other.first) ...
  ... second.compareTo(other.second) ...
 }
}
```

error: cannot find compareTo method

- use bounds to require that members be comparable

```
final class Pair<X extends Comparable<X>,
                 Y extends Comparable<Y>>
 implements Comparable<Pair<X,Y>> {
 …
 public int compareTo(Pair<X,Y> other) {
  ... first.compareTo(other.first) ...
  ... second.compareTo(other.second) ...
 }
}
```

now fine

comparison

- the proposed implementation does not permit pairs of "incomparable" types
  - such as `Pair<Number,Number>`
  - two flavours of generic pair class would be ideal

```
class Pair<X,Y>
```

and

```
class Pair<X extends Comparable<X>,
           Y extends Comparable<Y>>
  implements Comparable<Pair<X,Y>>
```

  - cannot define two flavors of same generic class

multi-class solution

- define separate classes

```
class Pair<X,Y>  {
  protected final int compareToImpl
   (Pair<? extends Comparable<X>,
         ? extends Comparable<Y>> other) {
    ...
  }
}

class ComparablePair<X extends Comparable<X>,
                     Y extends Comparable<Y>>
  extends Pair<X,Y>
  implements Comparable<ComparablePair<X,Y>> {
       public int compareTo(ComparablePair<X,Y> other) {
              return super.compareToImpl(other);
       }
}
```


multi-class solution - evaluation

- leads to an inflation of classes
  - `ComparablePair`, `CloneablePair`, `ComparableCloneablePair`, ...
- cannot compare compatible types
  - `ComparablePair<Integer,Integer>` cannot be compared to `ComparablePair<Number,Number>`
  - inconsistent with our implementation of `equals()`

single-class solution

- allow comparison of compatible pairs

```
final class Pair<X,Y> implements Comparable<Pair<?,?>> {
   …
   public int compareTo(Pair<?,?> other) {
    …
   }
}
```

- alternatively with raw type

```
final class Pair<X,Y> implements Comparable<Pair> {
   …
   public int compareTo(Pair other) {
    …
   }
}
```


"unchecked" warnings

```
final class Pair<X,Y> implements Comparable<Pair<?,?>> {
  public int compareTo(Pair<?,?> other) {
    ... ((Comparable)first).compareTo(other.first) ...
  }
}
```

warning: unchecked cast

- suppress with standard annotation

```
class Foo {
  @SuppressWarnings("unchecked")
  void f() {
     // code in which unchecked warnings are suppressed.
  }
}
```

clone

- two choices
  - two separate classes `Pair` and `CloneablePair`
  - one unified class `Pair`

```
class CloneablePair<X extends Cloneable,
                    Y extends Cloneable> extends Pair<X,Y>
  implements Cloneable {
             …
      public CloneablePair<X,Y> clone() {
             …
      }
}
```


single-class solution

- again: unavoidable „unchecked“ warnings
  - because `clone()` returns an `Object`

```
class Pair<X,Y> implements Comparable<Pair<?,?>>, Cloneable {
  public Pair<X,Y> clone()
    throws CloneNotSupportedException {
       ... (X)first.getClass().getMethod("clone",null)
                    .invoke(first,null); ...
  }
}
```

warning: unchecked cast

closing remarks

- greatest difficulty is clash between old and new Java
  - where generic Java meets non-generic Java
- rules of thumb:
  1. avoid raw types whenever you can
  2. avoid casts to parameterized types whenever you can

wrap-up

- refactoring legacy code to generic code
  - adds to the clarity of the code
  - is easy for code that uses generified abstractions
  - generifying an existing abstraction takes care
  - designing generic APIs is non-trivial

references

Generics in the Java Programming Language

a tutorial by Gilad Bracha, July 2004

`http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

Java Generics FAQ

a FAQ by Angelika Langer

`http://www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html`

more links ...

`http://www.AngelikaLanger.com/Resources/Links/JavaGenerics.htm`

authors

Angelika Langer

Trainer/Consultant

URL: `www.AngelikaLanger.com`

Email: `al@AngelikaLanger.com`

Klaus Kreft

Senior Consultant

Siemens Business Services

Email: `klaus.kreft@siemens.com`