Java SE Documentation

Improved Compiler Warnings and Errors When Using Non-Reifiable Formal Parameters with Varargs Methods

This page covers the following topics:

Heap Pollution

Most parameterized types, such as `ArrayList<Number>` and `List<String>`, are *non-reifiable types*. A non-reifiable type is a type that is *not* completely available at runtime. At compile time, non-reifiable types undergo a process called type erasure during which the compiler removes information related to type parameters and type arguments. This ensures binary compatibility with Java libraries and applications that were created before generics. Because type erasure removes information from parameterized types at compile-time, these types are non-reifiable.

Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation can only occur if the program performed some operation that would give rise to an unchecked warning at compile-time. An *unchecked warning* is generated if, either at compile-time (within the limits of the compile-time type checking rules) or at runtime, the correctness of an operation involving a parameterized type (for example, a cast or method call) cannot be verified.

Consider the following example:

```
    List l = new ArrayList<Number>();
    List<String> ls = l;       // unchecked warning
    l.add(0, new Integer(42)); // another unchecked warning
    String s = ls.get(0);      // ClassCastException is thrown
```

During type erasure, the types `ArrayList<Number>` and `List<String>` become `ArrayList` and `List`, respectively.

The variable `ls` has the parameterized type `List<String>`. When the `List` referenced by `l` is assigned to `ls`, the compiler generates an unchecked warning; the compiler is unable to determine at compile time, and moreover knows that the JVM will not be able to determine at runtime, if `l` refers to a `List<String>` type; it does not. Consequently, heap pollution occurs.

As a result, at compile time, the compiler generates another unchecked warning at the `add` statement. The compiler is unable to determine if the variable `l` refers to a `List<String>` type or a `List<Integer>` type (and another heap pollution situation occurs). However, the compiler does not generate a warning or error at the `get` statement. This statement is valid; it is calling the `List<String>.get` method to retrieve a `String` object. Instead, at runtime, the `get` statement throws a `ClassCastException`.

In detail, a heap pollution situation occurs when the `List` object `l`, whose static type is `List<Number>`, is assigned to another `List` object, `ls`, that has a different static type, `List<String>`. However, the compiler still allows this assignment. It must allow this assignment to preserve backwards compatibility with versions of Java SE that do not support generics. Because of type erasure, `List<Number>` and `List<String>` both become `List`. Consequently, the compiler allows the assignment of the object `l`, which has a raw type of `List`, to the object `ls`.

Furthermore, a heap pollution situation occurs when the `l.add` method is called. The static type second formal parameter of the `add` method is `String`, but this method is called with an actual parameter of a different type, `Integer`. However, the compiler still allows this method call. Because of type erasure, the type of the second formal parameter of the `add` method (which is defined as `List<E>.add(int,E)`) becomes `Object`. Consequently, the compiler allows this method call because, after type erasure, the `l.add` method can add any object of type `Object`, including an object of `Integer` type, which is a subtype of `Object`.

Variable Arguments Methods and Non-Reifiable Formal Parameters

Consider the method `ArrayBuilder.addToList` in the following example. It is a variable arguments (also known as varargs) method that adds the objects of type `T` contained in the `elements` varargs formal parameter to the `List listArg`:

```
import java.util.*;

public class ArrayBuilder {

  public static <T> void addToList (List<T> listArg, T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }

  public static void faultyMethod(List<String>... l) {
    Object[] objectArray = l;  // Valid
    objectArray[0] = Arrays.asList(new Integer(42));
    String s = l[0].get(0);    // ClassCastException thrown here
  }

}
```

```
import java.util.*;

public class HeapPollutionExample {

  public static void main(String[] args) {
    List<String> stringListA = new ArrayList<String>();
    List<String> stringListB = new ArrayList<String>();

    ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");
    ArrayBuilder.addToList(stringListA, "Ten", "Eleven", "Twelve");
    List<List<String>> listOfStringLists = new ArrayList<List<String>>();
    ArrayBuilder.addToList(listOfStringLists, stringListA, stringListB);

    ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
  }
}
```

The Java SE 7 compiler generates the following warning for the definition of the method `ArrayBuilder.addToList`:

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method `ArrayBuilder.addToList`, the compiler translates the varargs formal parameter `T... elements` to the formal parameter `T[] elements`, an array. However, because of type erasure, the compiler converts the varargs formal parameter to `Object[] elements`. Consequently, there is a possibility of heap pollution. See the next section, Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters, for more information.

Note: The Java SE 5 and 6 compilers generate this warning when the `ArrayBuilder.addToList` is called; in this example, the warning is generated for the class `HeapPollutionExample`. These compilers do not generate the warning at the declaration site. However, the Java SE 7 generates the warning at both the declaration site and the call site (unless the warnings are preempted with annotations; see Suppressing Warnings from Varargs Methods with Non-Reifiable Formal Parameters for more information). The advantage of generating a warning when a compiler encounters a varargs method that has a non-reifiable varargs formal parameter at the declaration site as opposed to the call site is that there is only one declaration site; there are potentially many call sites.

Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters

The method `ArrayBuilder.faultyMethod` shows why the compiler warns you about these kinds of methods. The first statement of this method assigns the varargs formal parameter `l` to the `Object` array `objectArgs`:

```
    Object[] objectArray = l;
```

This statement can potentially introduce heap pollution. A value that does match the parameterized type of the varargs formal parameter `l` can be assigned to the variable `objectArray`, and thus can be assigned to `l`. However, the compiler does not generate an unchecked warning at this statement. The compiler has already generated a warning when it translated the varargs formal parameter `List<String>... l` to the formal parameter `List[] l`. This statement is valid; the variable `l` has the type `List[]`, which is a subtype of `Object[]`.

Consequently, the compiler does not issue a warning or error if you assign a `List` object of any type to any array component of the `objectArray` array as shown by this statement:

```
    objectArray[0] = Arrays.asList(new Integer(42));
```

This statement assigns to the first array component of the `objectArray` array with a `List` object that contains one object of type `Integer`.

Suppose you call the `ArrayBuilder.makeArray` method with the following statement:

```
ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
```

At runtime, the JVM throws a `ClassCastException` at the following statement:

```
String s = l[0].get(0);    // ClassCastException thrown here
```

The object stored in the first array component of the variable `l` has the type `List<Integer>`, but this statement is expecting an object of type `List<String>`.

Suppressing Warnings from Varargs Methods with Non-Reifiable Formal Parameters

If you declare a varargs method that has parameterized parameters, and you ensure that the body of the method does not throw a `ClassCastException` or other similar exception due to improper handling of the varargs formal parameter (as shown in the `ArrayBuilder.faultyMethod` method), you can suppress the warning that the compiler generates for these kinds of varargs methods by using one of the following options:

- Add the following annotation to static and non-constructor method declarations:

  ```
  @SafeVarargs
  ```

  Unlike the `@SuppressWarnings` annotation, the `@SafeVarargs` annotation is a documented part of the method's contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.

- Add the following annotation to the method declaration:

  ```
  @SuppressWarnings({"unchecked", "varargs"})
  ```

  Unlike the `@SafeVarargs` annotation, the `@SuppressWarnings("varargs")` does not suppress warnings generated from the method's call site.

- Use the compiler option `-Xlint:varargs`.

For example, the following version of the `ArrayBuilder` class has two additional methods, `addToList2` and `addToList3`:

```
public class ArrayBuilder {

  public static <T> void addToList (List<T> listArg, T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }

  @SuppressWarnings({"unchecked", "varargs"})
```

```
  public static <T> void addToList2 (List<T> listArg, T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }

  @SafeVarargs
  public static <T> void addToList3 (List<T> listArg, T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }

  // ...

}

public class HeapPollutionExample {

  // ...

  public static void main(String[] args) {

    // ...

    ArrayBuilder.addToList(listOfStringLists, stringListA, stringListB);
    ArrayBuilder.addToList2(listOfStringLists, stringListA, stringListB);
    ArrayBuilder.addToList3(listOfStringLists, stringListA, stringListB);

    // ...

  }
}
```

The Java compiler generates the following warnings for this example:

- **`addToList`**:
  - At the method's declaration: `[unchecked] Possible heap pollution from parameterized vararg type T`
  - When the method is called: `[unchecked] unchecked generic array creation for varargs parameter of type List<String>[]`
- **`addToList2`**: When the method is called (no warning is generated at the method's declaration): `[unchecked] unchecked generic array creation for varargs parameter of type List<String>[]`
- **`addToList3`**: No warnings are generated either at the method's declaration or when it is called.

Note: In Java SE 5 and 6, it is the responsibility of the programmer who calls a varargs method that has a non-reifiable varargs formal parameter to determine whether heap pollution would occur. However, if this programmer did not write such a method, he or she cannot easily determine this. In Java SE 7, it is the responsibility of the programmer who *writes* these kinds of varargs methods to ensure that they properly handle the varargs formal parameter and ensure heap pollution does not occur.
