Security in Software Applications: Homework 2

Professor: Francesco Parisi-Presicce

Student: Riccardo Chiaretti

Contents

In the `.zip` file, apart from this report there are other two Java files:

- `NewBagRiccardoChiaretti_annotations.java` contains the code where JML annotations are used;
- `NewBagRiccardoChiaretti_code.java` contains the class where everything has been fixed.

1 Introduction

This report is a step-by-step discussion of performing static analysis on a Java class called `NewBag.java`. In particular, this analysis is done with the help of a static checker which can be used to analyze Java code: the name of the tool is ESC/Java2 which stands for the "Extended Static Checker for Java".
It is a programming tool which tries to find common runtime errors in Java programs when compiled. What the tool does, is to use a set of techniques for statically checking the correctness of program constraints: this approach is known as *extended static checking*. Since that performing this kind of check there is the need to use a theorem prover, the one used by ESC/Java2 is *Simplify*.
Like many other tools in this category, ESC/Java2 is neither *sound* nor *complete*: not every warning it triggers is really a vulnerability (known as *false positive* samples) and it may not find every possible vulnerability in a program (so, having *false negatives*). In particular, ESC/Java2 suffers from false negatives if modular arithmetic and/or multithreading are used.

In the following, the Java class mentioned above is analyzed and, with the help of the tool, appropriate corrections are made. Specifically, the discussion is divided almost into two parts:

- at first, **JML annotations** are added to the code. More recent versions of ESC/Java2, allow the use of JML annotations in the code. These annotations let the user specify the number and the kind of checks by annotating the program with special comments called *pragmas*;
- afterwards, the result of the previous step plus some more modifications on the code due to the incompleteness of the tool, are used to write a new version of the class where all vulnerabilities have been removed.

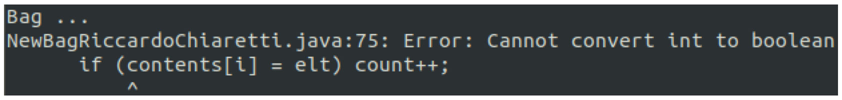
In addition, there are two more interesting features of the tool which can help in analyzing programs:

1. it groups warnings based on the method/constructor in which they are. Indeed, it is possible to work on a single method, going ahead only when all spotted warnings have been solved for that piece of code;
2. the second feature is that launching ESC/Java2 with the flag `-Suggest` *may* provide a suggestion which can be used to solve the specific warning.

2 Solution Using Annotations

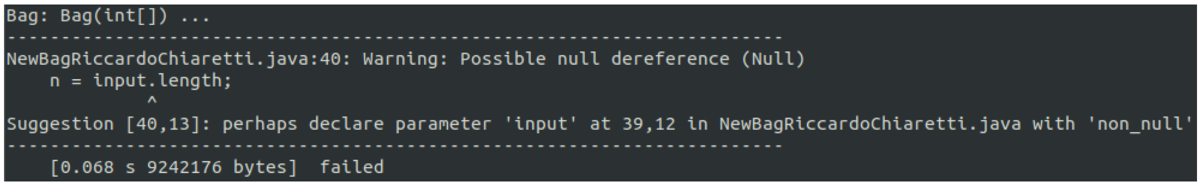
As already said above, in this first section it is provided a solution using JML annotations. These "special" comments are used to annotate a piece of code allowing the user to express the purpose of that code. In this way, ESC/Java2 can be run on the code to see whether the annotations (and so, the aim of the user) are coherent with respect to the implementation.

Launching the tool on the class as it is, it raises 1 error related to a conversion from `int` variable to a boolean condition.



```
Bag ...
NewBagRiccardoChiaretti.java:75: Error: Cannot convert int to boolean
      if (contents[i] = elt) count++;
          ^
```

Probably it is a typo but the tool cannot continue analyzing because there is no boolean expression to evaluate in the `if` statement. Since the function name is `getCount()` and takes as argument an `int` variable, it is assumed to count the number of occurrences of that variable so, the right change is `if (contents[i] == elt)`.
After the correction, the tool gives 22 warnings. The first of these, is about a possible null dereference of the array in input to the `Bag(int[])` constructor.

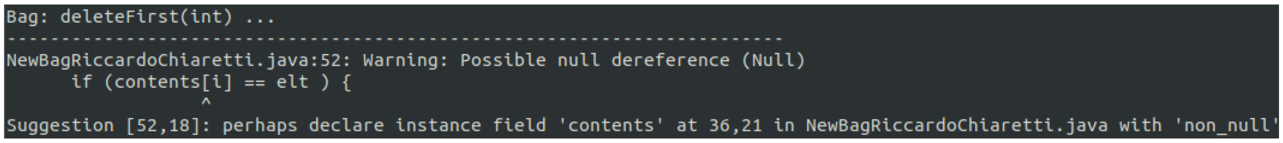


```
Bag: Bag(int[]) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:40: Warning: Possible null dereference (Null)
    n = input.length;
             ^
Suggestion [40,13]: perhaps declare parameter 'input' at 39,12 in NewBagRiccardoChiaretti.java with 'non_null'
------------------------------------------------------------------------
    [0.068 s 9242176 bytes]  failed
```

In this case the tool provides also the right suggestion. Indeed, since that the `input` array is dereferenced within the constructor, is must be not null to avoid a Null Pointer Exception when executing the code. The solution is using the `/*@non_null @*/` annotation just before the input parameter in the following way:

```
Bag(/*@non_null @*/int[] input) {
    ...
}
```

This annotation is a convenient short-hand annotation for expressing a pre-condition: it tells that the `input` array passed as parameter must not be a null reference. Another possibility could be using the `requires` annotation, imposing as precondition `input != null`.
After having added the annotation, the number of triggered warning is the same. This is due to the violation of the newly written precondition by another ”entity” which is using the constructor we have just modified. This warning will be covered later during the analysis.
The second method to be analyzed is `deleteFirst()`. The first problem spotted by ESC/Java2 is another null dereference.



```
Bag: deleteFirst(int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:52: Warning: Possible null dereference (Null)
      if (contents[i] == elt ) {
                  ^
Suggestion [52,18]: perhaps declare instance field 'contents' at 36,21 in NewBagRiccardoChiaretti.java with 'non_null'
```

Here, two possible solutions can be adopted:

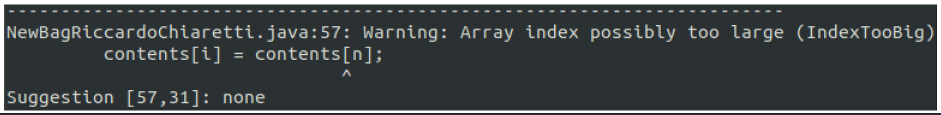
1. either we use the `requires` keyword forcing the precondition that, when the function is called, `contents` must be not null;
2. or it is better, as the tool is suggesting, to annotate the field `contents` in the class with the `non_null` keyword.

Indeed, in case the first solution is used, the annotation `/*@requires contents != null@*/` must be put on top of all functions which are either reading from the array or writing in it. On the other hand, the second choice is expressing an *invariant* which must hold in the whole class: the array `contents` must not be null[1].
Then, we have a `for` loop in which the array `contents` is both read and modified. In order to correctly handle the warnings related to it and to let the tool understand which should be the range where `n` is defined, the keyword `invariant` can be used. Indeed, since `n` keeps the number of element in the array `contents`, it should be in the range `[0, contents.length]`. The annotations for expressing this concept are the followings:

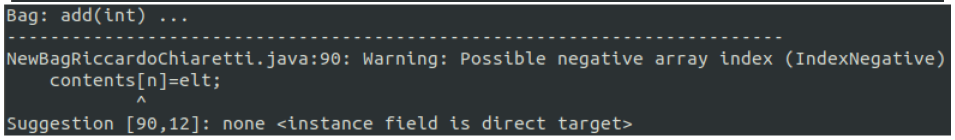
```
int n;
/*@invariant n >= 0; @*/
/*@invariant n <= contents.length; @*/
```

[1]Having chosen the first solution, the number of warnings is reduced to 19 while with the second one, the number is 21.

By using these annotations, the number of warnings is reduced from 19 to 15. This is because by expressing which is the range that the user had in mind for `n`, ESC/Java2 understands that some scenarios like `n` to be negative, do not happen anymore.



```
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:57: Warning: Array index possibly too large (IndexTooBig)
        contents[i] = contents[n];
                               ^
Suggestion [57,31]: none
```

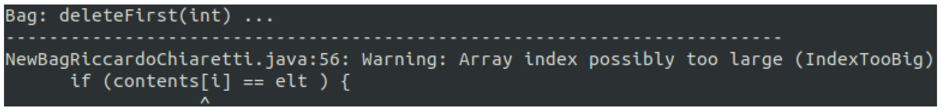


```
Bag: add(int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:90: Warning: Possible negative array index (IndexNegative)
    contents[n]=elt;
           ^
Suggestion [90,12]: none <instance field is direct target>
```

Indeed, the two scenarios described by the two pictures above, cannot happen anymore:

1. in the function `deleteFirst()`, since that `n` is at first decremented and then used as index for the array, the `IndexOutOfBoundsException` is not raised anymore. Indeed, even when `n = contents.length` it is at first decremented and then used to access the last element of `contents` (which is `contents[contents.length - 1]`);
2. the second warning is about using a negative integer as index for the array. While other languages like Python support using negative index to access elements of an array, Java does not. Using the annotations above (in particular, the one with `n >= 0`), we have explicitly said that `n` cannot be negative.

Next, the generated warning is about a possible too large index for accessing the array.



```
Bag: deleteFirst(int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:56: Warning: Array index possibly too large (IndexTooBig)
      if (contents[i] == elt ) {
                   ^
```

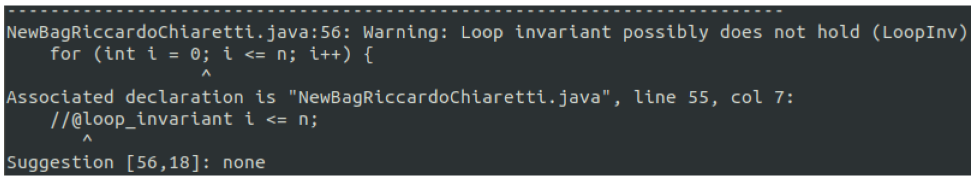
In order to let the tool understand the invariant which must hold when the loop is executed, there exist annotations for expressing *loop invariants*. Specifically, in this case, `i` should be in the range `[0, contents.length)`. Since `n` is at most `contents.length`, the annotations are the followings:

```
/*@loop_invariant i >= 0; @*/
/*@loop_invariant i < contents.length; @*/
for (...) {
    ...
}
```

However there is a problem: according to the way Esc/Java2 does *loop unrolling*, the second annotation should be changed in `/*@loop_invariant i <= contents.length; @*/`. Indeed, even if the aim of the user is to stop looping when `i` is at most `contents.length - 1`, consider the following scenario:

1. in case the first occurrence of `elt` is in the last position of the array, `n = contents.length` and `i = contents.length - 1`;
2. the condition in the `if` statement is verified (assuming as said before that the first occurrence of `elt` is at the end of `contents`);
3. `n` is decremented so, `n = contents.length - 1`;
4. at this point the condition `i == n` is true;

So, the tool warns about a possible violation of the loop invariant due to this boundary case. Modifying the annotation in `/*@loop_invariant i <= contents.length; @*/`, Esc/Java2 correctly warns a possible violation of the invariant.

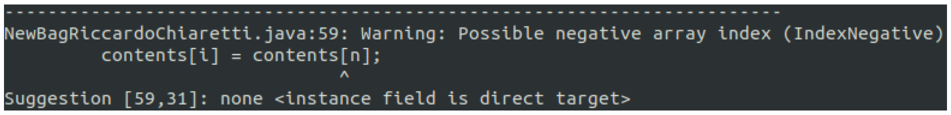


```
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:56: Warning: Loop invariant possibly does not hold (LoopInv)
    for (int i = 0; i <= n; i++) {
                  ^
Associated declaration is "NewBagRiccardoChiaretti.java", line 55, col 7:
    //@loop_invariant i <= n;
       ^
Suggestion [56,18]: none
```

Indeed, as it is, the stop condition in the `for` loop is wrong because `i` can be equal to `contents.length`, causing an `IndexOutOfBoundsException`. The right solution is to make strict the inequality in the loop:

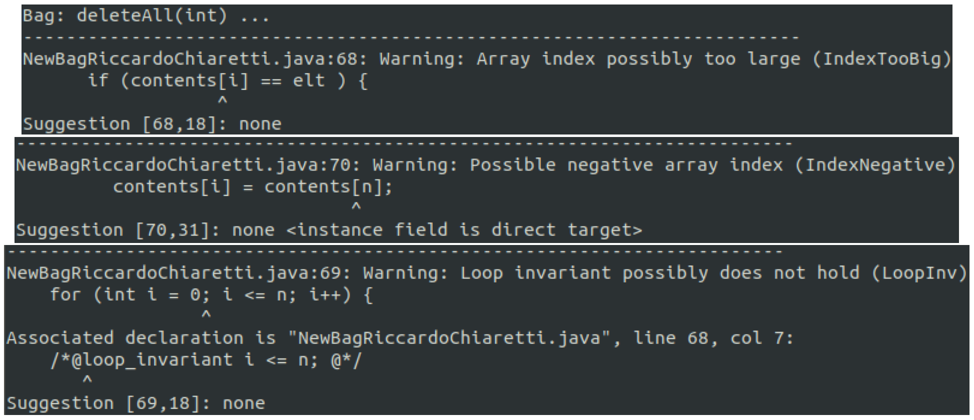
```
/*@loop_invariant i >= 0; @*/
/*@loop_invariant i <= contents.length; @*/
for (int i = 0; i < n; i++) {
    ...
}
```

Another warning was solved by the correction just made. It was about a possible negative value used as index in an array.



```
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:59: Warning: Possible negative array index (IndexNegative)
        contents[i] = contents[n];
                               ^
Suggestion [59,31]: none <instance field is direct target>
```

In case `n = 0` and the stop condition of the loop is `i <= n`, the variable `n` was decreased (reaching the value `-1`) before being used to access the last position of `contents`.
The warnings triggered for the method `deleteAll()` are exactly the same as the ones analyzed above for the `deleteFirst()` method.



```
Bag: deleteAll(int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:68: Warning: Array index possibly too large (IndexTooBig)
      if (contents[i] == elt ) {
                  ^
Suggestion [68,18]: none
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:70: Warning: Possible negative array index (IndexNegative)
        contents[i] = contents[n];
                               ^
Suggestion [70,31]: none <instance field is direct target>
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:69: Warning: Loop invariant possibly does not hold (LoopInv)
    for (int i = 0; i <= n; i++) {
                  ^
Associated declaration is "NewBagRiccardoChiaretti.java", line 68, col 7:
    /*@loop_invariant i <= n; @*/
       ^
Suggestion [69,18]: none
```

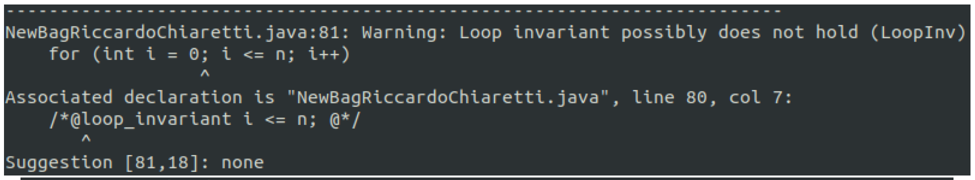
Thus, by making the same changes done to the `deleteFirst()` method, all warnings are solved apart from the one represented by the last picture. Here the difference is made by the missing `return` statement:

1. in case the first occurrence of `elt` is in the last position of the array, `n = contents.length` and `i = contents.length - 1`;
2. the condition in the `if` statement is verified (assuming as said before that the first occurrence of `elt` is at the end of `contents`);
3. `n` is decremented so, `n = contents.length - 1`;
4. at this point the condition `i == n` is true;
5. since there is no return statement, ESC/Java2 goes back to the start of the `for` loop verifying whether the stop condition is met or not;
6. due to the instruction `i++`, at the end of the previous loop, `i` is incremented reaching the value `contents.length` which is exactly `n + 1`.

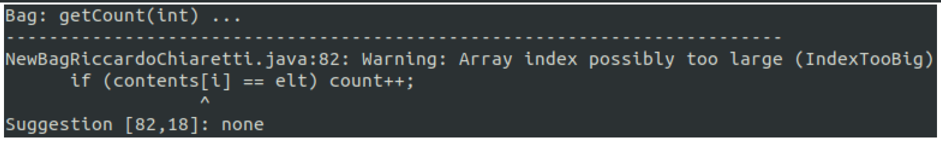
Due to this boundary case, the tool cannot verify the same loop invariant used in `deleteFirst()`; so, the right solution is the following:

```
/*@loop_invariant i >= 0; @*/
/*@loop_invariant i <= contents.length+1; @*/
for (int i = 0; i < n; i++) {
    ...
}
```

At this point, 11 warnings remain to be solved. Regarding `getCount()`, using the same `loop_invariants` as above, the only raised warning is one of those already analyzed: the variable `i` used as index in the array may possibly be too large.



```
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:81: Warning: Loop invariant possibly does not hold (LoopInv)
    for (int i = 0; i <= n; i++)
                    ^
Associated declaration is "NewBagRiccardoChiaretti.java", line 80, col 7:
    /*@loop_invariant i <= n; @*/
       ^
Suggestion [81,18]: none
```

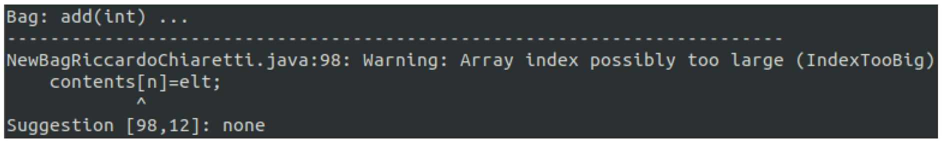


```
Bag: getCount(int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:82: Warning: Array index possibly too large (IndexTooBig)
      if (contents[i] == elt) count++;
                   ^
Suggestion [82,18]: none
```

Therefore, the solution adopted is the same as the one used in `deleteFirst()` apart from adding a new loop invariant for the `count` variable used in the loop:

```
/*@loop_invariant i >= 0; @*/
/*@loop_invariant i <= contents.length+1; @*/
/*@loop_invariant count >= 0; @*/
for (int i = 0; i < n; i++)
    ...
```

In the `add(int)` method, the warning raised by ESC/Java2 is about a possible too large index.



```
Bag: add(int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:98: Warning: Array index possibly too large (IndexTooBig)
    contents[n]=elt;
             ^
Suggestion [98,12]: none
```

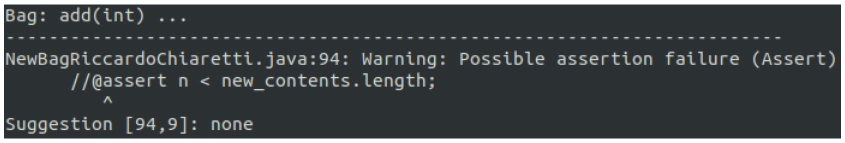
In this method, the paths that can be taken are two based on the condition of the if construct:

1. either `n < contents.length` so, there is still space where to put the new `int` element;
2. or `n = contents.length` and so there is the need to allocate more memory for `contents`.

In the end, what should be asserted is that when allocating a larger array, the situation is still that `n < contents.length`. In this way, when performing the insertion with `contents[n] = elt`, there is no risk to go out of the bounds of the array:

```
int[] new_contents = new int[2*n];
/*@assert n < new_contents.length; @*/
...
```

The tool, when evaluating the assertion, fails.

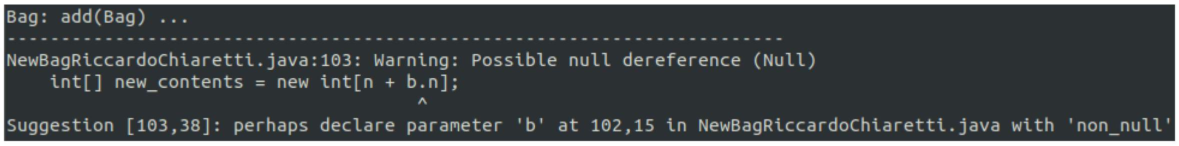


```
Bag: add(int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:94: Warning: Possible assertion failure (Assert)
      //@assert n < new_contents.length;
          ^
Suggestion [94,9]: none
```

Indeed, in case `n = 0` and `contents` has been initialized as in the `Bag()` constructor, the condition in the `if` statement is verified, `new_contents` is initialized with size `0` and when trying to put in the new element an `IndexOutOfBoundsException` is triggered. The right solution, is to modify the size of `new_contents` to be `2*n+1` and not only `2*n`:

```
int[] new_contents = new int[2*n+1];
/*@assert n < new_contents.length; @*/
...
```

For what concerns the `add(Bag)` method, the warning which ESC/Java2 raises is about a possible null dereference due to the object in input to be null.

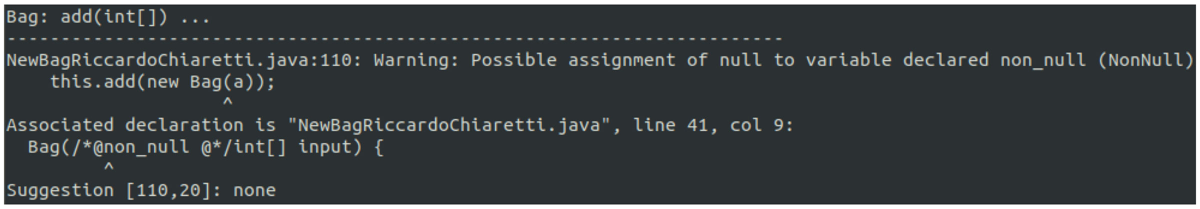


```
Bag: add(Bag) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:103: Warning: Possible null dereference (Null)
    int[] new_contents = new int[n + b.n];
                                     ^
Suggestion [103,38]: perhaps declare parameter 'b' at 102,15 in NewBagRiccardoChiaretti.java with 'non_null'
```

In this case, the suggestion of the tool is correct and so, requiring `Bag b` not to be null is the right solution:

```
void add(/*@non_null @*/Bag b) {
    ...
}
```

The next warning, which happens to be in the `add(int[])` method, is related to a precondition established in the constructor `Bag(int[])` previously analyzed.

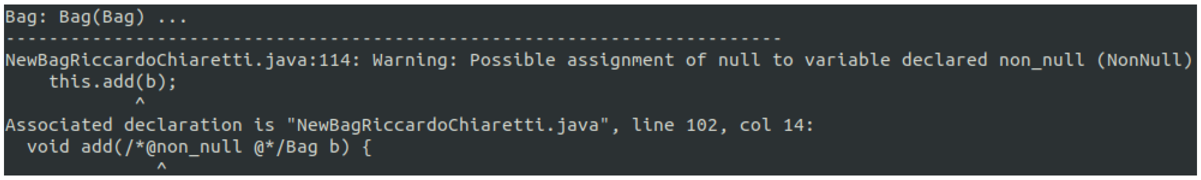


```
Bag: add(int[]) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:110: Warning: Possible assignment of null to variable declared non_null (NonNull)
    this.add(new Bag(a));
                    ^
Associated declaration is "NewBagRiccardoChiaretti.java", line 41, col 9:
  Bag(/*@non_null @*/int[] input) {
        ^
Suggestion [110,20]: none
```

Therefore, since that the `Bag(int[])` constructor takes in input a `non_null` array, also what the caller passes as input array must be not null:

```
void add(/*@non_null @*/int[] a) {
    ...
}
```

Regarding the constructor `Bag(Bag b)`, the tool triggers two warnings. The first one is the same one analyzed in the previous method.

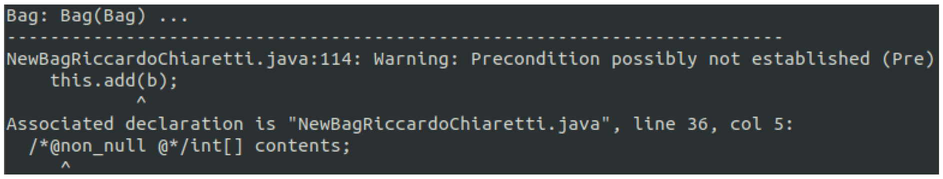


```
Bag: Bag(Bag) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:114: Warning: Possible assignment of null to variable declared non_null (NonNull)
    this.add(b);
            ^
Associated declaration is "NewBagRiccardoChiaretti.java", line 102, col 14:
  void add(/*@non_null @*/Bag b) {
              ^
```

Also the solution is the same:

```
Bag(/*@non_null @*/Bag b) {
    ...
}
```

The second warning is always about a violation but, this time, the constructor is violating the invariant on the field `contents`.

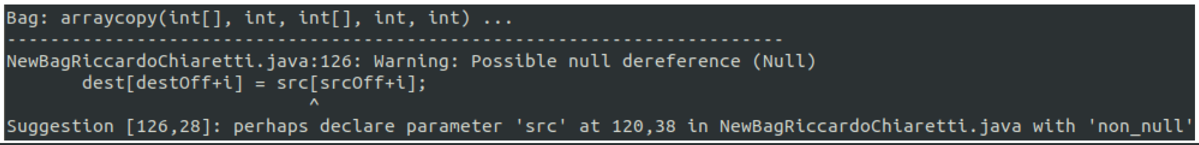


```
Bag: Bag(Bag) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:114: Warning: Precondition possibly not established (Pre)
    this.add(b);
            ^
Associated declaration is "NewBagRiccardoChiaretti.java", line 36, col 5:
  /*@non_null @*/int[] contents;
    ^
```

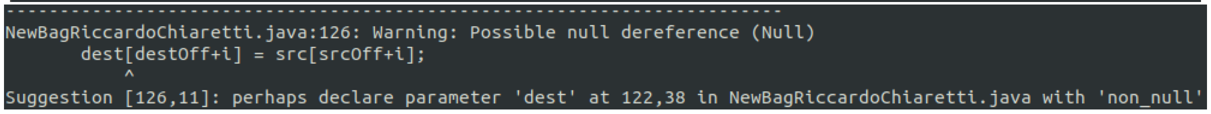
Indeed, being a constructor, it should instantiate both `n` and `contents`. Since it is not doing so and since that there is the invariant on `contents` which must not be null, the tool is highlighting that the invariant on the array does not hold. To fix it, the code should be modified (at least) in the following way:

```
Bag(/*@non_null @*/Bag b) {
    n = 0;
    contents = new int[0];
    ...
}
```

In the end, the last function to be analyzed is `arraycopy()` with 6 remaining warnings. The first two, are related to possible null dereference of the two input arrays `src` and `dest`.



```
Bag: arraycopy(int[], int, int[], int, int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:126: Warning: Possible null dereference (Null)
       dest[destOff+i] = src[srcOff+i];
                            ^
Suggestion [126,28]: perhaps declare parameter 'src' at 120,38 in NewBagRiccardoChiaretti.java with 'non_null'
```

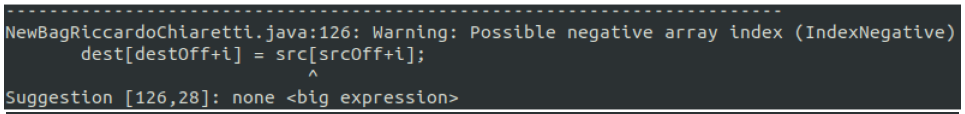


```
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:126: Warning: Possible null dereference (Null)
       dest[destOff+i] = src[srcOff+i];
           ^
Suggestion [126,11]: perhaps declare parameter 'dest' at 122,38 in NewBagRiccardoChiaretti.java with 'non_null'
```

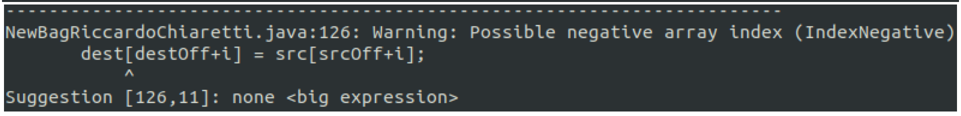
Since `src` is read while `dest` is written, it is required that both of them are not null. The suggestion of the tool is correct for both of them:

```
private static void arraycopy(
    /*@non_null @*/int[] src,
    int srcOff,
    /*@non_null @*/int[] dest,
    int destOff,
    int length) {
    ...
}
```

The other two, are related to a possible negative value used as index of an array.



```
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:126: Warning: Possible negative array index (IndexNegative)
       dest[destOff+i] = src[srcOff+i];
                            ^
Suggestion [126,28]: none <big expression>
```

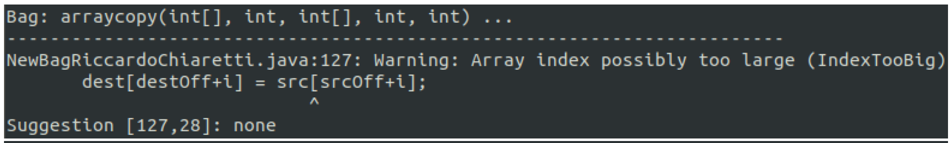


```
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:126: Warning: Possible negative array index (IndexNegative)
       dest[destOff+i] = src[srcOff+i];
           ^
Suggestion [126,11]: none <big expression>
```

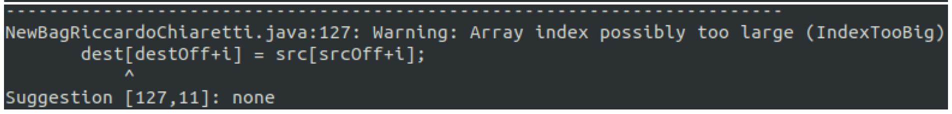
Indeed, since that there is no precondition on the offsets `srcOff` and `destOff`, in case even one of them is smaller than `length`, with most of the probability, an `IndexOutOfBoundsException` is triggered. The solution is to put a precondition on them to be positive:

```
/*@requires srcOff >= 0; @*/
/*@requires destOff >= 0; @*/
private static void arraycopy(...) {
    ...
}
```

The last two warnings are related to the value of the indexed inside the `for` loop.



```
Bag: arraycopy(int[], int, int[], int, int) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:127: Warning: Array index possibly too large (IndexTooBig)
       dest[destOff+i] = src[srcOff+i];
                            ^
Suggestion [127,28]: none
```

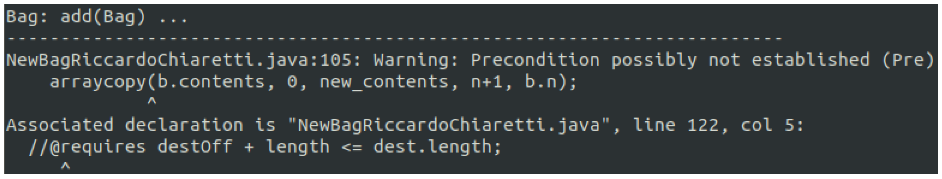


```
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:127: Warning: Array index possibly too large (IndexTooBig)
       dest[destOff+i] = src[srcOff+i];
           ^
Suggestion [127,11]: none
```

In particular, the case to avoid is that the sum of the offsets and the index of the loop do not go beyond the memory allocated for the arrays. Therefore, translating this statement in JML annotations the solution is the following:

```
...
/*@requires srcOff + length <= src.length; @*/
/*@requires destOff + length <= dest.length; @*/
private static void arraycopy(...) {
    ...
}
```

However, this is not enough for solving the two warnings and, even worse, another problem has arisen. It is another warning related to a precondition which is not established in the method `add(Bag)`.



```
Bag: add(Bag) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:105: Warning: Precondition possibly not established (Pre)
    arraycopy(b.contents, 0, new_contents, n+1, b.n);
             ^
Associated declaration is "NewBagRiccardoChiaretti.java", line 122, col 5:
  //@requires destOff + length <= dest.length;
    ^
```

Going back to that method, the precondition is not satisfied due to the fact that while the size of `new_contents` is `n + b.n`, the sum of the fourth and the last variables passed to `arraycopy()` is greater than `new_contents.length` (that is, `n + 1 + b.n > n + b.n = new_contents.length`). The solution is to delete that +1 in the third parameter:

```
void add(/*@non_null @*/Bag b) {
    ...
    arraycopy(b.contents, 0, new_contents, n, b.n);
    ...
}
```

Regarding the two warnings above the problem is the way the index of the loop is managed. In order to spot the problem a loop invariant can help. In particular the variable `i` must be strictly smaller than `length`:

```
/*@ loop_invariant i<= length;@*/
for(int i=0; i <= length; i++) {
    ...
}
```

As expected, the invariant does not hold. Indeed, remembering all the reasoning about the loop unrolling, in order to make it valid there is the need to change the stop condition of the loop in a strict inequality and to put another precondition on `length`:

```
...
/*@requires length >= 0; @*/
private static void arraycopy(...) {
    /*@ loop_invariant i<= length;@*/
    for(int i=0; i < length; i++) {
        ...
    }
}
```

This was the last warning triggered by ESC/Java2. However, remembering that it is unsound and incomplete, there still something to do which is discussed in the next section.

3 The New Class **Bag**

As already said in the introduction, there are problems which ESC/Java2 is not able to able to spot. While using annotations it is possible to understand why there may be a problem, there is still the need to fix it the code. This is what it is done in this section. In addition, there are also problems related to the semantics behind the program. Indeed, for example, the function `deleteFirst(int elt)` is expected to delete the first occurrence of `elt` while in the end it may not.

At first, regarding the access modifier of the class `Bag`, by default it is *package-private*, meaning that every class defined within the same package can extend the class `Bag`. In order to prevent malicious classes to override methods defined in `Bag`, a good programming rule is to make the class `final`. It is important to notice that in this way other classes can still use methods defined by the class `Bag` but their behavior cannot be modified.
A similar reasoning can be done on the two fields `n` and `contents`. Indeed without specifying any access modifier, by default they are `public`. This means that every class can access and modify them without any restriction. Since both `n` and `contents` can be read but not modified, to enforce *information hiding* they should be defined as `private` and retrieved using the appropriate `get()` method. So, in the original code two new methods are introduced:

- `getContents()` is the one which returns a *copy* of the content array. Indeed, just returning the reference to it, would create *aliases* which can be exploited by malicious code to change the elements of the referenced object (which, in this case, is the array `contents`);
- `getN()` on the other hand, is the method which returns the value of `n`.

Both of them are discussed later.

The first problem encountered in the section above was related to a possible null dereference of the input array in the constructor `Bag(int[])`. Having a null object as input will trigger, with most of probability a `NullPointerException` when getting the length of that array. Therefore, in case `input != null` the constructor correctly initializes both `n` and `contents` while, in the other case, it does not do anything.

```
/*@ensures (contents != null  && n == contents.length) || (n == 0 && contents == null); @*/
Bag(int[] input) {
    if (input != null){
        n = input.length;
        contents = new int[n];
        arraycopy(input, 0, contents, 0, n);
    }
}
```

The `ensures` keyword is establishing a postcondition on the function[2]: either one of the two situations happens depending on `input`.
Then there are the two methods which can be used to get `n` and a copy of `contents`. The postconditions on the functions do not trigger any warning in ESC/Java2[3].

```
/*@ensures (contents == null ==> \result == null) || (contents != null ==> (\forall int j; j >= 0
     && j < n; \result[j] == contents[j])); @*/
int[] getContents(){
    if (contents != null){
        int[] retContents = new int[contents.length];
        if (n > 0 && n <= contents.length){
            for (int i = 0; i < n; i++)
                retContents[i] = contents[i];
        }
        return retContents;
    }
    return null;
}

/*@ensures \result == n; @*/
int getN(){
    return n;
}
```

For the `deleteFirst(int elt)` method, apart from doing the same changes done in the previous section, there is a semantic problem. The method should delete the first occurrence of `elt` in `contents`; however, in case the last element of that array is `elt`, it is deleting the last occurrence and not the first one.

[2]In case the postcondition may not hold, ESC/Java2 triggers a warning related to a possible violation of a postcondition. Indeed, since the behavior of the function reflects what it is written in the annotation, the tool does not give any problem.
[3]The keyword `\result` is used to refer to the returned value/object by the function.

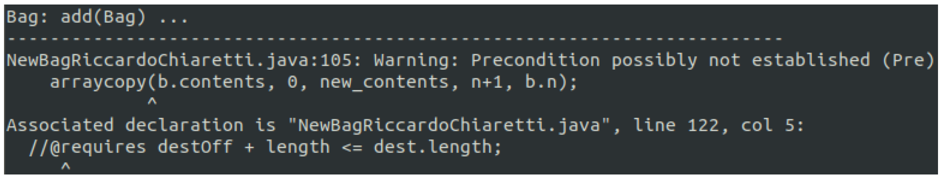
```
void deleteFirst(int elt) {
    if (contents != null && n <= contents.length && n > 0){
        for (int i = 0; i < n; i++) {
            if (contents[i] == elt) {
                int[] new_contents = new int[contents.length];
                for (int j = 0; j < i; j++)
                    new_contents[j] = contents[j];
                for (int j = 0; j < n-i-1; j++)
                    new_contents[j+i] = contents[j+i+1];
                n--;
                contents = new_contents;
                //@assert (\forall int j; j >= 0 && j < i; new_contents[j] != elt);
                //@assert (i + 1 < \old(n)) ==> ((contents[i] == elt) <==> (\old(contents[i+1])
                    == elt));
                return;
            }
        }
    }
}
```

Indeed, the assertions are stating the following:

1. the first one is asserting that if `elt` has been found at position `i`, then before that position there is no occurrence of `elt`;

2. the second assertion refers to the case in which, if `elt` is appearing *also* in position `i+1`, due to the deletion the new array has `elt` in position `i`.

The same problem is present also in the next method, that is, the `deleteAll(int elt)`. The purpose of this method is to delete all occurrences of the element `elt` in the array `contents`. Translating this statement into a JML postcondition, the obtained result is a warning of the tool.



```
Bag: add(Bag) ...
------------------------------------------------------------------------
NewBagRiccardoChiaretti.java:105: Warning: Precondition possibly not established (Pre)
    arraycopy(b.contents, 0, new_contents, n+1, b.n);
             ^
Associated declaration is "NewBagRiccardoChiaretti.java", line 122, col 5:
  //@requires destOff + length <= dest.length;
      ^
```

Indeed, the main problem with this function is the following:

1. `contents[n] = elt` and, at a generic iteration `i = k`, `contents[i] = elt`;

2. the condition in the `if` statement is true so the body is executed;

3. `n` is decremented and the instruction `contents[i] = contents[n]` is executed. At this point `contents[i]` is still equal to `elt`;

4. since `i` is incremented the position `k` is never considered anymore and there is still one element which is `elt`.

On the other hand, changing the implementation in the following way, `deleteAll()` does exactly its job without having the tool complaining about any violation of the postcondition:

```
/* the next method should remove from the array *all* the occurrences of elt */
/*@ ensures ((contents != null && n <= contents.length && n > 0) ==> (\forall int j; j >= 0 && j
    < n; contents[j] != elt)); @*/
void deleteAll(int elt) {
    int contents_n = n;
    if (contents != null && n <= contents.length && n > 0){
        for (int i = 0; i < n; i++) {
            if (contents[i] == elt ) {
                if (n == 1){
                    n--; //n = 0
                    contents = new int[contents.length];
                    return;
                } else{
                    n--;
                    contents[i] = contents[n];
                    i--; //To handle the fact that contents[i] can be elt again
                }
            }
        }
        if (contents_n != n) { //Only in case n is smaller than the beginning of the function,
            all elements after n are removed
            int[] new_contents = new int[contents.length];
            arraycopy(new_contents, 0, contents, 0, n);
            contents = new_contents;
        }
    }
}
```

In the `getCount()` method, the only things to add, are some checks on `contents` and `n` because it is possible that the first one is null and that `n` may go beyond the length of the array.

```
/*@ensures \result >= 0; @*/
int getCount(int elt) {
    int count = 0;
    if (contents != null && n <= contents.length && n > 0){
        for (int i = 0; i < n; i++)
            if (contents[i] == elt) count++;
    }
    return count;
}
```

Regarding the method `add(int elt)`, in the previous section it has already been explained the reason why the size of the new array should be `2*n + 1`. The code needs to be modified in such a way that the invariants expressed with JML annotations must hold:

```
/*@ensures (contents != null ==> ((n >= 0 && n <= contents.length) ==> (n == \old(n) + 1 &&
    contents[\old(n)] == elt))); @*/
void add(int elt) {
    if (contents != null){
        if (n == contents.length) {
            int[] new_contents = new int[2*n+1];
            /*@ assert n < new_contents.length;@*/
            arraycopy(contents, 0, new_contents, 0, n);
            contents = new_contents;
        }
        if (n >= 0 && n <= contents.length){
            contents[n] = elt;
            n++;
        }
    }
}
```

The postcondition is not violated according to ESC/Java2 and, indeed, it is right. What the annotations is expressing, is the following: if `contents` is not null and if `n` is in the correct range of values, then `elt` is appended at the end of `contents` and `n` is incremented.
In the next method, that is the `add(Bag b)`, all the precondition expressed as annotations need to be translated into the right checks in the code. Indeed, while the invariants on the fields `n` and `contents` are handy because they save many checks, when making the right changes to the code, things get more complicated.

```
void add(Bag b) {
    if (b == null)
        return;
    int b_n = b.getN();
    int[] b_contents = b.getContents();
    if (b_contents == null || contents == null)
        return;
    if (n < 0 || b_n < 0 || n > contents.length || b_n > b_contents.length)
```

```
        return;
    int size = n + b_n;
    int[] new_contents = new int[n + b_n];
    for (int i = 0; i < n; i++)
        new_contents[i] = contents[i];
    for (int j = 0; j < b_n; j++)
        new_contents[j+n] = b_contents[j];
    contents = new_contents;
    n = n + b_n;
    /*@assert (\forall int i; i >= 0 && i < \old(n); contents[i] == \old(contents[i])); @*/
    /*@assert (\forall int j; j >= 0 && j < b_n; contents[j+\old(n)] == b_contents[j]); @*/
    /*@assert n == contents.length; @*/
}
```

Apart from several checks which are needed to exit safely in case there are errors, the new stuff are the assertions. They are effectively asserting that the new `contents` array contains all the elements of the previous array plus the content of the array from the input `Bag b`. The last assertion, is as well important because in the original code, apart from (erroneously) copying the arrays, the increment of `n` was not done.
For what concerns the method `add(int[])` and the constructor `Bag(Bag)`, the only things to add are some checks on the input parameters not to be null when passed to other methods.

```
void add(int[] a) {
    if (a != null)
        this.add(new Bag(a));
}

Bag(Bag b) {
    if (b != null)
        this.add(b);
}
```

In the end, for the `arraycopy()`, all the preconditions expressed by JML annotations should be translated into conditions to be satisfied in an `if` construct. Also all the other changes done before should be applied.

```
//@ensures ((src != null && dest != null && srcOff > 0 && destOff > 0
&& srcOff + length <= src.length && destOff + length <= dest.length)
==> (\forall int i; i >= 0 && i < length; dest[destOff+i] == src[srcOff+i]));
private static void arraycopy(int[] src, int srcOff, int[] dest, int destOff, int length) {
    if (src != null && dest != null && srcOff >0 && destOff >0 && srcOff + length <= src.length
        && destOff + length <= dest.length)
    for(int i = 0 ; i < length; i++) {
        dest[destOff+i] = src[srcOff+i];
    }
}
```

The purpose of this method is to copy the source array into the destination one according to the specified offsets. This is exactly what the postcondition above expresses and, indeed, the tool does not raise any warning.

4 Points for Reflection

4.0.1 First Point

As said more than one time during the discussion, ESC/Java2 is neither sound nor complete. This means that during the analysis of a program, apart from those warnings related to false positive samples, the most dangerous are the false negatives.
That is the reason why: "A fool with a tool is still a fool!"; because it is responsibility of the tester to be able to spot vulnerabilities when the tool is not helpful.
In this case, there were some scenarios in which the tool did not give any help. Taking for example the function `add(Bag b)`, the tool did not give any warning about the violation of the invariant on the variable `n`. Indeed, the function was providing a (possibly) larger array of contents without correctly modifying the number of elements that is, `n`.
This is to explain that, in this situations, it is the experience of the person who is testing which makes the

real difference.
Regarding the Java class discussed in this paper, lots of problems have been solved both with the help of the tool and without it. However, there is a good chance that some subtle problems are still present.

4.0.2 Second Point

ESC/Java2 is somehow a powerful tool compared to others in the same category. In the following, there are some aspects of (more or less negative) the tool and of the specification that have jumped out during the analysis of the Java class:

- the first one is about ESC/Java2 *loop unrolling*. Indeed, the way according to which the tool interprets loop invariants can be misleading for a user. When for example the stop condition is `i < n`, which means that inside the body of the loop `i` is at most `n-1`, requiring the invariant to be `i <= n` may deceive the user to think that there may be an off-by-one error inside the loop;
- another limitation of the tool regards the usage of complex annotations. In particular, during the analysis of the class `Bag`, it happened that to express a postcondition on some function, there was the need of complex annotations. However, most of the times ESC/Java2 was not able to evaluate such complete expressions;
- the last thing which limited the analysis in consideration, is the fact that it is not possible to express pre/post-conditions using variables defined within the piece of code on which the assertion is written.

4.0.3 Third Point

There are several static code analyzer to be used instead of ESC/Java2. Apart from some disadvantages like those listed above, this tool is quite good because is able to spot the majority of errors in the code.
One missing feature which can be helpful in complex projects but which was not required for the small piece of code just analyzed, is the possibility to spot unused variables. Among the various tools, one which can be helpful in this situation is **PMD[1]**.

References

[1] https://pmd.github.io/pmd-6.9.0/index.html