Generics Written in the Java™ Programming Language with Parameterized Types in Java Platform 5.0

Angelika Langer

Trainer/Consultant
www.AngelikaLanger.com

TS-5627

Goal

Give an overview of generics for the Java™ programming language

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);
```

Compile-time Error

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);
```

Incompatible Argument Type

Trying to Fix It…

- Try a wildcard instantiation

Wildcarded version

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

Error: ? Does Not Have a Draw Method

- 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);
```

fine

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
  - Constructors
  - Getters and setter
  - Equality and hashing
  - Comparability
  - Cloning
  - Value semantics

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


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—First Naïve 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

error: incompatible types

- Does not compile

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

warning: unchecked cast

- Insert 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();   <- 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

```
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;
}}}
```

- Accepts same type pair
- Rejects alien pair

```
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

```
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().longValue();
```

Compatible Type Argument

```
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;
}}}
```

- Accepts compatible pair

```
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();
```

fine

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");    <-- Error
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);   <- error: class literal does not exist
 }
 private static <T> T makeObject(Class<T> clazz) { …  }
}
```

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

```
error:
identical
type erasure
```

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

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

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 platform 5.0 has language features for definition and use of generic types and methods
  - Type parameters and 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
- More links...
  - http://www.AngelikaLanger.com/Resources/Links/JavaGenerics.htm

Q&A

Angelika Langer

Submit Session Evaluations for Prizes!

Your opinions are important to Sun

- You can win a $75.00 gift certificate to the on-site Retail Store by telling Sun what you think!
- Turn in completed forms to enter the daily drawing
- Each evaluation must be turned in the same day as the session presentation
- Five winners will be chosen each day (Sun will send the winners e-mail)
- Drop-off locations: give to the room monitors or use any of the three drop-off stations in the North and South Halls

Note: Winners on Thursday, 6/30, will receive and can redeem certificates via e-mail.

Generics Written in the Java™ Programming Language with Parameterized Types in Java Platform 5.0

Angelika Langer

Trainer/Consultant
www.AngelikaLanger.com

TS-5627