Programming with Java Generics

Angelika Langer
Training/Consulting
www.AngelikaLanger.com

agenda

- generics overview
- 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);
```

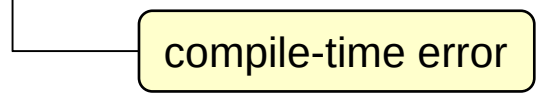
fine at compile-time, but fails at runtime

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);
```



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
- building a generic abstraction

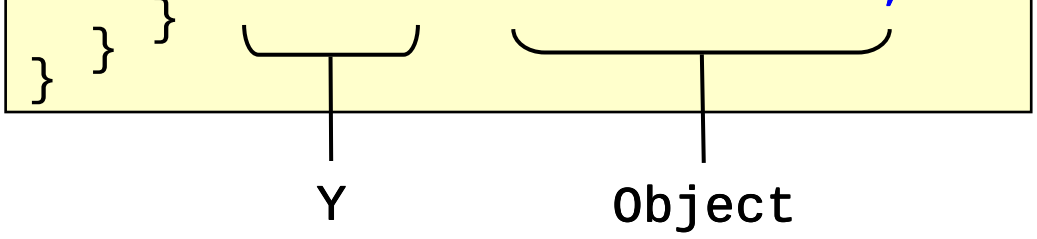
a generic Pair class

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

```
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;
    }
  }
}
```



- 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

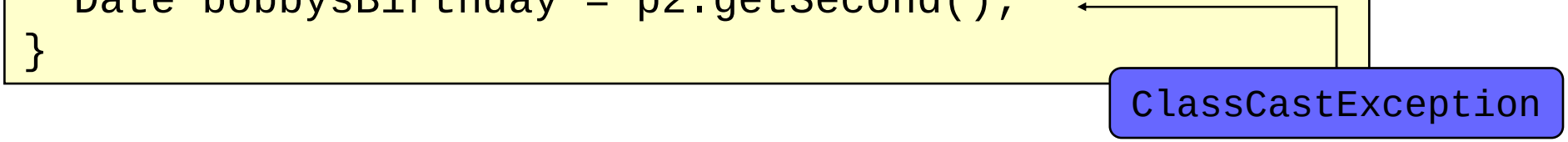
- insert cast

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();
}
```



- error detection at runtime
  long after debatable assignment 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);
  ...
  Date bobbysBirthday = p2.getSecond();
}
```

error: no matching ctor

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);
  Long thePlanetsAge = p2.getSecond();
}
```

error: no matching ctor

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);
  Long thePlanetsAge = p2.getSecond();
}
```

now fine

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

equals

- straightforward traditional implementation

```
public boolean equals(Object other) {
  if (this == other) return true;
  if (other == null) return false;
  if (getClass() != other.getClass()) return false;
  Pair otherPair = (Pair)other;
  ...
  if (!first.equals(otherPair.first)) return false;
  ...
  return true;
}
```

cast to raw type
to avoid unchecked warning

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>  {
  ...
}
```

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

  - leads to a large number of classes

single-class solution

- allow comparison of compatible pairs
  - comparison will fail (with ClassCastException) if parts are incomparable

```
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: method invocation on raw type

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

does not help; Cloneable is empty

covariant return type

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

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: `info@AngelikaLanger.com`

Klaus Kreft

Senior Software Architect

Siemens AG

Email: `klaus.kreft@siemens.com`