Java Reification

Michele Orrù

presents:

Java Reification: how is changed working with Arrays from the introduction of Generics

taken from:

Java Generics and Collections

Maurice Naftalin & Philip Wadler

O'REILLY publications

Java Reification

array creation

Arrays reify their component types: this means that informations about their component types are available at run-time.

Java Reification
array creation

We can take this example:

```
Integer[] ints = new Integer[] {1,2,3};
Number[] nums = ints;
nums[2] = 3.14;  // array store exception
```

The exception is thrown because the assignment is not valid: the assigned double value is not compatible with the reified type of the array.

Java Reification

array creation

Another example:

```
import java.util.*;
class Annoying {
  public static <T> T[] toArray(Collection<T> c) {
    T[] a = new T[c.size()];  // compile-time error
    int i=0; for (T x : c) a[i++] = x;
    return a;
  }
}
```

Arrays must reify their component types: of course a **type variable is not a reifiable type**.

Java Reification

array creation

Another example with the same compilation error:

```
import java.util.*;
class AlsoAnnoying {
  public static List<Integer>[] twoLists() {
    List<Integer> a = Arrays.asList(1,2,3);
    List<Integer> b = Arrays.asList(4,5,6);
    return new List<Integer>[] {a, b};  // compile-time
                                        // error
  }
}
```

Arrays must reify their component types: of course a **a parameterized type is not a reifiable type**.

Java Reification

array creation

So…

is **not possible** to create **arrays with Generics**, because generics are implemented via erasure.

The best way to proceed instead to use arrays is to use data structures from the **Collections Framework**.

Java Reification

The Principle of Truth in Advertising

How to convert a collection to array?

....

Java Reification

The Principle of Truth in Advertising

The easy way to think is to **cast** every component of the collection to a generic type T[].

But???

....

Java Reification

The Principle of Truth in Advertising

The result is an unchecked cast, that can lead in more complex problems than a simple warning.

```
import java.util.*;
class Wrong {
  public static <T> T[] toArray(Collection<T> c) {
    T[] a = (T[])new Object[c.size()];  // unchecked cast(COMPILE
                                        // TIME)
    int i=0; for (T x : c) a[i++] = x;
    return a;
  }
  public static void main(String[] args) {
    List<String> strings = Arrays.asList("one","two");
    String[] a = toArray(strings);  // class cast error (RUNTIME)
  }
}
```

Java Reification

The Principle of Truth in Advertising

Let's see how the things work with type erasure, to better understand the errors:

```
import java.util.*;
class Wrong {
  public static Object[] toArray(Collection c) {
    Object[] a = (Object[])new Object[c.size()];  // unchecked cast
    int i=0; for (Object x : c) a[i++] = x;
    return a;
  }
  public static void main(String[] args) {
    List strings = Arrays.asList(args);
    String[] a = (String[])toArray(strings);  // class cast error
  }
}
```

The array in the main method contains only strings, but **its reified type indicates that it is an array of Object**, so the cast fails.

Java Reification

The Principle of Truth in Advertising

To avoid the previous problem, we have to follow the ***Principle of Truth in Advertising***: **the reified type of an array must be a subtype of the erasure of its static type**.

Using the previous example, the run-time error on class cast was because **Object** (*reified type of the array*) is not a subtype of **String** (*the erasure of its static type*).

Java Reification
The Principle of Truth in Advertising

The previous error reveal another problem..

The cast-iron guarantee is not properly respected: **in theory** *no cast inserted by erasure can fail*, but as we seen before, **in practice** *can happen*.

This is why code that generates unchecked warnings must be written with extreme care!!!

Java Reification
The Principle of Truth in Advertising

A way to solve the problem: reflection is used to allocate a new array with the same reified type as the old, if the array is NOT big enough to hold the collection.

```
import java.util.*;
class Right {
  public static <T> T[] toArray(Collection<T> c, T[] a) {
    if (a.length < c.size())
      a = (T[])java.lang.reflect.Array.newInstance
(a.getClass().getComponentType(), c.size());  // unchecked cast
    int i=0; for (T x : c) a[i++] = x;
    if (i < a.length) a[i] = null;
    return a;
  }
  public static void main(String[] args) {
    List<String> strings = Arrays.asList("one", "two");
    String[] a = toArray(strings, new String[0]);
    assert Arrays.toString(a).equals("[one, two]");
    String[] b = new String[] { "x","x","x","x" };
    toArray(strings, b);
    assert Arrays.toString(b).equals("[one, two, null, x]");
  } }
```

Java Reification
The Principle of Truth in Advertising

An alternative to using an array to create an array is to use an instance of class Class.

Instances of the class Class represent information about a class at run time.

In Java 5, the class Class has been made generic, and now has the form *Class*<*T*>.

We can define a variant of our previous method that accepts a class token of type Class<T> rather than an array of type T[].

Java Reification

The Principle of of Indecent Exposure

There exist another important principle to help us working with arrays, **The Principle of of Indecent Exposure**, that says:
never publicly expose an array where the components do not have a reifiable type (we add...especially if we write a library)

Why???

....

Java Reification
The Principle of of Indecent Exposure

Let's see an example:

```
List<Integer>[] intLists
  = (List<Integer>[])new List[] {Arrays.asList(1)};
                                              // unchecked cast
List<? extends Number>[] numLists = intLists;
numLists[0] = Arrays.asList(1.01);
int n = intLists[0].get(0);  // class cast exception
```

Java Reification
The Principle of of Indecent Exposure

Let's see another example defining a simple library:

```
DeceptiveLibrary.java:
import java.util.*;
public class DeceptiveLibrary {
  public static List<Integer>[] intLists(int size) {
    List<Integer>[] intLists =
      (List<Integer>[]) new List[size];  // unchecked cast
    for (int i = 0; i < size; i++)
      intLists[i] = Arrays.asList(i+1);
    return ints;
  }
}
```

In compile time (javac -Xlint:unchecked) only the unchecked cast is shown, so we can think that is innocuous...

Java Reification
The Principle of of Indecent Exposure

The class below uses the previous library:

```
InnocentClient.java:
import java.util.*;
public class InnocentClient {
  public static void main(String[] args) {
    List<Integer>[] intLists = DeceptiveLibrary.intLists(1);
    List<? extends Number>[] numLists = intLists;
    numLists[0] = Arrays.asList(1.01);
    int i = intLists[0].get(0);  // class cast error!
  }
}
```

As you can see the previous library was NOT so innocuous...

Java Reification

The Principle of of Indecent Exposure

Resuming:

The **Principle of Truth in Advertising** requires that the ***run-time type of an array is properly reified***.

The **Principle of Indecent Exposure** requires that the ***compile-time type of an array must be reifiable***.

Java Reification

how to define an ArrayList

There are very few cases in which is more useful declare arrays instead of collections.

One situation in which we NEED to use arrays is with ArrayList.

Java Reification

how to define an ArrayList

Implementations of those type of data structures like ArrayList need to be written with care, as they necessarily involve use of unchecked casts.

We will see in the IDE how the Principles of Indecent Exposure and of Truth in Advertising figure in the implementation.

Java Reification

array creation and varargs

Do you remember the new notation of Java 5 for varargs???

...

Well, using the ellipsis (...) we are able to create methods that can accept an indefinite number of arguments packing them into an array.

Useful, no???

Java Reification

array creation and varargs

The bad news is that such implementation suffer to the same **problems** that involve **reification as other arrays**.

Java Reification

array creation and varargs

Let's take java.util.Arrays.asList method, declared as following:

```
public static <E> List<E> asList(E... arr)
```

And let's see some examples:

```
List<Integer> a = Arrays.asList(1, 2, 3);
```

it could be declared also as

```
List<Integer> a = Arrays.asList(new Integer[] { 1, 2, 3 });
```

No problems until now...

But...

Java Reification

array creation and varargs

If we consider this example:

```
List<List<Integer>> x = Arrays.asList(a, b);
```

and we try to declare it, as before, in the equivalent way:

```
List<List<Integer>> x =
Arrays.asList(new List<Integer>[] { a, b });
```

...we will have an unchecked generic array creation WARNING at compile time, because List<Integer> is not a reifiable type

Java Reification

array creation and varargs

So...???

Never use varargs facilities when we are not sure if the arguments will be or not reifiable types.

All these problems with varargs would not exists now if the designers had been used a **collection instead** and **array**: T... to be equivalent to List<T> rather than T[].

Java Reification

Arrays as a Deprecated Type?

Why use arrays if collections have the following features:

- more precise typing, means that more errors can be detected at compile-time;
- more flexibility, in the declaration and during the use, large choice of methods and convenience algorithms;
- any type elements, instead of arrays that should have only reifiable types.

Java Reification
Arrays as a Deprecated Type?

There are some places in Java 5 where the use of collections instead of arrays would be preferred, like:

- varargs with their possible arguments of non-reifiable type, as we've analyze before;

- Principle of Indecent Exposure violated in Java libraries like

TypeVariable<Class<T>>[] java.lang.Class.getTypeParameters()
TypeVariable<Method>[] java.lang.Reflect.Method.getTypeParameters()

Java Reification

Arrays as a Deprecated Type?

Some comments of the work done by Java 5 designers:

- If the designers had been willing to restrict the notion of reified type, they could have simplified it by including raw types (such as List), but excluding types with unbounded wildcards (NOTE THAT THE ACTUAL SITUATION IS NOT LIKE THAT);

- also if only reifiable types are permitted with arrays, anyway is possible to **bypass this restriction casting to an array type that is not reifiable**, violating the cast-iron guarantee and creating unchecked warnings.

- Use of lists could be made easier by permitting Java programmers to write l[i] as an abbreviation for l.get(i), and l[i] = v as an abbreviation for l.put(i,v). (Some people like this sort of "syntactic sugar," while others think of it as "**syntactic rat poison**.")

Java Reification

Arrays as a Deprecated Type?

After all these considerations, why not consider arrays as deprecated types, ans use ONLY collections?