Course #4122
The New Java:
Java Generics and Beyond

Christian Kemper
JBuilder R&D, Borland

The New Java

- Generics
- Autoboxing
- Enhanced `for` Loop
- Enumerations
- Static imports

The Java Collection Framework

- **The Problem:**
  - **Java Collection framework is very powerful**
  - **but it comes at the expense of type safety (Everything is an `Object`)**

A Collections Example

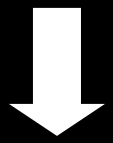
```
List stringList = new ArrayList();
...
stringList.add(Boolean.TRUE);
...
//This will be a runtime error
String value =
    (String)stringList.get(0);
```

- The Problem:
  - Runtime errors are hard to find!
  - The person finding it may be your best customer!

Generics

- Allows programmers to specify the types allowed for Collections
- Allows the compiler to enforce the type specifications

```
//This should only contain Strings
List stringList;
```



```
//This can only contain Strings
List<String> stringList;
```

A Typesafe Collections Example

- The solution:
  - Compile time errors are easy to find
  - Your integration team will find it!

```
List<String> stringList =
    new ArrayList<String>();
...
//This will be a compile time error
stringList.add(Boolean.TRUE);
...
String value = stringList.get(0);
```

Primitive Types and Collections

- The Problem:
  - It is difficult to use the primitive types (such as `int`, `boolean`) in Collections:

```
List intList = new ArrayList();
...
intList.add(new Integer(5));
...
int value =
  ((Integer)intList.get(0)).intValue();
```

Autoboxing

- Essentially bridges between the "primitive" types (such as `int`, `boolean`) and the "boxed" types (such as `Integer`, `Boolean`)

```
...
int primitive = 5;
Integer boxed = unboxed;
int unboxed = boxed + 5;
...
```

Autoboxing and Collections

- The solution:
  - Generics and Autoboxing working together

```
List<Integer> intList =
    new ArrayList<Integer>();
...
intList.add(5);
...
int value = intList.get(0);
```

Traversing a Generic Type

- Counting the letters in a document:

```
int countLetters(List<List<String>> doc) {
  int count = 0;
  for (Iterator<List<String>>i = doc.iterator();
       i.hasNext();) {
    List<String>line = i.next();
    for (Iterator<String>j = line.iterator();
         j.hasNext();) {
      String word = j.next();
      count += word.length();
    }
  }
  return count;
}
```

Enhanced `for` loop

- A new language construct for the `Iterator` pattern:

```
for (Iterator i = line.iterator();
     i.hasNext();) {
  String word = (String)i.next();
  ...
}
```

```
for (String word : line){
  ...
}
```

Traversing Example Revisited

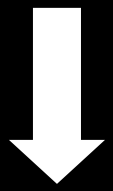
- Counting the letters in a document using the enhanced `for` loop:

```
int countLetters(List<List<String>> doc) {
  int cont = 0;
  for (List<String>> line : doc) {
    for (String word : line) {
      count += word.length();
    }
  }
}
```

Enhanced `for` loop

- And it works for arrays, too!

```
for (int i = 0; i < line.length; i++) {
  String word = line[i];
  ...
}
```



```
for (String word : line){
  ...
}
```

Enumerations

- Typical use today is the “int enum pattern”:

```
public static final int PENNY = 1;
public static final int NICKEL = 5;
```

Typesafe Enumerations

- Better is the “typesafe enum pattern”:

```
public class Coin {
  private final int value;
  public Coin(int value) {this.value = value};
  public static final Coin PENNY = new Coin(1);
  public static final int NICKEL = new Coin(5);
}
```

- Problem:
  - Lots of boilerplate code to write and maintain
  - Cannot easily use in `switch()` statements

Enumerations

```
public enum Coin {
  penny(1), nickel(5);
  private final int value;
  Coin(int value) {this.value = value}
}
```

- Essentially the “typesafe enum pattern”
- Simple C-style enumeration
- Can add behavior to each instance
- Can be used in `switch()` statement
- Efficient `Set` implementation

Static import

Questions?

Thank You

Please fill out the speaker evaluation

You can contact me further at …
Christian.Kemper@borland.com


BorCon™
2003
Borland®
Conference