Enhanced Type Safety in Java

Swadhin Kumar Barisal
Dept. of CSE, ITER
S'O'A University
Odisha, India

Gayatri Nayak
Dept. of CSE, ITER
S'O'A University
Odisha, India

Bighnaraj Naik
Dept. of IT, ITER
S'O'A University
Odisha, India

ABSTRACT
Java is known to be a strongly type safe language, but there are some coding conventions and when these are used in some applications like persistent storage through serialization may generate unreliable or wrong output. Such cases should be caught and modified as per requirement to produce a modified safe program. This can be achieved by designing a translator tool which can catch unsafe code segments and produce a modified safe code segment. When a singleton class is serialized it is necessary to include a special method from serializable interface within it then only it gives us right result. If this method is not there within the class then it produces unpredictable results. Such results may violate type safe property of object oriented programming. Here the translator is designed using ANTLR which is going to check availability of this method in the input java file. If this method is not found then add the method and generate a type safe program at output.The same translator can be applicable for generics and their limitations. Here the translator is going to trace if there are any unchecked warnings or runtime exception then modify the input program to generate a safe program at output. This will lead to minimize limitations of java generics.

General Terms

Type safety, Design, Reliable data types, object oriented languages

Keywords

Serialization, Java Generics, ANTLR

1. INTRODUCTION

Serialization [12] is used for persistent storage of Java objects to a file, database, network, process or any other system. Serialization flattens objects into serialized stream of bytes. The ordered stream of bytes can then be read at a later time, or in another environment, to recreate the original objects. So if the original object could not be recreated then any operation performed on it may break type-safety.

A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data. That is data should not be modified accidentally or intentional. A Java object may read and modify fields (and invoke methods) private to another object. It may invoke operations not even defined for that object, which is causing completely unpredictable results. Thus Java security depends strongly on type-safety, which cannot be completely compromised.

There are two ways in which the violation of type-safety may be checked. First identify a necessary and sufficient conditions on class loaders such that if all the class loaders definable in a Java program satisfy this condition, then the program will not have any "bridge" classes at run-time, and hence will not exhibit this kind of type-spoofing. Thus one may still informally argue that a particular Java program may not exhibit this kind of type-spoofing, and one may design Java programs in the future to satisfy this condition. Second way is to design some mechanism to catch unsafe code segments from the running application program.

These java generics was introduced in 2004 as java1.5.The main purpose of generics was to give type-safety, avoid repeated type castings [9] and also to support polymorphism of methods and classes. It is also necessary to see if there are any other ways in which type-safety can be compromised in Java.

Some java applications are there which are not properly serializable unless some special care has been taken. Here the 1st example is serialization of singleton classes and user defined EnumType classes. Secondly Java Generics has some limitations where it fails to hold some object oriented properties. Sometimes Generics gives unchecked warnings or runtime class cast exceptions which does not give a clear idea to debug it easily. So this should be more expressive to programmer.

The implementation part shows serialization of singleton classes and user defined EnumType classes where deserialization process fails while reconstructing the objects of such classes. That means our translator will check if such type classes are there in our application and they are implementing java.io.Serializable then that class has to be checked for type-safety.

Secondly part shows introduction to generics and their limitations. Here the translator is going to trace if there are any unchecked warnings or runtime exception then modify the input program to generate a safe program at output.

The second section deals with the prerequisites or concepts required to understand each problem. In third section contains definition of each problem and their solution methodology. Fourth section contains results from experiment. Fifth section contains analysis of results. Sixth part contains conclusion and references.

2. PRILIMINARIES

Serialization [12] is a process of saving the current state of an object to bytes of stream, and restoring the same object from that stream whenever required. This stream representation functions as a container for the object. The byte stream format of the object includes a partial representation of the object's internal structure, including variable types, names, and values. Object to be serialized must be an instance of a class that implements either the Serializable or Externalizable interface.

2.1 Singleton Class

In some applications it may need to create a single instance of a given class. This helps in memory management, and for language like Java, in garbage collection. Single instance is also necessary or desirable for technological or business reasons. For an example; it may be required to create a single instance of a pool of database connections.

For this one better approach is to use static members and methods for the "singleton" functionality. So for these elements, no instance is required. To create singleton class it is necessary to make the class final with a private constructor in order to prevent any instance of the class from being created. The instance can also be made final so that no one can change it at any time during its life time.

```
1 final class Singleton implements Serializable{
2        static Singleton instance = null;
3        private Singleton() {…}
4        public static Singleton getInstance() {
5                if(instance == null) {
                    instance = new   Singleton();
                    }
6         return instance;  }
  }
```

OR

```
1 public final class Singleton implements Serializable{
2 static Singleton instance = new Singleton();
3 private Singleton() {…}
4   }
```

2.2 UserEnums

UserEnum means a user can create a class where its data members can behave exactly as like enum type in current java. This is as an old style declaration. But before java1.5 enum types were created in this way. These UserEnum constants are serialized not in the same way as ordinary serializable or externalizable objects. The serialized form of UserEnum constant consists of fields like its name, field values of the constant. While serializing an enum constant, ObjectOutputStream writes the value returned by the enum constant's name method and while deserilizing an enum constant, ObjectInputStream reads the constant name from the stream then at deserilization time the constant is obtained when the java.lang.Enum.valueOf() method is called by passing the constant's enum type and also the received constant name as arguments.

```
class Move  {
     public static final Move HORIZONTAL = new Move (1);
      public static final Move VERTICAL = new Move (2);
      private int value;

      private Move (int v) { value = v; }
}
```

2.3 Generics

Generics are found in java1.5.Java generics are implemented by generating single byte code representation for all methods and instances. This unique representation is done via type erasure where type information is lost after compilation phase. Because of this several surprising errors rises that have nothing to do with programming. These errors may be found while dealing with operations like subtype checking, subtype assignments [1], method overloading and so on. But Java generics supposed to provide the following benefits, but let us see how far achieved.

- Type safety
- Less explicit casts
- More APIs

Type safety guarantee:

Definition: If your entire application has been compiled without unchecked warnings, it is type safe.

Here type-safety means that there will be no unexpected class cast exception. That is if ClassCastException found then it must be caused by an explicit cast in the code fragment. For this reason implicit type castings are not to be added at compile time of generic code to raise runtime exceptions, because they would be difficult to understand and fix or debug.

```
List rawList = new List();
rawList.add("strings");          // unchecked warning
```

3. PROBLEM FORMULATION

This section defines three problems and their proposed solutions. The solution is provided with an implementation code for the first problem only and rest two problems can be implemented in same fashion.

3.1 Singleton Class Serialization

It is known that during serialization [12], an object is broken into stream of bytes and again build back during Deserialization process. But during Deserialization of Singleton class it forms a new object instead of returning original object. Thus gives unpredictable result. This is happing because of static instance of singleton class. To avoid this add a method called " readResolve()" to this singleton class which creates a bridge between serialization and Deserialization process. That means readResolve() is used for replacing the object read from the stream. So readResolve() method does a single task that is when an object is read, replace it with the singleton instance.

During deserialization, before returning, the jvm checks whether readResolve method is implemented in class whose object is being serialized. If yes, it would invoke the readResolve() method and again checks if it returns true, then the same instance would be returned, otherwise the one was desterilized would be returned. So readResolve() acts as a plugin method between serialization and Deserialization process.

3.2 UserEnum Serialization

Usually enum constants are static and final objects of enum class. So before the introduction of enum keyword in java, people used to declare enum constants using normal class having static final members. Therefore while serializing this enum class it gives unpredictable result due to its static reference members.

The solution to this problem is to use "readResolve()" method .This method acts as a connecting bridge between serialization and deserialization process to return actual object. The same translator can be used to add readresolve() method if not found in the class that implements serialization.

```
class Move implements Serializable {
      public static final Move HORIZONTAL = new Move (1);
      public static final Move VERTICAL = new Move (2);
```

```
    private int value;
    private Move (int v) { value = v; }
private Object readResolve() throws ObjectStreamException {
      if (value == 1) return Move.HORIZONTAL;
      if (value == 2) return Move .VERTICAL;
      return null;
    }
  }
```

3.3 Generic Limitations

3.3.1 Type erasure limitations

Java generics are implemented by using **type erasure** property [10], in which generic type parameters are simply removed during compilation, which means type information is not available to the JVM at runtime. Therefore serialization property cannot be used by the user in case of generic application programs.

2. Java generics also having a significant second Problem as it degrade run-time performance severely. The reason behind this is that the same block of code must work for all supported type parameter instances [6]. In Java generics all type parameter are converted to data as Object class references[7]. This means that all data going into storage must be up casted to Object. This is a process that has almost no run-time overhead. But when the same data is returned, it has to be downcast to its own type that is converted from an Object reference back to a reference to the correct type. This leads to a much more significant run-time overhead.

3. The third problem says, if a generic class has static data, then each generic type cannot have its own copy of that static data, because Java keeps a single copy of the static data for all generic types.

Sometimes these type erasure problems [10] (i.e. warning or runtime classcastExceptions) can be suppressed by applying reflection to generics. That is reflection can store data types before generics loose type information during compilation.

Case1:

Method overloading not permitted if return type is same. In java1.5 method overloading is not supported with generics but java1.6 has. Still if return type is same in all method signatures then java1.6 will show errors.

Code:

```
public class TestErasure {
 public static Object method(List<Object>  list_o) {
    System.out.println("method(List<Object>          list-of -
objects)");
    return null;
 }
 public static Object method(List<String>   list_str) {
    System.out.println("method(List<String>              list-of-
strings)");
    return null;
 }
public static void main(String[] args) {
    method(new ArrayList<Object>());
    method(new ArrayList<String>());
 }
}
```

Output: error: Method name clash "method(ArrayList<>)"

Solution: To overcome this, add extra dummy arguments in method signatures where this dummy argument is nothing to do with overloading. This can be done by this translator.

Case2: Generic at runtime

Due to erasure behavior of Generics in java, there is no sense in checking generic information at runtime. That means instanceof operator cannot be applied for generics. This can be verified by looking in to the following example:

```
if (list instanceof List<String>){ ....} //Illegal
```

Case3: Static data members

Java Generic classes cannot have references to their type parameters from their static data members or methods. More over it can be said that static types are not allowed in generics. Following example says how this is illegal:

```
public TestClass<T>{
   private static T value; //Illegal
   public static void test_static(List<T> list_static){ //Illegal
      ....
   }
}
```

Case4: **Generic exceptions**

It is not allowed to define a generic type that extends some Throwable class. The programmer may wish to handle different parameterized versions of the same exception at runtime.

The problem is that erasure does not allow it. JVM cannot distinguish at runtime between Type < Double > and Type <Integer>. They are both simply Type at runtime. Therefore there is no sense in having generic exceptions.

```
try{
   ....
}catch(Type<Float>){
   ....
}catch(Type<Integer>){
   ....
}
```

Case5: **Generic enums**

Defining generic enums is not allowed: The reason for this to be illegal is related to the static members limitation.

```
enum Direction <T>{ //Illegal
   EAST,WEST,NORTH,SOUTH;
    private T attribute;
    ……
    ……
      }
}
```

3.3.2 Bounded wildcard limitations

Wildcards [3][4]are mainly used to suppress unchecked warnings and for enhancement of type safe generics. Wildcards are using two keywords extends and super [8] to give bounds for data types. Let us see some examples that describe some sorts of ambiguity or do not give clear idea for debugging.

Case1: Let's assume there is a method that manipulates a list of numbers as follows.

```
public void TestWC(List<Number> list_num){
  ....
```

}

The problem here is that List<Integer> cannot be passed as parameter. If it is needed to pass lists of subclasses of Number, then use bounds as follows:

public <T extends Number>void TestWC(List<T> list_num){
....
}

So now it is easy to pass List<Integer>, List<Float> and so on, but a new limitation is found. That is in last method one can call *list_num*.add(new Integer(50)) or *list_num*.add(new Float(5.2)), but for such bounded wildcard the compiler will raise error, because it may not maintain the type homogeneousness in the list. This means add Integers and float types which may result violation of the type safety guarantee. So this says that introducing wildcards do not make java strongly type safe.

4. IMPLEMENTATION

The solution to first problem is implemented using ANTLR. ANTLR is called Another Tool for Language Recognition. ANTLR is a scanner and parser generator. Thus according to our assumption it can scan the input java file and generate a modified java file at output.

This ANTLR is run on a grammar file called java.g. That is use java as our platform. The grammar file is written according to java syntax conventions. Java.g file contains grammar statements for both scanner(i.e. Lexer) and parser(i.e Yacc). AntlrWorks is a GUI based tool as shown in Fig1,which is having different grafical windows for output,parse tree,stack trace and so on.

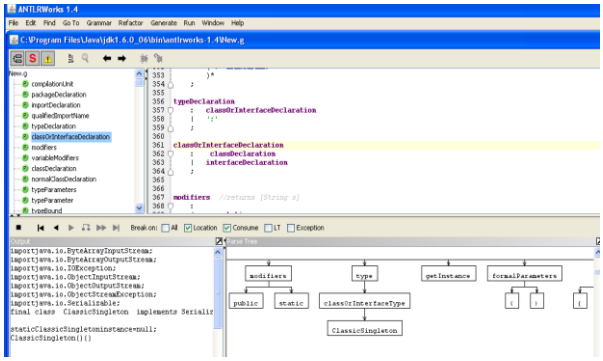

Fig 1: GUI of ANTLR

4.1 Serialization problem

The serialization problem is solved by checking availability of "readResolve()" method in the class that implements serialization from input file. If present just echo the same file to output otherwise add the method to the class that implements serialization and redirect to output which is shown in Fig3.

4.1.1 Singleton serialization

Here two sample codes are given that shows how that is modified and directed to output of ANTLR. The logic behind this implementation is to search the "readResolve()" in the class that implements serialization from input java file as in Fig2. A singleton class has only one instance, so it's very simple to write readResolve() method returning only instance name.

Input: Singleton.java

```
final class Singleton implements Serializable {
        static final long serialVersionUID = 7L;
        private Singleton() { }
       static final Singleton INSTANCE = new Singleton();
 public static void main(String args[]) throws IOException,
ClassNotFoundException{
 ByteArrayOutputStream bout = new ByteArrayOutputStream
(); ObjectOutputStream out = new ObjectOutputStream
(bout);
      Singleton e1 = Singleton.INSTANCE;
       out.writeObject(e1);
       out.flush ();
       ByteArrayInputStream bin = new ByteArrayInputStream
(bout.toByteArray ());
       ObjectInputStream in = new ObjectInputStream (bin);
       Singleton e2 = (Singleton) in.readObject ();
       System.out.println ((e2.equals(Singleton.INSTANCE)));
              }
}
```

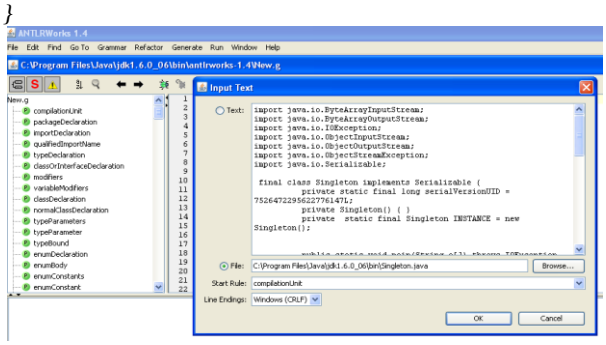

Fig 2: Input file for ANTLR

Output:

```
final class Singleton implements Serializable {

        static final long serialVersionUID = 7L;

        private Singleton() { }

        static final Singleton INSTANCE = new Singleton();

 public static void main(String a[]) throws IOException,
ClassNotFoundException{

 ByteArrayOutputStream bout = new ByteArrayOutputStream
(); ObjectOutputStream out = new ObjectOutputStream
(bout);

      Singleton e1 = Singleton.INSTANCE;

        out.writeObject(e1);

        out.flush ();

       ByteArrayInputStream bin = new ByteArrayInputStream
(bout.toByteArray ());

       ObjectInputStream in = new ObjectInputStream (bin);

       Singleton e2 = (Singleton) in.readObject ();
       System.out.println ((e2.equals(Singleton.INSTANCE)));
              }
private Object readResolve() throws ObjectStreamException {
                        return INSTANCE;
              }
}
```

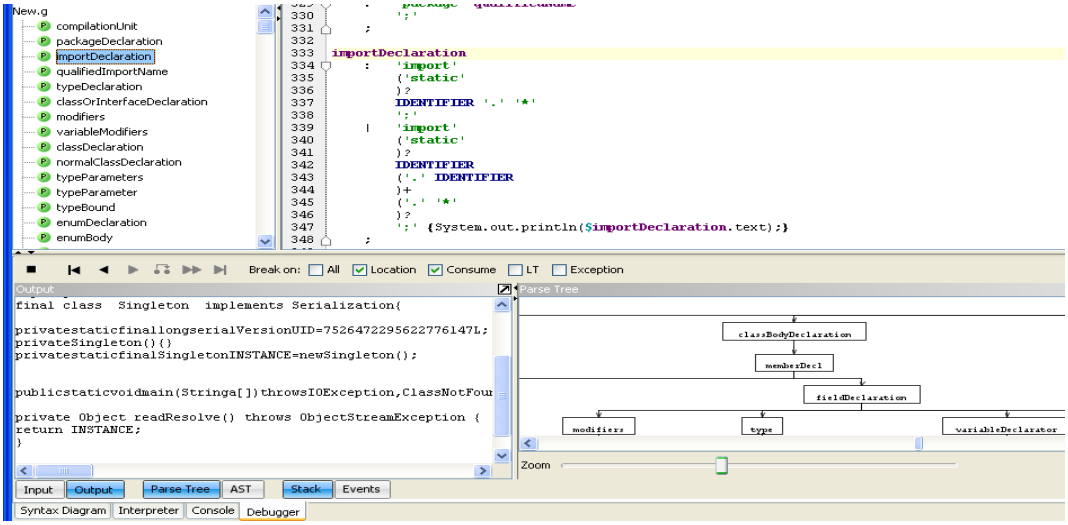

FIG 3: OUTPUT FILE FOR ANTLR

4.1.2 UserEnum serialization

In this case it is shown that how to implement readResolve() method and add this after scanning input java file. The logic is to scan how may enum constants are there in the UserEnum class then write a switch case having exactly same number of cases returning their corresponding instances. That is case1 is to return 1st instance of enum and so on.

Input:

class Move implements Serializable {
public static final Move HORIZONTAL = new Move (1);
public static final Move VERTICAL = new Move (2);
private int value;

private Move (int v)

{ value = v; }

}

Output:

class Move implements Serializable {

public static final Move HORIZONTAL = new Move (1);

public static final Move VERTICAL = new Move (2);

private int value;

private Move (int v) { value = v; }

protected Object readResolve() throws ObjectStreamException { if (value == 1) return Move.HORIZONTAL;

if (value == 2) return Move.VERTICAL;

return null;

}

}

5. RESULTS AND ANALYSIS

In case of serialization problem it works properly and gives us correct output. The translator is now capable enough to add "readResolve()" method in the class that implements serialization. This has been implemented for singleton class and working properly. In the same way readResolve() method can be designed and put in to UserEnum class.

Talking about java generics for type erasure problems, reflection can be used to keep track of type information. For generics, it can be said that due to type erasure and bounded wildcards, generics are not much more expressive to the programmer. It looks puzzle and ambiguous to the programmer while implementing generic applications. Sometimes this type erasure problems (i.e. warning or runtime classcastExceptions) can be suppressed by applying reflection to generics. That is reflection can store data types before generics loose type information during compilation.

6. CONCLUSION

To sum up, this translator is a better than Lex-Yacc and its performance is also better and more user friendly. It helps in translating or creating safe java program that are put into serialization process. It automatically catches all serializable classes and particularly serializing singleton classes.

For generics, it can be said that due to type erasure and bounded wildcards, generics are not much more expressive to the programmer. It looks puzzle and ambiguity to the programmer while implementing generic applications. Sometimes this type erasure problems (i.e. warning or runtime classcastExceptions) can be suppressed by applying reflection to generics. Thus generics is said to be complete type safe when it is free from unchecked warnings and run time ClassCastExceptions. The same can be solved by extending the capability of the translator.

7. REFERENCES

[1] Stephanie C. Weirich and Liang Huang. 2005. A Design for Type-Directed Programming in Java .ELSEVIER.

[2] Renaud Pawlak. Carlos Noguera and Nicolas Petitprez. May-2006. *A* Systèmes communicants. Technical Report.

[3] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ah´, Gilad Bracha and Neal Gafter.2004. Adding Wildcard to java programming Language. In *Journal of object oriented programming* . Published by ETH Zurich, New York.

[4] Maurizio Cimadamore, Mirko Viroli . 16 july 2008. On the reification of Java wildcards. ELSEVIER.

[5] Nabilel Boustani and Jurriaan Hage. 2003. Improving Type Error Messages for Generic Java . ACM Press.

[6] Eric E. Allena and Robert Cartwrightb. 2006. Safe instantiation in Generic Java. ELSEVIER.

[7] Bruno De Fraine.july 2009. Range Parameterized Types:Use-site Variance without the Existential Questions. Genova Italy.

[8] Halm Reusser and Peter Sommerlad. 2009. Refactoring towards Java generics.

[9] Kimb B. Bruce, Angela Schuett and Robert Van Gent.2003. A Type-Safe Polymorphic object oriented language. ACM Transactions on program languages and Systems .

[10] Jaime Nino. May 2007. The Cost of Erasure in Java Generics Type System. ACM Transactions.

[11] Ole Agesen, Stephen N. reund and John C. itchell.1997.Adding Type Parameterization to the Java Language. ACM Conf.

[12] David Willians, RanchHand.2009. Serialization and Deserialization of java enums.