Extented Java Typechecking with Checker Framework

Software Analysis - Assignment 2

Luca Di Bello

April 23, 2024

Contents

1 Introduction

This assignment aims to extend the Java type system with the *Checker Framework*[1]. The *Checker Framework* is a powerful tool that integrates with the Java compiler to detect bugs and verify their absence at compile time. This is done by pluggable type-checkers, which via explicit annotations in the code, are able to check for a wide range of errors, such as null pointer dereferences, type casts, and array bounds. It includes over 20 type-checkers, which can be used to verify a wide range of properties. Some of the most useful type-checkers include:

- **Nullness Checker**: Prevent any `NullPointerException` by ensuring that variables are not null when dereferenced.
- **Index Checker**: Prevents array index out-of-bounds errors by ensuring that array accesses are always within bounds.
- **Regex Checker**: Prevents runtime exceptions due to invalid regular expressions by checking the syntax of regular expressions at compile time.

Note: a full list of available checkers can be found on the Checker Framework manual in the `Introduction` section [1]

[1] `https://checkerframework.org/`

To showcase the effectiveness of this tool, the *Checker Framework's Index Checker* has been integrated into an existing codebase to prevent any out-of-bounds access to arrays. The project is a legacy Java library named `RxRelay`[2], a small library that aims to extend the capabilities of a famous Java library called `RxJava` by providing a set of `Relay` classes that act as both an `Observable` and a `Consumer`. The library is used to relay events from one component to another, and it is widely used in Android applications. [2] The library is composed of a single package, `com.jakewharton.rxrelay3`, and it contains a total of 6 Java files:

- `Relay.java`: An interface that extends both RxJava's `Observable` and `Consumer` interfaces. This interface is implemented by all the relay classes to provide a general API for relaying events.
- `SerializedRelay.java`: An abstract class that implements the `Relay` interface and provides a thread-safe implementation of the relay logic. This class is used as a base class for all the concrete relay implementations.
- `AppendOnlyLinkedList.java`: An unconventional unbounded linked list implementation that is used by several relay classes to store events. Rather than using a traditional Node-based linked list, this implementation uses a single array, which is then expanded when it reaches its capacity. This is done using a clever trick that allows the array to be expanded without the need to copy the elements from the old array to the new one.
- `PublishRelay.java`: A concrete implementation of the `Relay` interface that relays events to all the subscribers that are currently subscribed to it.
- `ReplayRelay.java`: Another `Relay` implementation that relays events to all the subscribers that are currently subscribed to it, but it also caches a certain number of events and replays them to new subscribers when they subscribe.
- `BehaviorRelay.java`: A `PublishRelay` implementation that relays only the most recent event to new subscribers. This is done by caching the most recent event and replaying it to new subscribers when they subscribe.

[2] `https://github.com/JakeWharton/RxRelay`

2 Project setup

The project uses Java 8 and uses *Maven* as a build system. The *Checker Framework* is integrated into the project using the `checker-qual` dependency, which provides the necessary annotations to use the checkers. The `maven-compiler-plugin` has been configured to use the *Checker Framework* as annotation processor and to Google's `Error Prone` as a compiler plugin to provide additional static analysis checks.

Additional compiler flags have been added to the `maven-compiler-plugin` configuration to enable additional features of the framework. The configuration is as follows:

- `-Xmaxerrs 10000`: Set the maximum number of errors to display before stopping the compilation process.
- `-Xmaxwarns 10000`: Similar to `-Xmaxerrs`, but for warnings.
- `-Awarns`: Show *Checker Framework* errors as warnings instead of errors to allow the compilation process to continue even if errors are found.
- `-AresolveReflection`: Enable the reflection resolver, which is used to resolve reflection calls at compile time. This is useful to infer the type of reflection calls such as `Array.newInstance` or `Class.forName`.
- `-ArequirePrefixInWarningSuppressions`: Require the `SuppressWarnings` annotation to have a prefix that matches the checker name. This is useful to prevent accidental suppression of warnings from other checkers. For example, to suppress a warning from the `Index Checker`, the annotation should formatted as `SuppressWarnings("index:<specific_error_to_suppress>")`.
- `-AassumeAssertionsAreEnabled`: This flag enables the framework to infer additional information about the assertions made in the code. This is useful to help the framework understand complex control flows.

A small `Makefile` along with some scripts have been provided to simplify the testing and compilation of the project. The `Makefile` contains the following targets:

- `setup`: Setup the project by setting the right Java version and installing the necessary dependencies using *Maven*.
- `build`: Build the project using *Maven*.
- `compile`: Compile the project using *Maven*.
- `test`: Run unit tests and mock tests using *Maven*.

3 Integration of the Index Checker: Challenges and Solutions

The *Index Checker* has been successfully integrated into the project by adding the necessary annotations to the codebase to ensure that all array accesses are within bounds. At first, the checker was able to find several errors, which were then fixed one by one using the provided set of annotations.

Certain Java idioms used in the project were not supported out-of-the-box by the checker as they would require reflection to be resolved. To address this issue, the `-AresolveReflection` flag was enabled to allow *Checker Framework* to resolve reflection calls at compile time. Without this flag, the checker would not be able to infer the type of reflection calls such as `Array.newInstance` or `Class.forName`, used several times in the project specifically in the `AppendOnlyLinkedList` class to expand the array when it reaches its capacity.

Unfortunately, even after enabling the reflection resolver, the checker was unable to resolve certain reflection calls, specifically the ones related to creating new arrays using `Array.newInstance`. For example, the following code snippet was not correctly inferred by the checker:

```
if (array.length < s) {
    // create a new array of size s
    array = (T[]) Array.newInstance(
      array.getClass().getComponentType(),
      s
    );
}
```

Listing 1: Reflection call to create a new array

To fix this issue, the checker was instructed to leverage assertions to gather additional information about the code. This was done by enabling the `-AassumeAssertionsAreEnabled` flag. Along with this, as explicitly advised in the documentation of the checker, each assertion is accompanied by a comment that explains what checkers it is intended to help, with a brief description of why the assertion was needed. The code before has been modified as follows:

```
if (array.length < s) {
    // create a new array of size s
    array = (T[]) Array.newInstance(
      array.getClass().getComponentType(),
      s
    );

    // As the CheckerFramework is not able to infer that the array
    // is of length s, we need to make an assertion to help it
    assert array.length == s: "@AssumeAssertion(index): The array is
       exactly of length s, as we have just created it";
}
```

Listing 2: Using assertions to help the Index Checker infer the array length

As the project uses many unconventional idioms which leverage reflection and particularities of the Java language, the use of assertions was crucial to fix the various array access issues found by the *Index Checker*.

An example of such idioms can be found in the `AppendOnlyLinkedArrayList` class, where the a generic unbounded linked list has been implemented using arrays rather than traditional linked nodes. This class represent one of the core components in the library, and also one of the most difficult to annotate due to its unconventional implementation.

The `AppendOnlyLinkedArrayList` generic class takes a type parameter `T` which defines the type of the elements stored in the list and, an initial non-negative capacity which defines the initial size of the list. In Listing 3 is possible to see the annotated class fields and the constructor.

```
/**
* A linked-array-list implementation that only supports appending
   and consumption.
*
* @param <T> the value type
*/
class AppendOnlyLinkedArrayList<T> {
    // Initial capacity of the list
    private final @NonNegative @LTLengthOf({"head", "tail"}) int
       capacity;
    // Reference to first object of array
    private final @MinLen(1) Object[] head;
    // Reference to the last object of the array
    private @MinLen(1) Object[] tail;

    // Position of the first empty spot in the array (by
       construction, could be at most equal to capacity)
    private @NonNegative @LTLengthOf("tail") int offset;

    /**
    * Constructs an empty list with a per-link capacity.
    * @param capacity the capacity of each link
    */
    AppendOnlyLinkedArrayList(@NonNegative int capacity) {
        this.head = new Object[capacity + 1];
        this.tail = head;
        this.capacity = capacity;
    }

    ...
}
```

Listing 3: AppendOnlyLinkedArrayList

From the *Checker Framework's Index Checker* annotations present in the code, it is possible to grasp details about the implementation of such linked list: the `head` is the reference to the array that stores the objects of type `T`, and `tail` (at start) references the same array as `head`. From the class fields, it is also possible to notice the presence an additional variable `offset` (initialized at 0 by default) which represent the position of the first empty slot in the array where new elements can be added.

The class offers a method `add(T value)`, useful to add elements of type T to the list. When the list reaches its capacity, the array is automatically expanded using a clever trick: rather than instantiating a new array with increased size and copying iteratively the elements from the old array to the new one, the method creates a new array with the same initial capacity and assigns it as last

element of the array (it requires an unchecked cast from type `Object[]` to type `Object`). Right after, it will update the `tail` variable to point to the new array, and resets the `offset` to 0 so that new elements can be added to the new array in the right slots. The code for this operation is shown in Listing 4.

```
@SuppressWarnings("value:assignment") // Suppress he warning about
    the unchecked cast
void add(T value) {
    // Save locally capacity of the array and the current offset (
        index to save next element)
    final @NonNegative @LTLengthOf("tail") int c = capacity;
    // Get the index of the last element
    @NonNegative @LTLengthOf("tail") int o = offset;
    // Check if o is higher than the capacity of the array (not a
        valid index)
    if (o == c) {
        // If the capacity is reached, we create an array of size c
            +1 (we double the size)
        @MinLen(1) Object[] next = new Object[c + 1];

        // Assign the newly created array to the last element of the
            current array
        //   (this will create a warning as we are casting Object[]
            to Object)
        tail[c] = next;

        // We move the tail on the new array, meaning we are now
            working on the newly created array
        // This effectively doubles the size of the linked list!
        tail = next;

        // Reset the offset to 0 in order to use the new array in
            its entirely
        o = 0;
    }

    // The offset is now guaranteed to be less than the capacity of
        the array!
    assert o < c: "@AssumeAssertion(index): CheckerFramework is not
        able to understand that o is always less than c";

    // Save the value at the current offset
    tail[o] = value;
    // Increase the offset to the next slot
    offset = o + 1;
}
```

Listing 4: Expanding the array in the AppendOnlyLinkedArrayList class

This implementation is used by several classes in the project, and it is crucial to the library's functionality.

Apart from the `AppendOnlyLinkedArrayList` class, the other classes were annotated without major issues: the `PublishRelay`, `ReplayRelay`, and `BehaviorRelay` classes have been annotated correctly, enabling the *Checker Framework* to fully verify the absence of out-of-bounds array accesses in the project.

4 Conclusions

The integration of the *Checker Framework's Index Checker* into the `RxRelay` project has been successful, and the checker has been able to detect several out-of-bounds array accesses in the codebase. By using the provided annotations and leveraging assertions, it has been possible to fix all the issues found by the checker thus ensuring that all array accesses are within bounds.

From this experience, it is clear that the *Checker Framework* is a powerful tool that, if integrated at early stages of development, can help prevent to introduge bugs and vulnerabilities in the codebase.

References

[1] J. Wharton. Chekerframework - manual. `https://checkerframework.org/manual`. Last accessed: 20.04.2024.

[2] J. Wharton. Rxrelay - readme.md. `https://github.com/JakeWharton/RxRelay/blob/master/README.md`. Last accessed: 19.04.2024.