# Security in Software Applications

Project 2 - Program verification with ESC/Java2

**Matteo Mariani**

1815188

# Contents

# Introduction

*ESC/JAVA2*, the "Extended Static Checker for Java", is a programming tool that attempts to find common runtime errors in Java programs at compile time.
The underlying approach used in ESC/Java is referred to as extended static checking, which is a collective name referring to a range of techniques for statically checking the correctness of various program constraints.
The tool consists of a:

1. parsing phase (syntax checks);
2. typechecking phase (type and usage checks);
3. static checking phase (reasoning to find potential bugs) - running a prover called *Simplify*.

Parsing and typechecking produce cautions or errors. Static checking produces warnings. The focus of ESC/Java2 is particularly on static checking.

*JML* is a behavioral interface specification language for Java modules. JML provides semantics to formally describe the behavior of a Java module, preventing ambiguity with respects to the module designers' intentions.

## Assignment

Our goal is to correct `NewBag.java` class in order to solve the warnings that ESC/JAVA2 discovers.
We can pursue two strategies:

- either add JML annotations to the code;
- or directly fix the bugs in the code.

Since the code was semantically wrong in different parts, some modifications of the code were necessary in order to correct the behavior of the class.

I divided the assignment in **three phases**:

- first phase: I updated the code with JML annotations in order to inject the logic that a `Bag` class must follow. This is particular useful because JML helps ESC/JAVA2 to understand the meaning of some portions of the code without producing a multitude of false positive warnings;
- second phase: I modified the code to solve any warning that ESC/JAVA2 shows. This is done because some parts of the code can't be solved by simply notify the tool about the intention of our class;
- third phase: the goal is to correct some errors in the `Bag` class, mostly related to the lack of common and suggested Java programming rules.

# Phase 1: Injection of JML annotations

In the following section we're inserting some JML annotations into the original code in order to inject the semantic that an object `Bag` must follow. This preliminary step was done in order to give to ESC/JAVA2 the capability to understand which behavior should be followed by the class.

The JML annotations used in the analysis are:

- `requires`, used to specify a precondition that must hold when a method is called;
- `ensures`, used to specify a postcondition that must hold when a method ends;
- `assert`, used to specify a property that must hold at some point in the code;
- `invariant`, used to specify a property that must be maintained by all methods;
- `loop_invariant`, used to specify a property that must hold during the entire loop;
- `non_null`, a shorthand used to specify that a variable must not be `null` across the entire code or during a function call;
- `assignable`, a property used to limit possible side-effects of a method. In particular, a variable that is *assignable* in a method must be the *only* variable that the method can modify;
- `\old(<var>)`, used to refer to the value of `<var>` before the execution of a method. This is particularly useful in conjunction with other annotations;
- `\forall(...)`, used to check whether some condition holds during a specified loop. This is particularly useful in conjunction with other annotations;
- `\result`, a shortcut used to refer to the return value of a function. This is particularly useful in conjunction with other annotations.

First of all: what is a `Bag`? It's an object that contains some numbers and has two fields:

- `contents`, that is an array of integers;
- `n`, that represents the number of integers inside the object.

**Original code**

```
class Bag {

int[] contents;
int n;
(...)
```

**Original code with JML annotations**

```
class Bag {

/*@ non_null @*/ int[] contents;
int n;
/*@ invariant 0 <= n; @*/
/*@ invariant n <= contents.length; @*
(...)
```

Since a `Bag` can contain less numbers than the ones that it can actually store, the relations `n <= contents.length` and `n >= 0` should always hold.
Notice that the elements that should have meaning for a `Bag` object are from `contents[0]` to `contents[n-1]`. The ones from `contents[n]` to `contents[contents.length-1]` should not be considered since `n` is used exactly to keep track of the "good" elements inside the array.

Semantically speaking, it's also correct thinking that the property `contents != null` should be always satisfied inside the class. In this way we allow `contents` to be empty but the `NullPointerException` is handled appropriately.

Thanks to these invariants, all methods in `newBag.java` must satisfy these properties in every point. For this reason, writing something like `@ensures/requires n >= 0` will be useless since this semantic is already part of the model that has to be checked by the tool.

Let's consider the constructors.

**Original code**

```
Bag(int[] input) {
    n = input.length;
    contents = new int[n];
    arraycopy(input, 0, contents, 0, n);
}

Bag() {
    n = 0;
    contents = new int[0];
}

Bag(Bag b) {
    this.add(b);
}
```

**Original code with JML annotations**

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

Bag() {
    n = 0;
    contents = new int[0];
}

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

The first one creates an object `Bag` with the numbers contained in the array passed in input.
A precondition that must be satisfied is `input != null`, otherwise the creation of a `Bag` object doesn't make any sense. Notice that an empty `Bag` can be created either by passing an array that is empty (`input.length == 0`) or by calling the second constructor.
Once the constructor ends, the code must guarantee that the fields are correctly initialized: `contents.length` and `n` should be equal to the length of the array in input. However, there is a problem: since we don't know if all the elements inside `input` are "good", we can't properly initialize `n` in such a way to respect the logic that a `Bag` must follow. In phase 2 we will modify this ambiguity by slightly changing the constructor.

The second constructor creates an empty `Bag` so it doesn't need any annotation since the invariants put directly on `n` and `contents` are the only ones that are necessary.

The third one creates a `Bag` that is equal to the `Bag b` passed as parameter. This should only have the precondition that `b != null`.

Now, let's consider the methods.

`deleteFirst(elt)` should delete the first occurrence of `elt` from a `Bag`.

**Original code**

```
void deleteFirst(int elt) {
  for (int i = 0; i <= n; i++) {
    if (contents[i] == elt ) {
        n--;
        contents[i] = contents[n];
        return;
    }
  }
}
```

**Original code with JML annotations**

```
void deleteFirst(int elt) {
  //@ loop_invariant (i <= n);
  for (int i = 0; i <= n; i++) {
    if (contents[i] == elt) {
        n--;
        contents[i] = contents[n];
        return;
    }
  }
}
```

The most important constraint that this method should have is the one that denies an improper access to `contents`. By inserting a loop invariant `i <= n` we're saying that we have to consider only the first `n` elements of `contents` (from `contents[0]` to `contents[n-1]`).
Why <= instead of < ? At the end of the last correct iteration of the loop, we've `i = n-1`. In the next one, we'll have that `i` is incremented (so `i == n`), then it is evaluated. Since the condition is violated, the loop ends *but* after that we've still `i == n`. The loop invariant annotation considers the last step as a tentative to enter in the loop, so its condition must hold also there. For this reason we've to insert `i <= n` instead of `i < n`.
Notice that this logic will also hold in the following loops.
As can be clearly seen, the code doesn't satisfy this property since an access to `contents[n]` can happen (as in all other loops in `newBag.java`), but any modification of the code is postponed in phase 2.

`deleteAll(elt)` should delete *all* occurrences of `elt` from a `Bag`.

**Original code**

```
void deleteAll(int elt) {
  for (int i = 0; i <= n; i++) {
    if (contents[i] == elt ) {
        n--;
        contents[i] = contents[n];
    }
  }
}
```

**Original code with JML annotations**

```
/*@ ensures (\forall int j; j>=0 && j<n;
    contents[j] != elt); @*/
void deleteAll(int elt) {
  //@ loop_invariant (i <= n+1);
  for (int i = 0; i <= n; i++) {
    if (contents[i] == elt ) {
        n--;
        contents[i] = contents[n];
    }
  }
}
```

As in the previous method, a loop invariant is needed in order to guarantee a correct access of `contents`. *However* this time the condition is slightly different in order to take into account the absence of the `return` statement once `n` is decremented.
Moreover, since we can have multiple occurrences of `elt` to eliminate, `n` can decrement multiple times and the invariant can't hold with the condition written above. This problem will be handled in phase 2 when we'll modify the code, adjusting `i` when needed.

In addition, an ensure JML annotation is inserted in order to check whether all occurrences of `elt` are correctly deleted from `contents` after the method ends.

`getCount(elt)` is used to count the occurrences of `elt` inside a `Bag`.

**Original code**

```
int getCount(int elt) {
  int count = 0;
  for (int i = 0; i <= n; i++)
    if (contents[i] == elt)
      count++;
  return count;
}
```

**Original code with JML annotations**

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

Since a loop is present, a loop invariant is needed to correctly access `contents`.
An additional loop invariant is used to check if `count` is properly incremented during the loop.
Finally, the postcondition that must hold is that the return value of the method is positive (`\result >= 0`).

`add(elt)` is used to add `elt` into a `Bag`. If the `Bag` is full then a new array `contents` with doubled size is created in order to store the new number.

**Original code**

```
void add(int elt) {
    if (n == contents.length) {
       int[] new_contents = new int[2*n];
       arraycopy(contents, 0,
                 new_contents, 0, n);
       contents = new_contents;
    }
    contents[n]= elt;
    n++;
}
```

**Original code with JML annotations**

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

This method must guarantee two things after its execution.
The first one is that the value of `n` is incremented by one at the end of the execution of the method.
The second postcondition is used to check whether `elt` was correctly inserted in `contents`, in the right position.

The assertion is used to check if the value of `n` is strictly less than the length of the new array. This is done in order to handle the case in which this method is called by a `Bag` that is empty. In that case, both `n` and `contents.length` are 0 so the initialization of the new array will be wrong, creating an other empty one (since it will have `2*n` elements). The code will be accordingly modified in phase 2.

`add(Bag)` is used to merge the elements of the caller with the ones of the `Bag` passed as parameter.

**Original code**

```
void add(Bag b) {
    int[] new_contents = new int[n + b.n];
    arraycopy(contents, 0,
              new_contents, 0, n);
    contents = new_contents;
  }
```

**Original code with JML annotations**

```
/*@ ensures n == contents.length; @*/
void add(/*@ non_null @*/ Bag b) {
    int[] new_contents = new int[n + b.n];
    arraycopy(contents, 0,
              new_contents, 0, n);
    arraycopy(b.contents, 0,
              new_contents, n+1, b.n);
    contents = new_contents;
}
```

The `Bag` passed as parameter must not be `null` is order to correctly get access to its fields.
The postcondition is that the number `n` of "good" elements in `contents` is exactly equal to the new `contents.length`. This is done because we're creating a new array containing only the "good" numbers coming from both `this.contents` and `b.contents`.

`add(int[])` is used to merge the elements of the caller `Bag` with the ones of the array passed as parameter.

**Original code**

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

**Original code with JML annotations**

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

A trivial precondition is that the array must not be `null`.
Notice that this method decides to accomplish its task by creating a new `Bag` object with the elements of `a` and then calling the previous method `add` for merging two `Bags`.

`arraycopy(...)` is used to copy `length` elements from position `srcOff` of `src` array to `dest` array, starting from the position `destOff`.

**Original code**

```
private static void arraycopy(int[] src,
                              int srcOff,
                              int[] dest,
                              int destOff,
                              int length)
                                  {
    for( int i=0 ; i<=length; i++) {
       dest[destOff+i] = src[srcOff+i];
    }
}
```

**Original code with JML annotations**

```
/*@ requires srcOff >= 0; @*/
/*@ requires srcOff + length <=
             src.length; @*/
/*@ requires destOff >= 0; @*/
/*@ requires destOff + length <=
             dest.length; @*/
/*@ requires length >= 0; @*/
/*@ assignable dest[*]; @*/
/*@ ensures (\forall int m;
             m>=0 && m<length;
    dest[destOff+m] == src[srcOff+m]); @*/
private static void arraycopy(
            /*@ non_null @*/ int[] src,
                             int srcOff,
            /*@ non_null @*/ int[] dest,
                             int destOff,
                             int length) {
    //@ loop_invariant (i <= length);
    for( int i=0 ; i<=length; i++) {
      dest[destOff+i] = src[srcOff+i];
    }
}
```

This method has a lot of preconditions.
The number of elements to copy (`length`) and the starting position of both arrays (`srcOff` and `destOff`) must be non-negative.
In order to avoid overflow in accessing arrays, the sum between the starting position and the number of elements to copy must be less or equal than the length of their correspondent arrays.
Since this method should only read the elements in `src` and write them into `dest`, the only things that can be modified during this call are the elements of `dest` array: `assignable dest[*]` is used to take into account this constraint.
The postcondition is trivial: once the method ends the elements copied into `dest` must be the ones present in `src`, respecting the offsets given in input.
Lastly, there is a loop invariant to have the proper access on both arrays.

# Phase 2: ESC/JAVA2 application and code modification

Now that we've inserted the necessary JML annotations, we can safely run ESC/JAVA2 tool over the code and try to solve the warnings.
Thanks to the annotations, the probability that the tool produces false positives is very low due to the lack of knowledge about the semantic that an object `Bag` should follow.

In order to enhance ESC/JAVA2's capabilities on spotting defects, I used the option `-Loop <iteration_count>` to enable loop unrolling[1] up to `<iteration_count>`. This is particularly useful for checking bounds of arrays when accessing them during a loop.

As done in phase 1, let's consider the constructors.
The warnings raised by the tool are:

```
Bag: Bag(int[]) ...
    [0.522 s 8856024 bytes]  passed

Bag: Bag() ...
    [0.063 s 9174032 bytes]  passed

Bag: Bag(Bag) ...
------------------------------------------------------------------------
progressBag.java:24: Warning: Precondition possibly not established (Pre)
    this.add(b);
             ^
Associated declaration is "progressBag.java", line 3, col 6:
  /*@ non_null @*/int[] contents;
       ^
------------------------------------------------------------------------
    [0.204 s 9108296 bytes]  failed
```

**Original code with JML annotations**

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

Bag() {
    n = 0;
    contents = new int[0];
}

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

**Solution to ESC/JAVA2's warnings**

```
/*@ ensures n == n_input; @*/
/*@ ensures
    input.length == contents.length; @*/
/*@ requires 0 <= n_input; @*/
/*@ requires n_input <= input.length; @*/
Bag( /*@ non_null */ int[] input,
                     int n_input) {
  n = n_input;
  contents = new int[input.length];
  arraycopy(input, 0, contents, 0, n);
}

Bag() {
  n = 0;
  contents = new int[0];
}

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

[1] The goal of loop unrolling - or loop unwinding - is to increase a program's speed by reducing or eliminating instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration.

The tool complains only for the last constructor, specifying that `contents` may be `null` after the call to `this.add(b)`. Indeed, this is true: neither `n` nor `contents` are properly initialized before calling the method `add`.
Since this constructor creates an object `Bag` copying entirely an other `Bag b`, we can solve the warning by simply initialize the `Bag` with an empty array (`contents = new int[0]`).

Even if the tool says nothing about the first constructor, I decided to modify it in order to better respect the logic that an object `Bag` should have. Since we're creating an object that must keep track of the number of "good" elements that are inside it, we should know which is the number of "good" elements that `input` has (`n_input`). This is particularly useful if we want a modular constructor that is able to dynamically build a `Bag` with a maximum size that is greater than the number of "good" elements inside `input`.
This choice isn't related neither to solve any warning or to correct some part of the code, it's just a way to fully respect the semantic that an object `Bag` has.

Notice that some additional JML annotations are inserted in order to take into account the new variable `n_input`.

Now let's consider the methods.
For `deleteFirst(elt)`, the warnings raised are:

```
Bag: deleteFirst(int) ...
------------------------------------------------------------------------
progressBag.java:31: Warning: Array index possibly too large (IndexTooBig)
    if (contents[i] == elt) {
                ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 30, col 2.

------------------------------------------------------------------------
progressBag.java:33: Warning: Possible negative array index (IndexNegative)
        contents[i] = contents[n];
                      ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 30, col 2.
    Executed then branch in "progressBag.java", line 31, col 28.

------------------------------------------------------------------------
progressBag.java:30: Warning: Loop invariant possibly does not hold (LoopInv)
  for (int i = 0; i <= n; i++) {
              ^
Associated declaration is "progressBag.java", line 29, col 6:
  //@ loop_invariant (i <= n);
      ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 30, col 2.
    Executed else branch in "progressBag.java", line 31, col 4.
    Reached top of loop after 1 iteration in "progressBag.java", line 30, col 2.

------------------------------------------------------------------------
    [0.149 s 9011272 bytes]  failed
```

**Original code with JML annotations**

```
void deleteFirst(int elt) {
  //@ loop_invariant (i <= n);
  for (int i = 0; i <= n; i++) {
    if (contents[i] == elt) {
        n--;
        contents[i] = contents[n];
        return;
    }
  }
}
```

**Solution to ESC/JAVA2's warnings**

```
void deleteFirst(int elt) {
  int[] new_contents;
  //@ loop_invariant (i <= n);
  for (int i = 0; i < n; i++) {
    if (contents[i] == elt) {
        n--;
        new_contents =
            new int[contents.length];
        arraycopy(contents, 0,
            new_contents, 0, i);
        arraycopy(contents, i+1,
            new_contents, i, n-i);
        contents = new_contents;
        return;
    }
  }
}
```

The method is wrong in different ways:

- there is a possible out-of-bounds access of `contents`. If `n < contents.length`, the access `contents[n]` should not be legal because we're accessing a wrong element of `Bag` - otherwise we're not respecting the semantic attributed to the field `n`. If `n = contents.length`, the access `contents[n]` will be an error (`ArrayIndexOutOfBoundsException`). All warnings are raised because of this possibility;
- we can't be sure that we're deleting the first occurrence of the `elt` element. Assuming that `contents[n]` is a legal access, if both `contents[i]` and `contents[n]` are `elt` then we're inserting in `contents[i]` the same value. Even if they are different entities, the logic is wrong: we're not deleting the first occurrence;
- we have a security problem: even if we're keeping track of the "good" elements inside the array by decreasing `n`, an access to something between `n` and `contents.length-1` is syntactically legal - even though not semantically. This possibility permits anyone to have access to some elements that can be either garbage (because they are old) or sensible. Notice that this problem is related to both `delete` methods.

The first problem can be solved by correcting the loop condition into `i < n`: in this way, we can safely access the array.
To solve the second one, I decided to create a new array `new_contents` and use the `arraycopy(...)` function: when `elt` is found at position `i`, first we copy the elements of `contents` into `new_contents` from position 0 to position `i-1` then we copy the remaining elements of `contents`, skipping the position `i`. It's like left-shifting the elements of `contents` from the position in which `elt` is found.
The last problem is handled by instantiating `new_contents` with a number of elements equal to `contents.length`. In this way the original length of the array is maintained and no elements are present beyond `contents[n-1]`, they are all zeroed.

For `deleteAll(elt)`, the warnings raised are:

```
Bag: deleteAll(int) ...
------------------------------------------------------------------------
progressBag.java:45: Warning: Array index possibly too large (IndexTooBig)
    if (contents[i] == elt ) {
                ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 44, col 2.

------------------------------------------------------------------------
progressBag.java:47: Warning: Possible negative array index (IndexNegative)
        contents[i] = contents[n];
                      ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 44, col 2.
    Executed then branch in "progressBag.java", line 45, col 29.

------------------------------------------------------------------------
progressBag.java:44: Warning: Loop invariant possibly does not hold (LoopInv)
  for (int i = 0; i <= n; i++) {
                ^
Associated declaration is "progressBag.java", line 43, col 6:
  //@ loop_invariant (i <= n+1);
      ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 44, col 2.
    Executed then branch in "progressBag.java", line 45, col 29.
    Reached top of loop after 1 iteration in "progressBag.java", line 44, col 2.
    Executed then branch in "progressBag.java", line 45, col 29.
    Reached top of loop after 2 iterations in "progressBag.java", line 44, col 2.

------------------------------------------------------------------------
    [0.181 s 9375104 bytes]  failed
```

**Original code with JML annotations**

```
/*@ ensures (\forall int j; j>=0 && j<n;
    contents[j] != elt); @*/
void deleteAll(int elt) {
  //@ loop_invariant (i <= n+1);
  for (int i = 0; i <= n; i++) {
    if (contents[i] == elt ) {
        n--;
        contents[i] = contents[n];
    }
  }
}
```

**Solution to ESC/JAVA2's warnings**

```
/*@ ensures (\forall int j; j>=0 && j<n;
    contents[j] != elt); @*/
void deleteAll(int elt) {
  int n_old = n;
  //@ loop_invariant (i <= n+1);
  for (int i = 0; i < n; i++) {
    if (contents[i] == elt ) {
        if (n == 1){
            n--;
            contents =
               new int[contents.length];
            return;
        } else {
            n--;
            contents[i] = contents[n];
            i--;
        }
    }
  }
  if (n_old != n) {
    int[] new_contents =
              new int[contents.length];
    arraycopy(new_contents, 0,
              contents, 0, n);
    contents = new_contents;
  }
}
```

Again, the warnings can be solved by adjusting the loop condition into `i < n`.
*However* the real problem is that the code suffers from the same problems that I have pointed out in the previous function: it can happen that not all occurrences of `elt` are properly eliminated.
Since `contents[n]` can be `elt` as well, if we just put that into `contents[i]` without checking this possibility we will not delete all `elt` occurrences from a `Bag`. To handle this, we can decrement `i` by one when `elt` is found: in this way we are able to check the same position on the array.
Unfortunately, this mechanism doesn't hold when `n = 1`: not only because `i` will become negative, but also because we'll not remove the element at position 0. The first branch was inserted exactly to handle this possibility, creating an empty array.

Since we want to handle the same security problem seen in `deleteFirst(elt)` function, we can create a new array in the same way we have described before.

For `getCount(elt)`, the warnings raised are:

```
Bag: getCount(int) ...
------------------------------------------------------------------------
progressBag.java:58: Warning: Array index possibly too large (IndexTooBig)
    if (contents[i] == elt) count++;
                ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 57, col 2.

------------------------------------------------------------------------
progressBag.java:57: Warning: Loop invariant possibly does not hold (LoopInv)
  for (int i = 0; i <= n; i++)
              ^
Associated declaration is "progressBag.java", line 55, col 6:
  //@ loop_invariant (i <= n);
      ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 57, col 2.
    Executed then branch in "progressBag.java", line 58, col 28.
    Reached top of loop after 1 iteration in "progressBag.java", line 57, col 2.

------------------------------------------------------------------------
    [0.103 s 9325568 bytes]  failed
```

**Original code with JML annotations**

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

**Solution to ESC/JAVA2's warnings**

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

The tool complains about possible wrong accesses to `contents`.
Again, these warnings can be solved by adjusting the loop condition into `i < n`.

For `add(elt)`, the warnings raised are:

```
Bag: add(int) ...
------------------------------------------------------------------------
progressBag.java:70: Warning: Possible assertion failure (Assert)
       //@ assert n < new_contents.length;
              ^
Execution trace information:
    Executed then branch in "progressBag.java", line 68, col 30.

------------------------------------------------------------------------
    [0.106 s 9521472 bytes]  failed
```

**Original code with JML annotations**

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

**Solution to ESC/JAVA2's warnings**

```
/*@ ensures n == \old(n)+1; @*/
/*@ ensures contents[\old(n)] == elt; @*/
void add(int elt) {
  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;
  }
  contents[n] = elt;
  n++;
  }
```

The tool warns about the possibility that the assertion may fail. The annotation was, indeed, introduced to cover the case in which `n = 0` and the problem is solved instantiating a new array with `2*n+1` positions.

For `add(Bag)`, the warnings raised are:

```
Bag: add(Bag) ...
------------------------------------------------------------------------
progressBag.java:82: Warning: Precondition possibly not established (Pre)
    arraycopy(b.contents, 0, new_contents, n+1, b.n);
              ^
Associated declaration is "progressBag.java", line 94, col 6:
  /*@ requires destOff + length <= dest.length; @*/
      ^
------------------------------------------------------------------------
    [0.19 s 9212808 bytes]  failed
```

**Original code with JML annotations**

```
/*@ ensures n == contents.length; @*/
void add(/*@ non_null @*/ Bag b) {
    int[] new_contents = new int[n + b.n];
    arraycopy(contents, 0,
              new_contents, 0, n);
    arraycopy(b.contents, 0,
              new_contents, n+1, b.n);
    contents = new_contents;
}
```

**Solution to ESC/JAVA2's warnings**

```
/*@ ensures n == contents.length; @*/
void add(/*@ non_null @*/ Bag b) {
  int[] new_contents = new int[n + b.n];
  arraycopy(contents, 0,
            new_contents, 0, n);
  arraycopy(b.contents, 0,
            new_contents, n, b.n);
  contents = new_contents;
  n = this.n + b.n;
}
```

The warning comes from the fact that when we're calling `arraycopy` we're violating the precondition imposed on `dest` array - that in this case is `new_contents`. In particular, if the starting position is `n+1` then the last position that we're trying to access (`n+1+b.n`) is clearly out of bounds.
In fact, we're wrongly "excluding" one position from the first to the second call to `arraycopy`: we should start from `n`, not `n+1`.

Although there is a precondition, ESC/JAVA2 doesn't recognize properly the use of `b.n` in the code and in the annotation. So, even if `n` is *never* updated in the code, there is no warning about the possibility that `n == contents.length` might not hold after the call. Actually, this will never hold and the last line of the code is used to correctly update `n`.

For `add(int[])`, there are no warnings.

```
Bag: add(int[]) ...
    [0.053 s 9642912 bytes]  passed
```

**Original code with JML annotations**

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

**Solution to ESC/JAVA2's warnings**

```
/*@ requires 0 <= n_a; @*/
/*@ requires n_a <= a.length; @*/
void add(/*@ non_null @*/ int[] a,
                          int n_a) {
    this.add(new Bag(a,n_a));
}
```

Although there are no warnings, the code was altered in order to handle the modification done to the constructor `Bag(int[])` already discussed at the beginning of this section.

For `arraycopy(...)`, the warnings raised are:

```
Bag: arraycopy(int[], int, int[], int, int) ...
------------------------------------------------------------------------
progressBag.java:105: Warning: Array index possibly too large (IndexTooBig)
      dest[destOff+i] = src[srcOff+i];
                        ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 104, col 4.

------------------------------------------------------------------------
progressBag.java:105: Warning: Array index possibly too large (IndexTooBig)
      dest[destOff+i] = src[srcOff+i];
            ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 104, col 4.

------------------------------------------------------------------------
progressBag.java:104: Warning: Loop invariant possibly does not hold (LoopInv)
    for( int i=0 ; i<=length; i++) {
                 ^
Associated declaration is "progressBag.java", line 103, col 8:
    //@ loop_invariant (i <= length);
        ^
Execution trace information:
    Reached top of loop after 0 iterations in "progressBag.java", line 104, col 4.
    Reached top of loop after 1 iteration in "progressBag.java", line 104, col 4.

------------------------------------------------------------------------
    [0.023 s 9807824 bytes]  failed
```

**Original code with JML annotations**

```
/*@ requires srcOff >= 0; @*/
/*@ requires srcOff + length <=
             src.length; @*/
/*@ requires destOff >= 0; @*/
/*@ requires destOff + length <=
             dest.length; @*/
/*@ requires length >= 0; @*/
/*@ assignable dest[*]; @*/
/*@ ensures (\forall int m;
             m>=0 && m<length;
    dest[destOff+m] == src[srcOff+m]); @*/
private static void arraycopy(
            /*@ non_null @*/ int[] src,
                             int srcOff,
            /*@ non_null @*/ int[] dest,
                             int destOff,
                             int length) {
    //@ loop_invariant (i <= length);
    for( int i=0 ; i<=length; i++) {
      dest[destOff+i] = src[srcOff+i];
    }
}
```

**Solution to ESC/JAVA2's warnings**

```
/*@ requires srcOff >= 0; @*/
/*@ requires srcOff + length <=
             src.length; @*/
/*@ requires destOff >= 0; @*/
/*@ requires destOff + length <=
             dest.length; @*/
/*@ requires length >= 0; @*/
/*@ assignable dest[*]; @*/
/*@ ensures (\forall int m;
             m>=0 && m<length;
    dest[destOff+m] == src[srcOff+m]); @*/
private static void arraycopy(
            /*@ non_null @*/ int[] src,
                             int srcOff,
            /*@ non_null @*/ int[] dest,
                             int destOff,
                             int length) {
    //@ loop_invariant (i <= length);
    for( int i=0 ; i<length; i++) {
      dest[destOff+i] = src[srcOff+i];
    }
}
```

All the warnings can be solved by adjusting the condition in the loop in the same way done so far (`i < length`).

All the warnings raised by ESC/JAVA2 are correctly solved.

# Phase 3: Improving security aspects

Although ESC/JAVA2 doesn't recognize any more problems, the current version of `NewBag.java` has some potential flaws that derive from the lack of common and suggested Java programming rules.
Indeed, a Java class may still be vulnerable to attacks by untrusted code even if it is semantically and syntactically correct.

• Since we don't how this class will be used outside, I decided to maintain the default visibility (`package private`) and restrict the privileges only when needed. `Public` visibility is avoided in order to follow the *principle of least privilege*;

• I defined the `Bag` class as `final` in order to prevent malicious subclasses. In this way the inheritance/extension of the class is forbidden, increasing security;

• I defined both `n` and `contents` fields as `private` in order to prevent unauthorized changes and to preserve integrity. By default - without an access modifier -, a class member is accessible throughout the package in which it's declared: so it can be accessed by the class itself, other classes within the same package, but not outside of the package. Thanks to `private` visibility, the surface of possible attacks is reduced.
This change has some impact on other parts of the code. In `add(Bag b)` method we are accessing both fields of `b` through `b.contents` and `b.n`: this can't be done anymore due to `private` visibility. To solve this, methods `getN()` and `getContents()` are inserted: in this way we can access both private fields from outside.

**Solution to ESC/JAVA2's warnings**

```
/*@ ensures n == contents.length; @*/
void add(/*@ non_null @*/ Bag b) {
  int[] new_contents = new int[n + b.n];
  arraycopy(contents, 0,
            new_contents, 0, n);
  arraycopy(b.contents, 0,
            new_contents, n, b.n);
  contents = new_contents;
  n = this.n + b.n;
}



/* getN() AND getContents()
            ARE NOT PRESENT */
```

**Enhanced security solution**

```
/*@ ensures n == contents.length; @*/
final void add(/*@ non_null @*/ Bag b) {
  int b_n = b.getN();
  int[] b_contents = b.getContents();
  //@ assume b_n >= 0;
  //@ assume b_n <= b_contents.length;
  //@ assume b_contents != null;
  int size = n + b_n;
  //@ assert size >= 0;
  int[] new_contents = new int[size];
  arraycopy(this.contents, 0,
            new_contents, 0, n);
  arraycopy(b_contents, 0,
            new_contents, n, b_n);
  contents = new_contents;
  n = this.n + b_n;
}
int getN(){
  return this.n;
}
int[] getContents(){
  int[] copied_contents =
            new int[contents.length];
  arraycopy(contents, 0,
            copied_contents, 0,
            contents.length);
  //@ assert copied_contents.length
             == contents.length;
  return copied_contents;
}
```

Notice that, some JML annotations were inserted in order to let understand ESC/JAVA2 that `b.n` and `b.contents` have the same invariants of an object `Bag`.
Moreover, `getContents()` doesn't return a reference to `contents` but its copy (`copied_contents`): we're safe against undesirable aliasing;

• I defined the methods as `final`. Since a method can be changed by overriding, declaring it as `final` prevents this possibility.
According to *Java Language Specification* [1]: " Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding". For this reason, constructors are not marked as final;

• `Final` variables cannot be changed. In our class, both `n` and `contents` can't be `final` because are modified in all methods.
However an array marked as `final` has a slightly different meaning. An array is an object and the variable used to access it is a reference (in our case, `contents`). If that reference is marked as final then the variable will always refer to the same array *but* we are allowed to change its elements.
Since in our methods the reference is changed many times (`contents = new_contents`), we can't mark `contents` as `final`;

• The class doesn't suffer from any undesirable aliasing. We are safe with respect to *import reference aliasing problem* since we never assign directly something coming from outside the class to `contents` - we use `arraycopy`. We are also safe with respect to *export reference aliasing problem* since we never directly return `contents` to outside - again, we use `arraycopy`;

• Regardless all these changes, through reflection we can still have access to private members and methods. The use of reflection is highly discouraged in order to prevent data disclosure. If we want to disable reflection we have to build an ad-hoc *security manager*.
The JVM has security mechanisms built into it that allow you to define restrictions to code through a Java security policy file. The Java security manager uses the Java security policy file to enforce a set of permissions granted to classes. The permissions allow specified classes running in that instance of the JVM to permit or not permit certain runtime operations. If we enable the Java security manager but do not specify a security policy file, the Java security manager uses the default security policies. Using this mechanism, we can inject a policy that negates the usage of reflection for accessing private members (a call to `setAccessible(true)` is not possible).

# Conclusions and Reflections

The assignment suggests some questions to answer as a reflection about the usage of the tool:

1. *In the end, do you think that you found all the problems and that the code is correct?*
2. *Can you think of ways in which the tool or the specification language could be improved?*
3. *Instead of the tool we used, can you think of other ways (formal or informal, tool-supported or not) to find the problems that the tool found? If so, would these alternatives: (i) find fewer problems, the same, or more? (ii) find problems sooner or later than the current approach? (iii) require more work or less work?*

ESC/JAVA2 is **not complete** and **not sound**.
Since it's not complete, the tool may produce warnings about correct programs. The uncompleteness derive from Simplify's limited reasoning capabilities (arithmetic, quantifiers).
Since it's unsound, the tool may fail to produce warnings about incorrect program. The sources of unsoundness are multiples:

- Loops are handled by a not completely true unrolling. Indeed, let's compare what a faithful loop unrolling should do and what the ESC/JAVA2's default unrolling does:

$$\{P\}\ if(e)\ \{c;\ if(e)\ \{c;\ while\ (e)\ c\}\}\ \{Q\} \tag{1}$$

$$\{P\}\ if(e)\ \{c;\ if(e)\ \{\ assume\ false;\ \}\}\ \{Q\} \tag{2}$$

  where $P$ and $Q$ are precondition and postcondition.
  (1) is the correct one, (2) is the one used by the tool.
  Notice how not unrolled execution of loop is replaced by "*assume false*" - this means that from false, everything can be concluded. No more verification takes place in this branch.
  This is obviously wrong;

- JML annotation `assume` adds unverified knowledge to the tool that might be not correct;
- Object invariants are not verified on all existing objects;
- The way how modular arithmetic works (i.e. if `n+1` occurs and there could be an arithmetic overflow, how the tool can be sure about which relation between `n+1>n` and `n+1<<n` is true?);

As it can be seen, the tool could be improved in different ways but it remains extremely hard to properly inject the intent of the programmer inside the code, even if we can take advantage of JML annotations and so reducing false positives.
Despite this, ESC/JAVA2 can be extremely useful for detecting many programming errors like array index bound violations, reference to objects that may be null, etc.

Although ESC/JAVA2 doesn't show any warning, the final version of the code is *neither* completely correct *nor* safe.
Let's consider the case in which a malicious user decides to use the corrected version of `newBag.java` class discussed until now. He's obviously able to call constructors and methods with any parameter that he wants.[2]
For example, he can make a call to `Bag(int[] input)` constructor by passing an array that is `null`.
Even though this possibility is semantically forbidden thanks to JML annotations, these don't prevent a real malicious call from the outside.
In order to make the code secure also in real scenarios, we have to modify the class transforming most of the JML annotations (generally the ones related to `requires` statements) in real code.

For this reason, the report will be delivered with two Java classes:

- `NewBagMatteoMariani.java`, that is the one described during this report, composed by JML annotations and partial modifications of the code;
- `NewBagSafeMatteoMariani.java`, that is a version in which some JML annotations are transformed into real Java code.

There are many other tools and techniques that have been developed for automatically finding bugs by analyzing source code or intermediate code statically (at compile time).
One of the most important is *FindBugs*, a bug pattern detector for Java. FindBugs uses a series of ad-hoc techniques designed to balance precision, efficiency and usability. One of the main technique that FindBugs uses is to syntactically match source code with known suspicious programming practice. In some cases, this tool also uses dataflow analysis to check for bugs.
According to "*A Comparison of Bug Finding Tools for Java*" [2]: " Without user annotations a tool like ESC/Java2 that is still unsound yet much closer to verification produces even more warnings than JLint, PMD, and FindBugs".
This means that JML annotations represent a real need rather than something merely additional.
To conclude, it's interesting to see which categories of bugs different tools can spot:

| **Bug Category** | **Example** | ESC/Java | FindBugs | JLint | PMD |
|---|---|---|---|---|---|
| General | Null dereference | √* | √* | √* | √ |
| Concurrency | Possible deadlock | √* | √ | √* | √ |
| Exceptions | Possible unexpected exception | √* | | | |
| Array | Length may be less than zero | √ | | √* | |
| Mathematics | Division by zero | √* | | √ | |
| Conditional, loop | Unreachable code due to constant guard | | √ | | √* |
| String | Checking equality using == or != | | √ | √* | √ |
| Object overriding | Equal objects must have equal hashcodes | | √* | √* | √* |
| I/O stream | Stream not closed on all paths | | √* | | |
| Unused or duplicate statement | Unused local variable | | √ | | √* |
| Design | Should be a static inner class | | √* | | |
| Unnecessary statement | Unnecessary return statement | | | | √* |

√ - tool checks for bugs in this category  * - tool checks for this specific example

There is no tool that entirely wins over the others: they should be used according to what a programmer wants to achieve.

[2]The only constraint that he must follow is related to the types of the inputs.

# References

[1] *Java Language Specification*, Oracle
[https://docs.oracle.com/javase/specs/jls/se11/html/jls-8.html#jls-8.8]

[2] *A Comparison of Bug Finding Tools for Java*, University of Maryland
[https://www.cs.tufts.edu/~jfoster/papers/issre04.pdf]