OBJ03-J. Prevent heap pollution

Heap pollution occurs when a variable of a parameterized type references an object that is not of that parameterized type. (For more information on heap pollution, see The Java Language Specification (JLS), §4.12.2, “Variables of Reference Type” [JLS 2015].)

Mixing generically typed code with raw typed code is one common source of heap pollution. Generic types were unavailable prior to Java 5, so popular interfaces such as the Java Collection Framework relied on raw types. Mixing generically typed code with raw typed code allowed developers to preserve compatibility between nongeneric legacy code and newer generic code but also gave rise to heap pollution. Heap pollution can occur if the program performs some operation involving a raw type that would give rise to a compile-time unchecked warning.

When generic and nongeneric types are used together correctly, these warnings can be ignored; at other times, these warnings can denote potentially unsafe operations. Mixing generic and raw types is allowed provided that heap pollution does not occur. For example, consider the following code snippet.

```
List list = new ArrayList();
List<String> ls = list; // Produces unchecked warning
```

In some cases, it is possible that a compile-time unchecked warning will not be generated. According to the JLS, §4.12.2, "Variables of Reference Type" [JLS 2015]:

> Note that this does not imply that heap pollution only occurs if an unchecked warning actually occurred. It is possible to run a program where some of the binaries were compiled by a compiler for an older version of the Java programming language, or by a compiler that allows the unchecked warnings to be suppressed. This practice is unhealthy at best.

Heap pollution can also occur if the program aliases an array variable of non-reifiable element type through an array variable of a supertype that is either raw or nongeneric.

Noncompliant Code Example

This noncompliant code example compiles but results in heap pollution. The compiler produces an unchecked warning because a raw argument (the `obj` parameter in the `addToList()` method) is passed to the `List.add()` method.

```
class ListUtility {
    private static void addToList(List list, Object obj) {
        list.add(obj); // Unchecked warning
    }

global static void main(String[] args) {
    List<String> list = new ArrayList<String> ();
    addToList(list, 42);
    System.out.println(list.get(0)); // Throws ClassCastException
}
```

Heap pollution is possible in this case because the parameterized type information is discarded before execution. The call to `addToList(list, 42)` succeeds in adding an integer to `list`, although it is of type `List<String>`. This Java runtime does not throw a `ClassCastException` until the value is read and has an invalid type (an `int` rather than a `String`). In other words, the code throws an exception some time after the execution of the operation that actually caused the error, complicating debugging.

Even when heap pollution occurs, the variable is still guaranteed to refer to a subclass or subinterface of the declared type but is not guaranteed to always refer to a subtype of its declared type. In this example, `list` does not refer to a subtype of its declared type (`List<String>`) but only to the subinterface of the declared type (`List`).

Compliant Solution (Parameterized Collection)

This compliant solution enforces type safety by changing the `addToList()` method signature to enforce proper type checking:
class ListUtility {
    private static void addToList(List<String> list, String str) {
        list.add(str); // No warning generated
    }

    public static void main(String[] args) {
        List<String> list = new ArrayList<String> ();
        addToList(list, "42");
        System.out.println(list.get(0));
    }
}

The compiler prevents insertion of an object to the parameterized list because `addToList()` cannot be called with an argument whose type produces a mismatch. This code has consequently been changed to add a `String` instead of an `int` to the list.

### Compliant Solution (Legacy Code)

The previous compliant solution eliminates use of raw collections, but implementing this solution when interoperating with legacy code may be infeasible.

Suppose that the `addToList()` method is legacy code that cannot be changed. The following compliant solution creates a checked view of the list by using the `Collections.checkedList()` method. This method returns a wrapper collection that performs runtime type checking in its implementation of the `add()` method before delegating to the back-end `List<String>`. The wrapper collection can be safely passed to the legacy `addToList()` method.

```java
class ListUtility {
    private static void addToList(List list, Object obj) {
        list.add(obj); // Unchecked warning, also throws ClassCastException
    }

    public static void main(String[] args) {
        List<String> list = new ArrayList<String> ();
        List<String> checkedList = Collections.checkedList(list, String.class);
        addToList(checkedList, 42);
        System.out.println(list.get(0));
    }
}
```

The compiler still issues the unchecked warning, which may still be ignored. However, the code now fails when it attempts to add the integer to the list, consequently preventing the program from proceeding with invalid data.

### Noncompliant Code Example

This noncompliant code example compiles and runs cleanly because it suppresses the unchecked warning produced by the raw `List.add()` method. The `printNum()` method intends to print the value 42, either as an `int` or as a `double` depending on the type of the variable `type`. 

```java
class ListUtility {
    private static void addToList(List<String> list, String str) {
        list.add(str); // No warning generated
    }

    public static void main(String[] args) {
        List<String> list = new ArrayList<String> ();
        addToList(list, "42");
        System.out.println(list.get(0));
    }
}
```
class ListAdder {
    @SuppressWarnings("unchecked")
    private static void addToList(List list, Object obj) {
        list.add(obj); // Unchecked warning suppressed
    }

    private static <T> void printNum(T type) {
        if (!(type instanceof Integer || type instanceof Double)) {
            System.out.println("Cannot print in the supplied type");
        }
        List<T> list = new ArrayList<T>();
        addToList(list, 42);
        System.out.println(list.get(0));
    }

    public static void main(String[] args) {
        double d = 42;
        int i = 42;
        System.out.println(d);
        ListAdder.printNum(d);
        System.out.println(i);
        ListAdder.printNum(i);
    }
}

However, despite list being correctly parameterized, this method prints 42 and never 42.0 because the int value 42 is always added to list without being type checked. This code produces the following output:

42.0
42
42
42

Compliant Solution (Parameterized Collection)

This compliant solution generifies the addToList() method, eliminating any possible type violations:
class ListAdder {
    private static <T> void addToList(List<T> list, T t) {
        list.add(t);     // No warning generated
    }

    private static <T> void printNum(T type) {
        if (type instanceof Integer) {
            List<Integer> list = new ArrayList<Integer>();
            addToList(list, 42);
            System.out.println(list.get(0));
        } else if (type instanceof Double) {
            List<Double> list = new ArrayList<Double>();
            addToList(list, 42.0);  // Will not compile with 42 instead of 42.0
            System.out.println(list.get(0));
        } else {
            System.out.println("Cannot print in the supplied type");
        }
    }

    public static void main(String[] args) {
        double d = 42;
        int i = 42;
        System.out.println(d);
        ListAdder.printNum(d);
        System.out.println(i);
        ListAdder.printNum(i);
    }
}

This code compiles cleanly and produces the correct output:

42.0
42.0
42
42

If the method addToList() is externally defined (such as in a library or as an upcall method) and cannot be changed, the same compliant method printNum() can be used, but no warnings result if addToList(list, 42) is used instead of addToList(list, 42.0). Great care must be taken to ensure type safety when generics are mixed with nongeneric code.

Noncompliant Code Example (Variadic Arguments)

Heap pollution can occur without using raw types such as java.util.List. This noncompliant code example builds a list of lists of strings before passing it to a modify() method. Because this method is variadic, it casts list into an array of lists of strings. But Java is incapable of representing the types of parameterized arrays. This limitation allows the modify() method to sneak a single integer into the list. Although the Java compiler emits several warnings, this program compiles and runs until it tries to extract the integer 42 from a List<String>.
class ListModifierExample {
    public static void modify(List<String>... list) {
        Object[] objectArray = list;
        objectArray[1] = Arrays.asList(42);  // Pollutes list, no warning

        for (List<String> ls : list) {
            for (String string : ls) {  // ClassCastException on 42
                System.out.println(string);
            }
        }
    }

    public static void main(String[] args) {
        List<String> s = Arrays.asList("foo", "bar");
        List<String> s2 = Arrays.asList("baz", "quux");
        modify(s, s2);  // Unchecked varargs warning
    }
}

This program produces the following output:

```
foo
bar
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
    at ListModifierExample.modify(ListModifierExample.java:13)
    at ListModifierExample.main(ListModifierExample.java:25)
    at Java.main(Java.java:33)
```

Noncompliant Code Example (Array of Lists of Strings)

This noncompliant code example is similar, but it uses an explicit array of lists of strings as the single parameter to `modify()`. The program again dies with a `ClassCastException` from the integer 42 injected into a list of strings.

class ListModifierExample {
    public static void modify(List<String>[] list) {
        Object[] objectArray = list;  // Valid
        objectArray[1] = Arrays.asList(42);  // Pollutes list, no warning

        for (List<String> ls : list) {
            for (String string : ls) {  // ClassCastException on 42
                System.out.println(string);
            }
        }
    }

    public static void main(String[] args) {
        List<String> s = Arrays.asList("foo", "bar");
        List<String> s2 = Arrays.asList("baz", "quux");
        List list[] = {s, s2};  // Unchecked conversion warning
        modify(list);
    }
}

Compliant Solution (List of Lists of Strings)

This compliant solution uses a list of lists of strings as the argument to `modify()`. This type safety enables the compiler to prevent the `modify()` method from injecting an integer into the list. In order to compile, the `modify()` method instead inserts a string, preventing heap pollution.
class ListModifierExample {
    public static void modify(List<List<String>> list) {
        list.set(1, Arrays.asList("forty-two")); // No warning
        for (List<String> ls : list) {
            for (String string : ls) { // ClassCastException on 42
                System.out.println(string);
            }
        }
    }
    
    public static void main(String[] args) {
        List<String> s = Arrays.asList("foo", "bar");
        List<String> s2 = Arrays.asList("baz", "quux");
        List<List<String>> list = new ArrayList<List<String>>();
        list.add(s);
        list.add(s2);
        modify(list);
    }
}

Note that to avoid warnings, we cannot use `Arrays.asList()` to build a list of lists of strings because that method is also variadic and would produce a warning about variadic arguments being parameterized class objects.

Risk Assessment

Mixing generic and nongeneric code can produce unexpected results and exceptional conditions.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJ03-J</td>
<td>Low</td>
<td>Probable</td>
<td>Medium</td>
<td>P4</td>
<td>L3</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parasoft Jtest</td>
<td>2022.1</td>
<td>CERT.OBJ03.AGBPT</td>
<td>Avoid conversions from parameterized types to raw types</td>
</tr>
</tbody>
</table>

Bibliography

[Bloch 2008] Item 23, "Don't Use Raw Types in New Code"
[Bloch 2007]
[Bloch 2005] Puzzle 88, "Raw Deal"
[Darwin 2004] Section 8.3, "Avoid Casting by Using Generics"
[JavaGenerics 2004]
[Java Tutorials] "Heap Pollution"
[JLS 2015] §4.9, "Raw Types"
[§4.12.2, "Variables of Reference Type"
[Chapter 5, "Conversions and Promotions"
[§5.1.9, "Unchecked Conversion"
[Langer 2008] Topic 3, "Coping with Legacy"
[Naftalin 2006] Chapter 8, "Effective Generics"
[Naftalin 2006b] "Principle of Indecent Exposure"
[Schildt 2007] "Create a Checked Collection"