The Cost of Erasure in Java Generics Type System

Jaime Niño
Computer Science department
University of New Orleans
New Orleans, LA 70148
jaime@cs.uno.edu

ABSTRACT

Java generics, found in versions 1.5 and later, are implemented by generating only one byte code representation of a generic type or method; a compiler maps all the instantiations of a generic type or method to that unique representation via type erasure. The essence of type erasure is the removal during compilation of all information that is related to type parameters and type arguments. From the point of view of the programmer, (s)he is left to negotiate a series of compiler and run-time errors that have little to do with programming errors but a lot to do with the Java generics implementation choice and leaving programmer sometimes with no choice but to live with unclean compiles.

INTRODUCTION

Java generics are implemented by generating only one byte code representation of a generic type or method; a compiler maps all the instantiations of a generic type or method to that unique representation via type erasure. This choice of implementation rises to several rather surprising errors that have nothing to do with programming but only with the consequences of this choice. Beginner Java generics programmers are left to sort out lists of errors which sometimes could not be completely done away with. This may lead to the unpleasant fact that instructors may need to admit that there may be times a clean compile may not be achievable; thus opening a rat's nest of errors for which a student may unwittingly settle. In this paper we will discuss the consequences of type erasure and provide guidelines for instructors teaching Java generics.

The discussion starts with the basic notions of raw types and compiler error messages; then it proceeds to state the erasure mechanism. It then follows to define reifiable types which have complete runtime type information and discuss their relation to arrays and their component type. The paper then addresses runtime type information method `instanceof` and casting and their impact on the implementation of `equals` and `clone` methods. The paper closes by mentioning other limitations caused by type erasure.

We will assume basic reader's knowledge of Java's generics for the definition of a generic interface such as `List<T>` and two implementations `DefaultList<T>`, using `Vector<T>` and `LinkedList<T>`, which we will use throughout the paper. Consult [3] for details on their implementations. We also assume reader's knowledge of generic instantiations of a generic type, and of Java's wildcard type.[1,5]

RAW TYPES

The use and instantiation of a generic type without type arguments is called a *raw type*. As an example `List` is called the raw type produced from the type `List<T>`.

The raw type is the supertype of all its generic instantiations including wildcard instantations. Assignment of a generic instantiation to its raw type is permitted without warnings; assignment of the raw type to an instantiation yields an "*unchecked conversion*" warning.

Example (of assignment compatibility):

```
DefaultList rawList = new DefaultList();
DefaultList<String> stringList = new DefaultList<String>();
rawList    = stringList;
stringList = rawList;// unchecked warning
```

Regarding conversions, the usual reference widening conversion from subtype to supertype is allowed. The reverse is permitted too for reasons of compatibility between generic and non-generic types. It is the so-called *unchecked conversion* and is accompanied by an unchecked warning. The conversion from the raw type to the unbounded wildcard instantiation is warning-free. (See the implementation of `equals` below as well.)

It is worth noticing that the raw type `List` resulting from erasure is not the same as the parameterized type `List<Object>`; when using the raw type the compiler has not knowledge there are any restrictions on the type of elements allowed to be added by the list; but it lets you insert elements of any type along with a warning to indicate is not type safe: if you insert an element of the wrong type, you may get a `ClassCastException` at any point in the later execution of the program. But when

`List<Object>` is used, the compiler knows the list is allowed to contain elements of any type and no such exception will occur.

Raw types exist in Java 1.5 and later for reasons of compatibility with legacy java code; thus their use should only be considered in these situations. For new Java code such usage should be discourage. Students writing generics code make use of raw types only accidentally and instead of parameterized types; this accidental use results in the mixing of raw types with generic and parameterized types giving rise to untold error warning generated at compilation and which students tend to ignore at their on peril without sorting out the root cause of the warning.

COMPILER MESSAGES

An "`unchecked warning`" is a warning produced by the compiler to indicate that it cannot ensure type safety. It refers to the fact that the compiler and the runtime system do not have enough type information to perform all type checks that would be necessary to ensure type safety. The most common source of unchecked warnings is caused by the use of raw types because it does not have enough type information to perform all necessary type checks. As an example, consider:

```
List aL = new DefaultList();
aL.add("abc");   // unchecked warning
```

When add is invoked the compiler does not know whether it is safe to add a String object to the collection; the call is potentially unsafe and an unchecked warning is issued. Unchecked warnings are also reported when the compiler finds a cast whose target type is either a *parameterized* type or a *parameter type* as we will see in section 7. You need to resolve warnings due to accidental uses of raw types or else ignore them at your own risk resulting in potential ClassCastException during program execution.

GENERIC TYPES TRANSLATION: TYPE ERASURE

The compiler generates only one byte code representation of a generic type or method and maps all the instantiations of the generic type or method to that unique representation. This mapping is performed by *type erasure*. The essence of type erasure is the removal of all information that is related to type parameters and type arguments. In addition, the compiler adds type checks and type conversions where needed and inserts *synthetic bridge methods* when necessary. A bridge method is generated when a type extends or implements a parameterized class or interface and type erasure changes the signature of any inherited method. The steps performed during type erasure include:

Eliding type parameters: When the compiler finds the definition of a generic type or method, it removes all occurrences of each type parameter replacing it by its left most bound, or Object if no bound is specified.

Eliding type arguments: When the compiler finds a parameterized type, an instantiation of a generic type, it removes the type arguments. For instance, the type `List<String>` is translated to `List`.

A side effect of type erasure is that the virtual machine has no information regarding type parameters and type arguments; thus the JVM cannot tell the difference between a `List<String>` and a `List<Date>`; therefore we cannot overload a method based on types resulting from different instantiations of the same generic type.

We refer the reader to the appendix for a complete example of code before and after type erasure; for more detaisl on the erasure mechanism see [1,2].

REIFIABLE TYPES

Types that do not lose any information during type erasure are called *reifiable* types. Because some type information is erased during compilation, not all statically defined types are available at run time. Types that are completely available at runtime are known as reifiable types. The following are reifiable types: non-generic types, non-parameterized types, wildcard parameterized types, raw types, primitive types and arrays of reifiable types.

Array component type

The only allowed types for array creation are: primitive types, non-generic (or non-parameterized) reference types, unbounded wildcard instantiations, and raw types resulting from generic types.

Examples:

```
int[]     table1 = new int[10];
String[]  table2 = new String[10];
List<?>[] table3 = new DefaultList[10];
```

```
List []   table4 = new DefaultList[10];
```

What must be noticed is that the component type cannot be parameterized types. Because of type erasure, parameterized types do not have exact runtime type information. Thus, the array store check does not work because it uses the dynamic type information regarding the array's (non-exact) component type. Only arrays with an unbounded wildcard parameterized type as the component type are permitted. More generally, only reifiable types are permitted as component type of arrays.

Also, although you can declare arrays using a type parameter `T`, or using parametric classes, instances of such arrays are not allowed and will produce compilation errors. Specifically,

```
1.  public class MyContainer<T> {
2.   private T[] elements;
3.   private java.util.Collection<Integer> table;
4.   public Container(int size){
5.     elements = new E[size];
6.     table = new ArrayList<Integer> [size];
7.   }
8.     ...
9.  }
```

will produce "*generic array creation*" errors in the lines 5 and 6, where `elements` and `table` are created. Since arrays are used to implement collection of objects, we can get around it by creating the arrays having component `Object`:

```
public class MyContainer<T> {
  private T[] elements;
    ...
  public MyComp(int size){
    elements = (T[])new Object[size];
  }
  ...
}
```

The cast to `T[]` will produce an unchecked cast warning due to type erasure; but the specification of the `List`'s `add(T element)` will guarantee that only elements of type `T` will be added to the list. Thus, we will need to live with this warning when using arrays as data components in a generic type implementation.

Why does the code above not generate an `ArrayStoreException` when `elements` is accessed? After all, if `T` is instantiated with `String` you can't assign an array of `Object` to an array of `String`. Well, because generics are implemented by erasure, the type of `elements` is actually `Object[]`, because `Object` is the erasure of `T`. This means that the class is really expecting `elements` to be an array of `Object` anyway, but the compiler does extra type checking to ensure that it contains only objects of type `T`.

DYNAMIC TYPE CHECKING AND TYPE CASTING

Due to type erasure the run-time type information is incomplete and some types will result equivalent under runtime typing. For example, recall that `getClass`, defined in `Object` and inherited by all classes returns an instance of the class *java.lang.Class* that models the run-time class of an object. If

```
List<Student> roll = new DefaultList<Student>();
List<Circle> loops = new DefaultList<Circle>();
```

then

`roll.getClass().equals( loops.getClass())` returns `true`, and both

`roll.getClass().toString()`, and `loops.getClass().toString()` return `DefaultList`.

Now, only reifiable types are allowed for `instanceof` expressions, else a compile-time error is generated; hence, no parametric types are allowed because their dynamic type after type erasure is just the raw type. Examples:

```
Object obj = new LinkedList<Long>();
System.out.println(obj instanceof List);
System.out.println(obj instanceof List<?>);
```

While those lines are legal, the expressions below will generate errors:

```
obj instanceof List<Long>
obj instanceof List<? extends Number>
obj instanceof List<? super Number>
```

A cast whose target type is either a parameterized type or a bounded wildcard parameterized type or a type parameter is unsafe. The runtime part of the cast is performed based not on the exact static type, but on the resulting type erasure. In a way, the source code is misleading, because it suggests that a cast to the respective target type is performed, while in fact the

dynamic part of the cast only checks against the type erasure of the target type. The unchecked warning is issued to draw the programmer's attention to the mismatch between the static and dynamic aspect of the cast.

Occasionally, we would like to cast to a parameterized type, just to discover that the compiler flags it with an unchecked warning. Use of an *unbounded* wildcard parameterized type instead of a concrete parameterized type or a bounded wildcard parameterized type would help avoid the warning.

A typical example is the implementation of methods such as the `equals` method, that takes `Object` reference and were a cast down to the actual type must be performed; another such is the `clone` method.

`Equals` method in a generic class

Assume we define a generic class `Pair` with the `equals` method:

```
public class Pair<T> {
  private T fst, snd;
  public Pair(T t1, T t2){
    fst = t1;
    snd = t2;
  }
  public boolean equals( Object other) {
    if (other instanceof Pair){
      Pair<T> otherPair = (Pair<T>) other;
        return this.fst.equals(otherPair.fst) && this.snd.equals(otherPair.snd)
    }
    return false;
  }
  //...
}//end class Pair
```

First, note the use of `instaceof` for the testing of the argument's type; it uses the raw type `Pair` and not `Pair<T>`. It is followed by a parametric cast which will produce an unchecked warning error as the cast target is not a reifiable type. Although the type cast will be incorrect in many cases, when successful it will cast to `Pair`; the component-wise equality test will catch the different types between components of `this` and that of `otherPair` producing the expected `false` answer when they have different argument type for the generic instantiations; this is the case because every implementation of `equals` is responsible for performing a check for type matches.

But the program will not compile clean, having the warning error. We can eliminate the warning error by casting using a wildcard argument; in the `equals` version below we also rewrite the code by not using the `instanceof` operator but `getClass()` method to compare the actual runtime class representations:

```
public boolean equals( Object other) {
  if (this == other) return true;
  if (other == null) return false;
  if (this.getClass() != other.getClass())
    return false;
  Pair<?> otherPair = (Pair<?>) other;
  return this.fst.equals(otherTriple.fs) && this.snd.equals(otherTriple.snd;
}
```

Only the cast to `Pair<?>` is accepted without a warning. Note, this technique works only because we need no write access to the fields of the object referred via the wildcard parameterized type. Remember, use of the object that a wildcard reference variable references is restricted. In other situations use of a wildcard parameterized type might not be a viable solution, because full access to the referenced object might be needed.

When implementing the `equals` method for collections, it is a bit harder to do using the unbounded wildcard casting, although it can be done. It is to note that `java.util.collections` provides implementations of collections (`HashTable, HashSet`, etc) where the equality of collections is implemented using a parameterized casting and not an unbounded wildcard casting, hence yielding an unchecked warning error at compilation.

`clone` method and generic classes

In general one would proceed as it has been the usually common implementation of clone and invoke it on the type components. Unfortunately, you can't call the type parameter `clone()` because it has protected access in `Object`, and to call it you have to call do it through a reference to a class that has overridden `clone()` to be public. But the type parameter is not known to re-declare it as public, so this implementation is not workable; thus the implementation of the clone method for generic classes is done via reflection with three caveats.

First, since `super.clone()` returns `Object` we must cast the result to the generic type producing the inevitable unchecked warnings. Likewise, we must cast the result of cloning any fields to the type that the type parameter stands for producing same warnings. Finally, if the generic type has fields that are of a parameterized type and these fields must be cloned then a cast from `Object` to the parameterized type might be necessary and the compiler issues the usual unchecked cast warnings.

Second, we must invoke the fields' `clone()` method via reflection because we do not know whether the respective field has an accessible clone method. This applies to fields of generic types, which for our purposes are unknown. Even if we suspect that the type parameter will be cloneable for purposes of invoking a clone method the `Cloneable` interface is totally irrelevant, thus we cannot be sure the type parameter has a `clone()` method. Hence we must use reflection to find out whether the field has a public clone method. If it has a clone method, we invoke it.

Third, we must deal with exceptions generated by the reflection `getMethod()`. For a generic class we do not know anything about those fields of the class whose type is a type parameter. In particular, we do not know whether those fields are `Cloneable` and/or have a clone method. The attempted invocation of the members' clone method via reflection bears the risk of failure, indicated by a number of exceptions raised by `Class.getMethod` and `Method.invoke` such as `NoSuchMethodException , IllegalArgumentException`, etc. In this situation the clone method might in fact fail to produce a clone and it might make sense to indicate this failure by mapping all (or some) exceptions to a `CloneNotSupportedException`.

WARNING ERRORS DUE TO ERASURE

Just to summarize, in the use of generics in CS2 there is one unavoidable warning errors in an otherwise clean code. The first one is the type cast needed for an array of object to the desirable one, an array of a generic type parameter; the example was seen on the discussion on array types. The second one is the implementations of `equals` or `clone` for a generic class; this one can be avoided using an unbounded wildcard parameterized type as seen above. Code with raw types will produce warning errors; most likely the use of raw types is accidental as is only needed when mixing legacy code and new generic code.

Other Erasure casualties

Here we mention other casualties of type erasure. First, type parameters can't be used in the implements or extends clauses of class definitions. Second, a type must not directly or indirectly derive from two different instantiations of the same generic interface; for instance, a type cannot result from implementing `Comparable<String>` and `Comparable<Number>`, because both of these turn out to be in fact the same interface, `Comparable`, specifying the same `compareTo()` method twice. Third, method overloading using different parameterized types of the same generic type is not legal either. And finally, erasure is also responsible for the lack of construction of type parameter instances as the type is erased; anyway, you can not create an object of generic type because the compiler doesn't know what constructor to call. If your generic class needs to be constructing objects whose type is specified by generic type parameters, its constructors should take a class literal (`MyClass.class`) and store it so that instances can be created via reflection [1,4].

CONCLUSION

The use of type erasure for the implementation of generic types as well as parameterized classes leave these classes in an incomplete state with respect to runtime type information; in fact, erasure makes a hole in the Java's type system. The result is a performance hit due to the added casts and possible bridge method calls performed by the compiler, and lost of type safety. And because of the fact that at runtime the type safety of many operations cannot be fully checked, java's compiler generates warning errors that gives the programmer the option to ignore them at their own peril; they can produce later on a `ClassCastException` when the runtime system performs the cast expected to the type used by the given generic instantiation.

Students should be alerted to the sources of these warnings and the need to sort out each of them for actual elimination to a possible few as mentioned in the paper.

REFERENCES

[1] Gosling J., Joy B., Steel G., Bracha G. *The JavaLanguage Specification*. Third Edition. Addison-Wesley.Budd. 2005.

[2] Langer A. *Java Generics FAQ*. http://www.angelikalanger.com/GenericsFAQ

[3] J.Niño, F. Hosch. *An introduction to programming and Object Oriented design using Java 1.5*. Wiley 2005. 2nd edition.

[4] Sun Corporation. *Java API* for version 1.5 or later.

[5] J.Niño. *Java From The Trenches: Dealing with Object Orientation and Generics*. Submitted to CCSC. Monroe,LA. March 2007.

Appendix

An example of code before type erasure follows:

```
interface Comparable <T> {
 public int compareTo( T that);
}
class NumericValue implements Comparable<NumericValue>{
   private byte value;
   public  NumericValue (byte value) {this.value = value; }
   public  byte getValue() { return value; }
   public  int compareTo( NumericValue that) { return this.value - that.value; }
}//end Numeric Value
class Collections {
 public static <T extends Comparable<? super T>> T max(Collection <T> xs) {
   Iterator <T> xi = xs.iterator();
   T w = xi.next();
   while (xi.hasNext()) {
     T x = xi.next();
     if (w.compareTo(x) < 0) w = x;
   }
   return w;
  }
 }//end Collections
 class Test {
   public static void main (String[] args) {
    LinkedList <NumericValue> numberList = new LinkedList <NumericValue> ();
    numberList.add(new NumericValue((byte)0));
    numberList.add(new NumericValue((byte)1));
    NumericValue y = Collections.max( numberList );
   }
 }
```

Example of the code resulting from type erasure follows.

```
interface Comparable {
  public int compareTo( Object that);
}
class NumericValue implements Comparable {
  private byte value;
  public  NumericValue (byte value) { this.value = value; }
  public  byte getValue() { return value; }
  public  int compareTo( NumericValue that) { return this.value - that.value; }
  public  int compareTo(Object that) { //bridge method added by compiler
   return this.compareTo((NumericValue)that);
  }
}
class Collections {
 public static Comparable max(Collection xs) {
  Iterator xi = xs.iterator();
  Comparable w = (Comparable) xi.next();
  while (xi.hasNext()) {
    Comparable x = (Comparable) xi.next();
    if (w.compareTo(x) < 0) w = x;
  }
  return w;
 }
}
class Test {
 public static void main (String[] args) {
   // same first 3 lines
   NumericValue y = (NumericValue) Collections.max( numberList );
   }
}
```

The generic `Comparable` interface is translated to a non-generic interface and the unbounded type parameter `T` is replaced by `Object`. `NumericValue` class implements the non-generic `Comparable` interface after type erasure; the compiler adds a *bridge method,* `compareTo` with `Object` as parameter, so that `NumericValue` still implements `Comparable`. In `max`, the bounded parameter `T` is replaced by its left most bound, `Comparable`. `Iterator<T>` is translated to `Iterator` and the compiler adds a cast whenever an element is retrieved via the iterator. Uses of `LinkedList<NumericValue>` and generic `max` in the main method are translated to uses of the non-generic type and method and, again, the compiler must add a cast back to `NumericValue` where needed.