Java Generics

Angelika Langer
Trainer/Consultant
www.AngelikaLanger.com

http://www.AngelikaLanger.com/
last update: 6/15/2007 ,18:45

goal

- give an overview of Java generics
- what are generics?
    - language features of generics
- what are generics used for?
    - idioms for use of generics

speaker's qualifications

- author of Java Generics FAQ online
  - www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html
- independent trainer / consultant / author
  - teaching C++ and Java for 10+ years

agenda

- language features
- usage

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

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);
```

benefits of generic types

- increased expressive power
- improved type safety
- explicit type parameters and implicit type casts

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);
```

wildcards

- a wildcard denotes a representative from a family of types
  - unbounded wildcard `?`
    - all types
  - lower-bound wildcard `? extends Supertype`
    - all types that are subtypes of `Supertype`
  - upper-bound wildcard `? super Subtype`
    - all types that are supertypes of `Subtype`

example of a bounded wildcard

- consider a method
  - that draws objects from a class hierarchy of shapes

naïve approach

```
void drawAll(List<Shape> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

- method cannot draw a list of circles
  - because `List<Circle>` is not a subtype of `List<Shape>`

```
List<Circle> circles = ... ;
drawAll(circles);
```

trying to fix it ...

- try a wildcard instantiation

wildcarded version

```
void drawAll(List<?> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

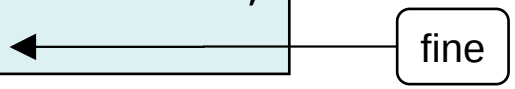
- compiler needs more information about "unknown" type

solution: upper bound wildcard

- "`? extends Shape`" stands for "any subtype of `Shape`"
  - `Shape` is the *upper bound* of the bounded wildcard
- `Collection<? extends Shape>` stands for "collection of any kind of shapes"
  - is the supertype of *all* collections that contain shapes (or subtypes thereof)

```
void drawAll(List<? extends Shape> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

```
List<Circle> circles = ... ;
drawAll(circles);
```



generic methods

- defining a generic method

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

type inference

- 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);
}
```

compilation model

- *Code specialization*.
  - new representation for every instantiation of a generic type or method
    - e.g. different code for a list of strings and list of integers
  - downside: code bloat
- *Code sharing*.
  - only one representation of a generic type or method
    - all concrete instantiations are mapped to this representation
    - implicit type checks and type conversions where needed
  - downside: no primitive types

type erasure - class definition

- generic type:

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

- after type erasure:

```
class LinkedList implements Collection {
  protected class Node {
    Object elt; Node next = null;
    Node (Object elt) { this.elt = elt; }
  }
  public void add (Object elt) { ...  }
  ...
}
```

type erasure - usage context

- generic type:

```
final class Test {
  public static void main (String[ ] args) {
    LinkedList<String> ys = new LinkedList<String>();
    ys.add("zero"); ys.add("one");
    String y = ys.iterator().next();
  }
}
```

- after type erasure:

```
final class Test {
  public static void main (String[ ] args) {
    LinkedList ys = new LinkedList();
    ys.add("zero"); ys.add("one");
    String y = (String)ys.iterator().next();
  }
}
```

additional cast

agenda

- language features
- usage

categories of usage

- using predefined generic types/methods
  - e.g. using collections such as `List<Date>`
  - requires relatively little learning effort
    - provide concrete type arguments to generic types
    - rely on type inference when calling generic methods

- designing and defining generic types/methods
  - e.g. implementing a generic `LinkedList<T>` class
  - requires sound understanding of generics

using a predefined generic type

```
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(iter.next().getPath(),
      directoryToBeRemovedFromPath));
  }
  return relativeFileNames;
}
```

- demonstrates a key benefit of generics:
  - source code is more readable and precise than without generics

same code without generics

```
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;
}
```

- it‘s difficult to tell what the collections contain

designing and defining a generic type

- case study:
  - implement a class that holds two elements of different types

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

  - constructors
  - getters and setter
  - equality and hashing
  - comparability
  - cloning
  - value semantics

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

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?

```
Pair<String,Integer> p1
   = new Pair<String,Integer>("Bobby",10);
Pair<String,Date> p2
   = new Pair<String,Date>(p1);
...
Date bobbysBirthday = p2.getSecond();
```

- 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

- accepts same type pair
- rejects alien pair

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

```
Pair<String,Integer> p1
     = new Pair<String,Integer>("Bobby",10);
Pair<String,Date> p2
     = new Pair<String,Date>(p1);
...
Date bobbysBirthday = p2.getSecond();
```

error: no matching ctor

downside

- implementation also rejects useful cases:

```
Pair<String,Integer> p1
   = new Pair<String,Integer>("planet earth",10000);
Pair<String,Number> p2
   = new Pair<String,Number>(p1);
long thePlanetsAge = p2.getSecond().longValue();
```

error: no matching ctor

compatible type argument

- accepts compatible pair

```
final class Pair<X,Y> {
 …
 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;
}}
```

```
Pair<String,Integer> p1
   = new Pair<String,Integer>("planet earth",10000);
Pair<String,Number> p2
   = new Pair<String,Number>(p1);
long thePlanetsAge = p2.getSecond().longValue();
```

now fine

equivalent implementation

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

- permits the same invocations

```
Pair<String,Integer> p1
   = new Pair<String,Integer>("planet earth",10000);
Pair<String,Number> p2
   = new Pair<String,Number>(p1);
long thePlanetsAge = p2.getSecond().longValue();
```

equivalent implementation

```
final class Pair<X,Y> {
 …
 public Pair(Pair<? extends X,? extends Y>  other) { … }
}
```

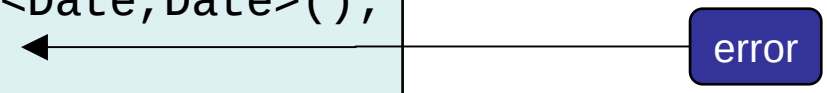
```
final class Pair<X,Y> {
 …
 public <A extends X, B extends Y> Pair(Pair<A,B> other) { … }
}
```

- permit same invocations
- difference lies in access to wildcard argument
  - not all methods of wildcard pair can be called
  - doesn‘t matter here, since we do not invoke any methods

wildcard access rules

- disallowed:
  - invoking methods that take arguments of "unknown" type
    - we do not know which type of elements a `List<?>` contains
    - hence we cannot add anything, except the typeless `null`
- allowed:
  - invoking methods that return objects of "unknown" type
    - after all it's an `Object`

```
Pair<?,?> p = new Pair<Date,Date>();
p.setFirst("xmas");
p.setFirst(null);
Object o = p.getFirst();
```



value semantics

- alternative semantics.
  - hold copies of constructor arguments, instead of just references

```
final class ValuePair<X,Y> {
  public ValuePair(X x, Y y) {
   first  = (x==null)?null:cloneObject(x);
   second = (y==null)?null:cloneObject(y));
  }

  private static <T> T cloneObject(T t) {
   try { return (T)t.getClass().getMethod("clone",null).invoke(t,null); }
   catch (Exception e) { return null; }
} }
```

cloning

- unchecked cast cannot be avoided

```
final class ValuePair<X,Y> {
  private static <T> T cloneObject(T t) {
    …
    return (T) t.getClass().getMethod("clone",null).invoke(t,null);
} } …
```

warning: unchecked cast

- use @SuppressWarnings annotation

```
final class ValuePair<X,Y> {
  @SuppressWarnings("unchecked")
  private static <T> T cloneObject(T t) {
    …
    return (T) t.getClass().getMethod("clone",null).invoke(t,null);
  } …
}
```

default constructed value pair

- default construction of a value pair is a problem
  - generic construction not permitted

```
final class ValuePair<X,Y> {
   …
   public ValuePair() {
        first  = new X();
        second = new Y();
   }
   …
}
```

```
error:
type variable not permitted
in new expression
```

- generic construction only possible via reflection
  - typically by a factory method

generic object creation

- static helper method that produces an object
  - if information about requested type of object is provided

```
final class ValuePair<X,Y>  {
 public ValuePair() {
    first  = makeObject(X.class);
    second = makeObject(Y.class);
 }
 private static <T> T makeObject(Class<T> clazz) {
   try { return clazz.getConstructor(new Class[0])
                     .newInstance(new Object[0]);
   } catch (Exception e) { return null; }
 }
}
```

class `Class<T>`

- class `Class` is generic
  - type parameter is the type that the `Class` object represents
  - e.g.: `String.class` is of type `Class<String>`

- used here to ensure consistency
  - to create a `T` object a `Class<T>` object must be provided

```
private static <T> T makeObject(Class<T> clazz) { … }
```

  - nonsense will not compile

```
Date date = makeObject(String.class);
```

error

no class literals for type variables

- default construction is still a problem
  - we cannot call the helper method
  - type variables have no class literal

```
final class ValuePair<X,Y>  {
 public ValuePair() {
    first  = makeObject(X.class);
    second = makeObject(Y.class);
 }
 private static <T> T makeObject(Class<T> clazz) { …  }
}
```

error: class literal does not exist

provide Class objects

- special purpose constructor
  - takes the required Class objects as arguments

```
final class ValuePair<X,Y>  {
 public ValuePair(Class<X> xType, Class<Y> yType) {
    first  = makeObject(xType);
    second = makeObject(yType);
 }
 private static <T> T makeObject(Class<T> clazz) { …  }
}
```

- the constructor would be used like this:
  - note the highly redundant repetition of type information

```
ValuePair<String,Date> pair
  = new ValuePair<String,Date>(String.class, Date.class);
```

factory method for value pair

- factory method = static method that produces an object
  - use instead of a constructor

```
final class ValuePair<X,Y>  {
 public static <U,V>
 ValuePair<U,V> makeDefaultPair(Class<U> uType, Class<V> vType) {
   return new ValuePair<U,V>(makeObject(uType),makeObject(vType));
 }

 private static <T> T makeObject(Class<T> clazz) { … }
}
```

- factory method is more convenient to use
  - type information is automatically inferred

```
ValuePair<String,Date> pair
  = ValuePair.makeDefaultPair(String.class, Date.class);
```

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

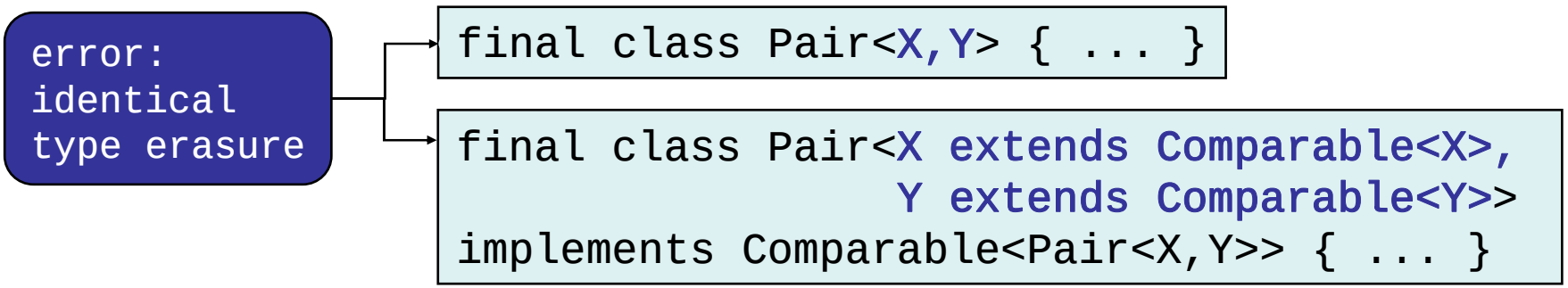
- use bounds to require that members be comparable

```
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 parameterized pair class would be ideal



multi-class solution

- defining separate classes
  - leads to an inflation of classes

```
final class Pair<X,Y>  {  ...  }
```

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

single-class solution

- requires cast

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

warning: unchecked cast

- use `@SuppressWarnings` annotation

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

summary

- Java 5.0 has language features for definition and use of generic types and methods
  - type parameters & arguments, wildcards, bounds, …
- using predefined generic types is easy
- designing and implementing generic APIs is challenging
  - requires understanding of compilation model, various restrictions and options, …

for more information

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 Language Specification, 3rd Edition
by Gosling, Joy, Steele, Bracha, May 2005
`http://java.sun.com/docs/books/jls`

Java Generics FAQ
a FAQ by Angelika Langer
`http://www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html`

Java Generics and Collections
Maurice Naftalin & Philip Wadler
O'Reilly, 2006

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

author

Angelika Langer

Training & Mentoring

Object-Oriented Software Development in C++ & Java

Email: **contact@AngelikaLanger.com**

http: **www.AngelikaLanger.com**

Java Generics

Q & A