

---

**A Process for the Specification of Core JDK Classes**

**Ralph Hyland - 07270861**

---

**MSc. Advanced Software Engineering**

**Supervisor: Dr Joe Kiniry**



UCD School of Computer Science and Informatics  
College of Engineering Mathematical and Physical Sciences  
University College Dublin

April 15, 2010

## Acknowledgements

My sincere thanks go to my supervisor Dr Joe Kiniry for all his time, patience, encouragement and helpful feedback throughout the months of this work. He was always open and approachable and made himself available for meetings at short notice.

I would also like to thank my friends and family for all their encouragement and support that they gave me over the past two years.

Finally, I'd like to thank my fiancée Gayle who put up with me for the past two years. She gave me constant encouragement and praise throughout and without her the task of writing this paper would have been a number of times more difficult.

This thesis is dedicated to my daughter Casey and the memory of my late mother (Betty) and father (John).

**Abstract**

New JML specifications for core JDK classes (e.g., `java.lang.String`) are typically written on-demand, as they are necessary for the testing and verification of new programs that exercise previously unused portions of the (very large) Java platform API. In the past, such specifications were written by experts who essentially translated Javadocs English into JML.

Unfortunately, such specifications are rarely generally useful because (a) they do not target a particular use-case (e.g., run-time checking versus verification), (b) they are not rigorously tested in any way, and (c) they are based upon erroneous data in the first place (i.e., Javadocs). Recently a new specification writing process has been introduced by Dr. David Cok. This process involves writing new specifications and complementary specification-centric unit tests that target a particular use-case. The purpose of this project is to evaluate and extend this process by incorporating existing available comprehensive unit tests suites, like those written by Sun Microsystems to certify new implementations of the Java programming language.

## Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Document Layout . . . . .	8
<b>2</b>	<b>Key Elements</b>	<b>9</b>
2.1	JCK Unit Tests . . . . .	10
2.2	An Introductory Example . . . . .	10
<b>3</b>	<b>Tools Overview</b>	<b>13</b>
3.1	Design By Contract . . . . .	13
3.1.1	What is it? . . . . .	13
3.1.2	The Contract . . . . .	13
3.2	JML . . . . .	15
3.2.1	What is it? . . . . .	15
3.2.2	Why is it used? . . . . .	15
3.2.3	How is it used? . . . . .	15
3.2.4	Tool Support . . . . .	16
3.2.5	What is its role in this project? . . . . .	17
3.3	ESC/Java2 . . . . .	17
3.3.1	What is it? . . . . .	17
3.3.2	Why is it used? . . . . .	17
3.3.3	Tool Support . . . . .	19
3.3.4	What is its role in this project? . . . . .	20
<b>4</b>	<b>Working with JML and ESC/Java2</b>	<b>21</b>
4.1	The Sample Application: Moosikbox . . . . .	21
4.2	The Account . . . . .	22
4.2.1	Write a method - <code>getBalance()</code> . . . . .	22
4.2.2	Write a test . . . . .	23
4.3	Adding specifications to <code>getBalance</code> . . . . .	23
4.3.1	<code>public normal_behavior</code> . . . . .	27
4.3.2	The <code>instance</code> keyword . . . . .	27
4.3.3	The <code>model</code> modifier . . . . .	27
4.3.4	The <code>invariant</code> keyword . . . . .	27
4.3.5	The <code>represents</code> keyword . . . . .	28
4.3.6	The <code>AccountImpl</code> specifications . . . . .	28
4.4	Re-running the tests for <code>getBalance</code> . . . . .	28
4.5	Adding specifications to the <code>credit</code> method. . . . .	30
4.5.1	The <i>normal</i> path . . . . .	31
4.5.2	The <code>\old</code> evaluator . . . . .	31

4.5.3	The <i>exceptional</i> path . . . . .	31
4.5.4	The signals clause . . . . .	32
<b>5</b>	<b>Verification of JDK classes</b>	<b>33</b>
5.1	Selecting the classes . . . . .	33
5.2	Set up a project . . . . .	33
5.3	Target class: ResourceBundle . . . . .	33
5.3.1	Run ESC/Java2 against the tests pre JML specification . . . . .	34
5.3.2	Writing JML specifications for the <code>getString(String key)</code> method . . . . .	34
5.4	Target class: PrintStream . . . . .	36
5.4.1	The existing specifications . . . . .	36
5.4.2	Running ESC/Java2 against the tests pre JML specification . . . . .	37
5.5	Target class: Stack . . . . .	39
5.5.1	Following the Javadocs . . . . .	39
5.5.2	Writing specifications for Stack . . . . .	40
5.5.3	Running ESC/Java2 against the Stack tests . . . . .	42
<b>6</b>	<b>Future Work</b>	<b>43</b>
6.1	Continuing the process . . . . .	43
6.2	Narrowing the gap . . . . .	43
6.3	OpenJML . . . . .	43
6.4	New JML Annotation . . . . .	43
<b>7</b>	<b>Conclusion</b>	<b>45</b>
7.1	Observations from writing specifications. . . . .	45
7.1.1	Combination of JML and ESC/Java2 . . . . .	45
7.1.2	The verification of JDK classes using JCK test classes, JML specifications and ESC/Java2 . . . . .	46
7.1.3	Standalone ESC/Java2 . . . . .	46
7.2	Findings . . . . .	46
	<b>Appendices</b>	<b>48</b>
<b>A</b>	<b>Moosikbox source code including specifications.</b>	<b>48</b>
A.1	The AccountImpl class . . . . .	49
A.2	The Artist interface . . . . .	50
A.3	The Broker interface . . . . .	51
A.4	The Collectable interface . . . . .	51
A.5	The Collector interface . . . . .	52
<b>A</b>	<b>JDK Target class signatures and Javadocs.</b>	<b>54</b>
A.1	Vector <code>addElement(Object obj)</code> . . . . .	54

A.2	Vector <code>addElement(Object obj)</code> JML specifications. . . . .	54
A.3	List <code>add(Object o)</code> . . . . .	55

## Listings

1	The Status class. . . . .	9
2	Our target class . . . . .	10
3	Our test class . . . . .	11
4	The Status. <code>failed(String)</code> static method. . . . .	11
5	The <code>calculate</code> method before specification. . . . .	18
6	Testing the Example class. . . . .	18
7	The <code>calculate</code> method with a single precondition. . . . .	18
8	Check with <code>jml-unix</code> and compile with <code>jmlc-unix</code> . . . . .	19
9	Execute with <code>jmlrac-unix</code> . . . . .	19
10	AccountImpl with only <code>getBalance()</code> shown. . . . .	22
11	The <code>assertEquals</code> utility method . . . . .	23
12	The unit test for <code>getBalance</code> . . . . .	24
13	The ESC/Java2 results after checking AccountImplSpecTest. . . . .	25
14	The JML specified <code>getBalance</code> method. . . . .	26
15	The AccountImpl with constructor and field specifications. . . . .	29
16	ESC/Java2 successful output after checking the AccountImplSpecTest class	29
17	The <code>credit</code> method before JML specification. . . . .	30
18	The test for the <code>credit</code> method. . . . .	32
19	The fully specified <code>credit</code> method. . . . .	32
20	The Javadocs for the <code>getString(String key)</code> method. . . . .	35
21	The specified <code>getString(String key)</code> method. . . . .	35
22	ESC/Java2 successful results for the <code>ResourceBundle0004()</code> test method. .	36
23	The specified <code>PrintStream(OutputStream os)</code> constructor. . . . .	37
24	Excerpt from the <code>ctorTest.java</code> JCK test class. . . . .	37
25	The public <code>Object push(Object item)</code> signature and Javadocs. . . . .	39
26	The proposed new JML annotation <code>@uses</code> . . . . .	44
27	The Account interface with JML specification. . . . .	48
28	The AccountImpl class. . . . .	49
29	The Artist interface. . . . .	50
30	The Broker interface. . . . .	51
31	The Collectable interface. . . . .	51
32	The Collector interface. . . . .	52
33	The AssertUtils helper class for testing. . . . .	53
34	Vector's <code>addElement(Object obj)</code> method signature and Javadocs. . . . .	54
35	Vector's <code>addElement(Object obj)</code> JML specifications. . . . .	54

36	List's <code>add(Object o)</code> signature and Javadocs. . . . .	55
37	The JML specifications for the Stack class. . . . .	56
38	The ResourceBundle0004 test in the <code>castingGetterTests</code> class from the JCK. . . . .	57

## 1 Introduction

To understand complex APIs like those available for Java, (e.g., the core classes in the `java.lang`, `java.io`, `java.util` packages etc.), one needs a precise description of the API's behavior. While natural language documentation, like that found in Sun Microsystems's Javadocs for JDK 5, has improved API documentation over the past ten years, such "specifications" are vague, imprecise, and error-prone.

The Java Modeling Language (JML) is the de facto standard for writing precise specifications of Java programs. JML is used by many courses around the world (including several at UCD), applied at several companies, and is supported by many tools like ESC/Java2, Daikon, KeY, AutoJML, JForge, CoJava, and Modern Jass<sup>1</sup>. JML is an extension of Java that permits one to write assertions, like invariants and pre- and post-conditions, about Java classes and interfaces. Various tools support using such specifications to, among other things, generate runtime assertion checks, generate Javadoc-like documentation, generate method-level unit tests, and check that method implementations fulfill their specifications.

The comprehensive unit test suite released by Sun Microsystems is called Java(TM) Compatibility Kit (JCK). Sun made this widely available under read-only terms to assist the Java community in gaining a better understanding of how testing of such a framework is accomplished.

The work put forward in this thesis will leverage the JCK framework in an attempt to verify actual implementations of selected Java classes (with a view to eventually verifying the complete Java API). Java 1.4 is the source version chosen for this project as it is compatible with the static verification tool ESC/Java2 which, is heavily used throughout the project. Although slightly behind the current version, this is not deemed to be a weakness in this process. The intention of this paper is to describe the process of verifying actual implementations of Java classes using the combination of JML and ESC/Java2 and in doing so document the issues and findings that occur along the way.

There are several key parts to the process which are summarised in Section 2.

*Note that throughout the remainder of this document, mention of “**the process**” refers to the collective steps undertaken to achieve the end goal i.e. a process for the specification of core JDK classes.*

<sup>1</sup><http://www.eecs.ucf.edu/~leavens/JML//download.shtml>



## 1.1 Document Layout

Section 2 presents an overview of the key elements of this project. This includes a simple example that introduces the reader to the important concepts within the paper. A brief description of the tools used over the course of this thesis are outlined in Section 3. In Section 4, an example project is built from the ground up in order to demonstrate the process using JML and ESC/Java2 . This was the sample project undertaken by the author whilst starting out on this project. This sample project doesn't focus on the JDK classes at all, it is used for demonstrating the process of specification/contract writing and the things learned along the way. Section 5 contains the bulk of the content for this paper and sees the chosen JDK classes being specified and tested in addition to some interesting findings. Section 5 also contains some brief information on setting up a project to carry out this process. Section 6 contains further work while Section 7 has the conclusions. Appendix A includes the *some* (mainly the classes referenced directly in the document) of the sample application Moosikbox code and Appendix A includes some Javadoc-annotated methods and JML specified classes and methods. The full source code can be found on the accompanying CD.

## 2 Key Elements

1. Selecting the target JDK class to specify is the first step in the process. One caveat when selecting the target class is to ensure that the JCK includes unit tests related to it.
2. Once selected, this class (or group of classes) is specified with JML. These JML specifications **must** be based solely on the exposed Javadocs of the class and **not** the implementation of the class. The specifications verify class invariants and pre/post conditions of every (important) method.
3. The next step is to determine the related unit tests for the target class. This is a simple matter of searching the JCK release until the test classes related to the target class are found.
4. The JCK contains a class called `com.sun.javatest.Status` which is used extensively throughout the framework. This class embodies the result of a test i.e. did it pass, fail, not run and why. Although the JCK licensing agreement prevents running the tests, statically checking the code using a tool such as ESC/Java2 is permitted. Here are the specifications of the two most important methods from the `Status`<sup>2</sup> class:

```

1  //@ public normal_behavior
2  //@ ensures \fresh(\result);
3  //@ ensures \result.isPassed();
4  //@ ensures stringEquals(\result.getReason(), reason);
5  public static /*@ pure non-null @*/ Status passed(/*@ nullable @*/
      String reason);
6
7  //@ public normal_behavior
8  //@ requires false;
9  //@ ensures \fresh(\result);
10 //@ ensures \result.isFailed();
11 //@ ensures stringEquals(\result.getReason(), reason);
12 public static /*@ pure non-null @*/ Status failed(/*@ nullable @*/
      String reason);

```

Listing 1: The Status class.

The specifications of the `Status` class, particularly the `'failed("")'` method, are the main ingredient to the overall process and will be explained in the following section.

<sup>2</sup>The `Status` class was specified with JML by Dr. Joe Kiniry (<http://www.kindsoftware.com/about/people/jrk.html>)

## 2.1 JCK Unit Tests

The unit tests contained in the JCK test framework are written using a standard pattern: (a) set up the test, (b) carry out some actions on the class under test, and (c) respond positively or negatively as a result of the actions carried out in (b). To accomplish those positive or negative responses, the JCK developers have written a class called `Status` in the package `javasoft.sqe.javatest`. A positive response is created by invoking the static method `Status.passed(String)` and a negative response is created by invoking the static method `Status.failed(String)`.

## 2.2 An Introductory Example

As always, the best way to portray an idea is with a simple example. The following (fictitious) example uses a target class called `Example`. A thorough comprehension of this simplified example is key to understanding the work described in this thesis.

As was stated in point number 1 of 'Key Elements' above, JML specifications are added to the target class.

```
1 public class Example {
2
3     //@ requires 0 < length;
4     //@ requires 0 < width;
5     //@ ensures \result == (length * width);
6     int area(final int length, final int width) {
7         return length * width;
8     }
9 }
```

Listing 2: Our target class

The `area(final int, final int)` method above is specified using JML (line numbers 3, 4, and 5). This defines the method's contract. Lines 3 and 4 state that this method will only accept positive integer values for the parameters `length` and `width` respectively. Line 5 states that the result of the `area` method will equal the `length` parameter multiplied by the `width` parameter<sup>3</sup>.

<sup>3</sup>Notice that both parameters are marked with the `final` modifier. It is good coding practice to use this modifier when parameter values will not be modified within the method [Blo08]. In addition to informing the compiler of our wishes it also leaves the users of the API and other tools such as ESC/Java2 in no doubt of the intentions regarding this data. Section 4 describes how JML highlights data for modification.

Next, `ExampleTest.java` is shown, which contains a unit test for `Example.java` above.

```

1 import com.sun.javatest.Status;
2
3 public class ExampleTest {
4
5     public void testArea() {
6         Example example = new Example();
7         final int length = 2;
8         final int width = 2;
9         if (example.area(length, width) == (length * width)) {
10            Status.passed("Got the correct result");
11        } else {
12            Status.failed("Incorrect result");
13        }
14    }
15 }

```

Listing 3: Our test class

Next, ESC/Java2 is executed against `ExampleTest.java`, see Section 3.3 for more details on this. On line 30 of the `testArea` method, `example.area` is invoked with parameters `length` and `width` having a value of 2. These values will satisfy the preconditions (the JML 'requires' clause) of the `area` method in Listing 2. Line 9 compares the result of the `area` method against `length` times `width`. Given that the contract of the `area` method is adhered to i.e. its pre and postconditions are fulfilled, the comparison is sure to result in true and subsequently `Status.passed(String)` will be invoked on line 10.

*Note, this process assumes the unit test (including its data) to be correct.*

`Status.failed(String)` will never be invoked here because the postcondition of `area` will always hold true and therefore `Status.passed(String)` is always invoked. In the specification of `Status.failed(String)`, shown below in Listing 4 for convenience, notice the precondition (JML requires false) on line 2.

```

1 //@ public normal_behavior
2 //@ requires false;
3 //@ ensures \fresh(\result);
4 //@ ensures \result.isFailed();
5 //@ ensures stringEquals(\result.getReason(), reason);
6 public static /*@ pure non_null @*/ Status failed(/*@ nullable @*/
    String reason);

```

---

Listing 4: The `Status.failed(String)` static method.

The precondition `requires false` actually means that this method can never be legally invoked. Therefore, it can be stated with great certainty that if `Status.failed(String)` *is* invoked, there are problems with the specification of the method under test i.e. `area` in this example. A problem with the specification highlights one of three things: (1) The specifications written for the target class are incorrect, (2) the information available i.e. Javadocs, when specifying the method, is inadequate or not explicit enough in one or more areas, or (3) the software does not do what is expected of it i.e. it contains a bug. Section 5 presents some inadequacies in the JDK Javadocs which eludes to a Javadoc bug.

## 3 Tools Overview

### 3.1 Design By Contract

#### 3.1.1 What is it?

Design By Contract (DBC) is a term created by Bertrand Meyer and is a built in feature of his Eiffel programming language. DBC is a software development technique based on formal specifications and Hoare logic[[Hoa83](#)] that enables one to build the notion of a contract into software modules.

DBC offers benefits to both the client and supplier of a class. It puts obligations on both parties and in return provides benefits to both parties. Clients know that once they supply valid values i.e. they meet the preconditions of a method, they are guaranteed to get a certain output (as stated by the postcondition). The supplier of the class will not guarantee anything unless the preconditions of a method are met. Class invariants are the properties that the client and supplier can be sure of and that hold true before and after each method call.

Contracts that can be manipulated by tools are most beneficial. This allows the specifications to evolve with software as opposed to becoming out of date like a lot of specification comments do e.g. Javadocs. Tool support provides abilities like static verification, automatic unit test generation, and compile time warnings.

#### 3.1.2 The Contract

A contract is a term familiar to most. For example, mobile phone subscribers enter into a contract with an operator (supplier) when signing up for a bill paying phone. Such a contract might say that the customer (client) must pay a basic fee of X amount per month. Using DBC terminology, this is a precondition of this contract. The contract may also state that the customer will receive 500 free SMS messages every month once the fee is paid. This is a post condition of the contract. The contract will also contain certain pieces of information that are guaranteed to remain true throughout the lifetime of the contract e.g. costs of a call to other mobile numbers on the same network will be X per minute. This is an invariants of the contract i.e. a piece of data that is guaranteed to remain true.

Design By Contract is expressed in terms of obligations and benefits[[LC05](#)]. The `area` method from `Example.java` in [Listing 2](#) requires clients to pass positive integer values as parameters. Such pre-conditions place an obligation on the client and at the same time benefit the supplier. Post conditions, on the other hand, put an obligation on the supplier of the class to deliver the guaranteed results which, in turn benefits the client. These two

seemingly simple clauses offer a powerful premise on which to design software. Applied correctly, DBC has many benefits including designation of 'blame' for problems with the software i.e. who is the bug assigned to; client or supplier? If the `area` method is invoked with one or more negative integers the supplier is under no obligation to guarantee anything about the behaviour of the `area` method. Any problems arising from such a situation can be traced back to the violation of the explicit pre-conditions on the method and therefore the client is responsible.

Software houses vary drastically in their strategy of documenting/specifying an API but here are three scenarios that broadly cover the full spectrum.

1. Software is released with incomplete, or nonexistent, API specifications. This leads to ambiguous software with regards to functionality and will inevitably cause problems for clients using the API.
2. Software is released with pre-/post-conditions and invariants outlined in the API specifications but **only** for normal execution paths. The supplier of the software will take no responsibility for malfunctioning software if the client fails to adhere to the pre-conditions specified. Although there is validity to the supplier's argument, the business relationship with the client may turn sour if this approach is taken when problems arise.
3. Software is released with comprehensive API specifications including how the software will react when preconditions are met **and** not met (exceptional cases). By writing such specifications the supplier makes it easy for clients to understand and debug certain problems if and when they arise. This style of software development involves defensive coding on the part of the class supplier. Such an approach is fairly common in industry and may suit most companies adequately. Such an approach also tends to leave boiler plate code throughout a repository, which over time becomes out-of-sync with the API documentation.

So what is the difference between writing API documentation and using DBC? One of the fundamental differences already eluded to above is that DBC typically uses tool readable specifications (as opposed to 'normal' english specifications of Javadocs). These provide the ability to compile the specifications into the executable code. This might be a feature only desired in development and is normally a configurable option. There are many benefits to detecting potential runtime errors early in the development process e.g. lead time, monetary value, company credibility, and perhaps (at the most extreme) the prevention of catastrophic events resulting from software malfunctions in mission critical systems.

The next section (Section 3.2) examines how Design By Contract is realised with JML and why JML is an integral part of the process used throughout this thesis.

## 3.2 JML

### 3.2.1 What is it?

A behavioral interface specification language (BISL) is a specification language that specifies both the syntactic interface of a module and its behaviour[LCC<sup>+</sup>05]. The Java Modelling Language is a BISL and is the de facto formal specification writing tool for the Java programming language. It can play the role of a Design By Contract language like Eiffel in addition to being the target of various tools such as runtime assertion checkers, unit test generation, formal verification, and theorem provers[LC05].

### 3.2.2 Why is it used?

JML provides a specification language that is written using various annotations and expressions that produce assertions about the code in question. These expressions, that have a syntax similar in structure to Java, provide the means of writing pre/post-conditions and class invariants (the three of which are assertions). Having a Java-like syntax makes JML appealing to Java community and means the learning curve involved is minimised[LCC<sup>+</sup>05]. As was stated above, JML is a language used by various tools and it is this feature that makes it a more attractive proposition than writing plain ad-hoc Javadocs which, are sometimes not even understandable to other humans.

### 3.2.3 How is it used?

JML annotations are applied in one of two ways. The simplest way is to write specifications directly above method signatures in the same place Javadocs are written. It is also possible to place the specifications in a separate file once a standard naming convention is adhered to[CKC08]. JML specifications can be compiled into the executable byte code. This can be achieved using the **jmlc** tool, which is akin to Java's **javac**. Once JML specifications are compiled into the byte code they can be executed at runtime using the **jmlrac** tool, which is equivalent to Java's **java** tool.

In the Design By Contract section (Section 3.1) above, one of the approaches to combat broken pre-conditions was for the class supplier to defensively code against invalid input. This is a widely used pattern in industry but is one that adds a lot of boiler plate code throughout a project i.e. code that has no business value. Using JML tools like **jmlc** and **jml** one can eliminate this kind of pattern from a codebase leaving concise code that expresses only the business functionality of each method.



Here are three very basic examples of JML that show the fundamentals of the language. More advanced assertion types will be shown later on in the paper when writing specifications for the example application and the JDK classes.

1. Preconditions

```
\\@ requires 0 < length;
\\@ requires 0 < width;
public int area(final int length, final int width);
```

2. Postconditions

```
\\@ ensures \result = (length * width);
public int area(final int length, final int width);
```

*The JML expression `\result` will have the same type and value as that of the method's return value.*

3. Invariants

*Properties that should hold true in all client-visible states* [LC05].

```
\\@ public invariant numberOfCalculations >= 0;
private int numberOfCalculations;
```

### 3.2.4 Tool Support

Tools useful for checking that JML annotated Java modules meet their specifications fall into two main categories: [CKLP06]:

1. Runtime assertion checking (RAC) tools

As the name suggests, these tools check the specifications at runtime. They report back special errors when violations of the specs occur. This is another idea that was made popular by the Eiffel language during the 1980s. The **jmlc** tool is the leading RAC tool in use for JML currently [CL02].

2. Static verification (SV) tools

The main (extended) static checker for JML is ESC/Java2 [KC05] of which a brief overview is given in Section 3.3. In static verification, logical techniques are used to prove, before runtime, that no violations of specifications will take place at runtime. The adjective “static” emphasises that verification happens by means of a static analysis of the code, i.e., without running it [CKLP06].

### 3.2.5 What is its role in this project?

JML is the foundation of this thesis and provides the specifications that enable static verification to be carried out on both the example application and the verification of the JDK classes and finally the JCK tests.

## 3.3 ESC/Java2

### 3.3.1 What is it?

ESC/Java2 is a tool for statically checking program specifications[CKC08]. ESC/Java2 extends the original work in ESC/Java [FLL<sup>+</sup>99]. Implementation notes about ESC/Java2 can be found here [CKC08]. The tool can be invoked against Java code that is, or is not, supplemented with JML specifications. ESC/Java2 relies internally on a configurable theorem prover (Simplify) to carry out the mathematical proofs about the code being statically checked. Although it is quite common to use ESC/Java2 in tandem with JML specifications, the tool is capable of statically checking 'plain' Java code i.e. code without JML specifications added. The use of such a tool is invaluable in the development process as it highlights errors early providing essential feedback to developers which promotes less defects, quicker release times, and a more stable end product.

### 3.3.2 Why is it used?

Apart from the reasons mentioned above, here is a very simple, but important, reason why ESC/Java2 is best used in tandem with JML.

The `calculate` method in Listing 5 contains two execution branches/paths. The `main` method in Listing 6 contains three tests that cover each execution path resulting in 100% test coverage for the `calculate` method. Compiling and executing (using the normal Java tools) the program produces successful results i.e. the program terminates cleanly and no exceptions are raised. The observant reader will notice the possible divide-by-zero error waiting to happen in the `calculate` method.

If the values of `x` and `y` are set to zero (line 10 in the `main` method) then a `java.lang.ArithmeticException` is the outcome when the `main` method is executed.

To avoid the divide-by-zero error a number of steps are required. A JML `requires` clause is added to the `calculate` method and then the JML checker (`jml`) is ran. The checker ensures the well-formedness of the specification syntax[KC05]. Next, the class is compiled using the JML compiler (`jmlc`) which, is the JML equivalent of the `javac` tool. In addition to compiling the java code (as usual), this tool translates the JML specifications into

code then compiles and combines the results with the original class. In this example, the compilation produces no errors or warnings.

```

1 public int calculate(int x, int y) {
2     int z = 0;
3     if (x < y || x == y) {
4         z = x / y;
5     } else {
6         z = y / x;
7     }
8     System.out.println("z is " + z);
9     return z;
10 }

```

Listing 5: The `calculate` method before specification.

```

1 public static void main(String[] args) {
2     Example ex = new Example();
3     // Test the (x < y) branch
4     int x = 1; int y = 2;
5     ex.calculate(x, y);
6     // Test the (y < x) branch
7     x = 2; y = 1;
8     ex.calculate(x, y);
9     // Test the (x == y) branch
10    x = 1; y = 1;
11    ex.calculate(x, y);
12 }

```

Listing 6: Testing the `Example` class.

In order for JML to detect and report the potential error, the compiled class must be executed using the JML equivalent of the `java` tool - `jmlrac`. This tool will detect the potential specification error and terminate the program with a pre-condition exception. See the output in Listing 9.

```

1 // @ requires 0 < y;
2 public int calculate(int x, int y) {...}

```

Listing 7: The `calculate` method with a single precondition.

Note that the JML project contains the tools mentioned above (`jml`, `jmlc`, and `jmlrac`), each tailored to suit various operating systems. The unix version of those tools are used in this example (as is evident by the '-unix' postfix). To run these tools, `JML_HOME` is added to the user path. One final thing to note about the examples below is the '-Q' switch

that is passed as an option to the `jml-unix` and `jmlc-unix` tools. This option suppresses the (verbose) output from the tool.

```

1 ralphhyland$ jml-unix -Q com/hyland/Example.java
2 ralphhyland$
3 ralphhyland$ jmlc-unix -Q com/hyland/Example.java
4 ralphhyland$

```

Listing 8: Check with `jml-unix` and compile with `jmlc-unix`.

```

1 ralphhyland$ jmlrac-unix com/hyland/Example
2 z is 0
3 Exception in thread "main" org.jmlspecs.jmlrac.runtime.
   JMLInternalPreconditionError: by method Example.calculate
4   at com.hyland.moosikbox.impl.Example.main(Example.java:512)

```

Listing 9: Execute with `jmlrac-unix`.

Listing 9 proves that a RAC tool like JML (whether employed via the command line or an integrated development environment plugin) undoubtedly provides an important line of defense to software developers. The drawback with this approach is that such errors are uncovered at runtime (granted, that is most likely to be runtime in a development environment).

Using a SV tool like ESC/Java2 in tandem with JML is an even more powerful proposition. As is the case with the majority of tools, ESC/Java2 can be used on the command line or through an IDE plugin. An eclipse plugin called Mobius, developed by Joe Kiniry's team, is used throughout this thesis<sup>4</sup>.

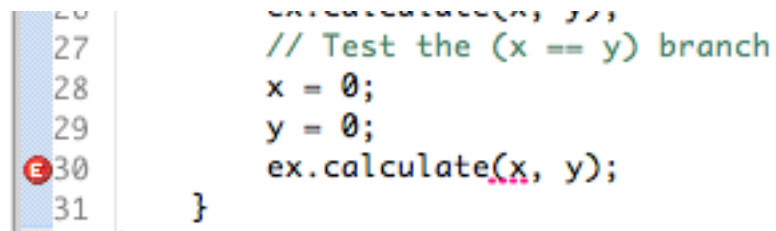
Using the example from Listing 7 and running the ESC/Java2 checker against the class will pin-point the potential error statically i.e. before the code is executed. This is a better solution that saves development time and effort. See the red marker 'E' on the left hand side of Figure 1. A warning message 'Precondition possibly not established' is emitted from the ESC/Java2 tool's command line output also.

In fact, ESC/Java2 will detect the possible divide-by-zero error without using JML specifications at all.

### 3.3.3 Tool Support

As mentioned above, there is an eclipse plugin called Mobius available for ESC/Java2 that provides configurable checking levels, theorem provers, and environment settings such as

<sup>4</sup><http://kind.ucd.ie/products/opensource/Mobius/>

A screenshot of a code editor window. On the left side, there is a vertical line of line numbers: 27, 28, 29, 30, and 31. Line 30 is highlighted in blue and has a red circle with a white 'E' next to it, indicating an error. The code on the right side of the editor is as follows:

```
calculate(x, y);  
// Test the (x == y) branch  
x = 0;  
y = 0;  
ex.calculate(x, y);  
}
```

The code is color-coded: comments are green, literals like '0' are blue, and identifiers like 'ex' and 'calculate' are black. The closing brace on line 31 is also black.

Figure 1: ESC/Java2 marks a potential error.

whether to run the tool automatically in the background.

### 3.3.4 What is its role in this project?

ESC/Java2's main purpose in this project is to statically verify test classes from the JCK against Java classes from the JDK.

## 4 Working with JML and ESC/Java2

A sample application is used to demonstrate each part of the process and to explain how JML and ESC/Java2 contribute to accomplishing the verification goals.

### 4.1 The Sample Application: Moosikbox

The Moosikbox application models a market type system that allows music collectors to buy and sell collectables through an intermediary broker. It is not a fully fledged system. Instead, it is intended to capture the essence of an online broker application for music lovers. The Discogs<sup>5</sup> website is the motivation behind this sample application.

A particular goal of JML is to document existing code [LCC<sup>+</sup>05]. Typically, JML demonstrations are shown by using full or partial concrete classes [LC05] [CKLP06] [KC05]. While this approach is satisfactory for the demonstration of JML syntax it does not explicitly show how JML is combined with the development process from the ground up. The Moosikbox application demonstrates the combination of writing professional software in tandem with applying JML specifications.

The core of the system is made up of the following abstractions:

1. An Account. This has a monetary balance and can be debited and credited.
2. An Artist. This has a name and a label.
3. A Broker. This handles the sale of collectable items between buyers and sellers.
4. A Collectable. This has a reference number and price and is associated with an artist.
5. A Collector. A buyer or seller in the system.

The Account and Collector abstractions are chosen here to demonstrate the application of JML specifications and are the target of the ESC/Java2 checker once the specifications are applied.

So the approach is:

1. Write a method.
2. Write a test. Each test will harness the `Status` class mentioned in Section 2.
3. Run ESC/Java2 against the test. This will catch any specification inaccuracies.
4. Write specifications on the method to make the test pass.

<sup>5</sup><http://www.discogs.com>

Note that in this process, JML specifications are applied to interfaces first, at the most abstract level, and then to the implementation classes. It is also possible to place the JML specifications in separate files once the suffix of those files are ones used by JML to find files. JML will scan the project for files with various suffixes such as '.refines-java', '.jml', and '.spec'.

## 4.2 The Account

The Account abstraction contains (among others) a method called `getBalance()`.

### 4.2.1 Write a method - `getBalance()`

This shows a simple query method that returns a literal value. Notice the Javadocs state that the current balance, that is returned, should never be negative. This will later be extrapolated into a JML invariant later on. The process requires close reading of the Javadocs in order to decide how the specifications will be written if at all.

```
1 package com.hyland.moosikbox.impl;
2
3 import com.hyland.moosikbox.Account;
4
5 /**
6  * Default implementation of the {@code Account} interface.
7  *
8  * @author ralphhyland
9  */
10 public class AccountImpl implements Account {
11     private long balance = 0;
12     /**
13      * Retrieve the current balance.
14      * @return The current balance which should never be negative.
15      */
16     public long getBalance() {
17         return balance;
18     }
19     ....
20 }
```

Listing 10: AccountImpl with only `getBalance()` shown.

### 4.2.2 Write a test

Some setup is required first. While writing tests for this sample application some duplicate code was refactored into a utility class called `AssertUtil`. Most tests use the static method `assertEquals(final long, final long)` shown in Listing 11. The method has a familiar API to those used in xUnit APIs e.g. JUnit<sup>6</sup>.

```
1  //@ public normal_behavior
2  //@ requires actual == expected;
3  public static /*@ pure @*/ void assertEquals(final long actual, final
4     long expected) {
5     if (actual == expected) {
6         Status.passed("");
7     }
8     Status.failed("");
9 }
```

Listing 11: The assertEquals utility method

The unit test for `getBalance` is quite simple as shown in Listing 12. It actually tests the one arg constructor of `AccountImpl` in addition to `getBalance()` as this is the easiest way to demonstrate an `Account` that has some balance value greater than zero (the default constructor initialises the balance to zero).

Listing 13 shows partial output from ESC/Java2 following its analysis of the `AccountImplSpecTest` class. Both tests fail with the warning “Precondition possibly not established (Pre)”. This precondition failure is on the `AssertUtils.assertEquals` method. Listing 33 shows that the method in question contains a JML pre-condition.

So, the problem here is that the `AssertUtils.assertEquals` method expects the `actual` and `expected` parameter values to be equal but the test class uses the result of the invocation to `getBalance` as a parameter (or parameters in `testInitialAccountBalance`) even though it is not possible to make any guarantees (yet) about the return value from the `getBalance` method.

## 4.3 Adding specifications to getBalance

The javadocs for this method state that the return value should never be negative. This is a basic post condition that can be specified using an `ensures` clause. Another interesting property of such an accessor method like `getBalance` is that it should have no side-effects

<sup>6</sup><http://www.junit.org>



when invoked i.e. it should be idempotent. JML highlights such a notion with the keyword **pure**. Multiple invocations of **getBalance** should yield the same result provided that none of the non-idempotent methods (that may modify the state of the **Account**) are invoked in the meantime.

```
1 package com.hyland.moosikbox.impl;
2
3 public class AccountImplSpecTest {
4
5     public void testBalanceIsCorrectAfterInitialisation() {
6         Account account = new AccountImpl(10L);
7         AssertUtil.assertEquals(account.getBalance(), 10L);
8     }
9
10    public void testInitialAccountBalances() {
11        Account account1 = new AccountImpl(10L);
12        Account account2 = new AccountImpl(account1.getBalance());
13        AssertUtil.assertEquals(account1.getBalance(), account2.getBalance
14                                ());
15    }
```

Listing 12: The unit test for **getBalance**.

The **GET** and **HEAD** methods of the **HTTP** API are examples of idempotent methods although **GET** can be (and is) used to carry out dynamic behaviour in web applications<sup>7</sup>.

The important point to note is that a client of **getBalance** does not request (nor expect) an invocation of that method to have side-effects. By the same token, the class supplier does not intend this method to have side-effects. It must be considered safe (side-effect free) and used purely for retrieval purposes. JML facilitates specifications to portray such a property with the **pure** modifier. In Meyer's Eiffel language, these methods are referred to as **query** methods[Mey92]. A client of the **getBalance** method is not accountable for any side-effects caused by invoking the method. The inverse of this means the supplier of the class is obligated to ensure the method is side-effect free.

Back to the specification of **getBalance**. It is evident from the **AccountImpl** class that the account's balance is represented by a **long** type variable. This is purely an implementation detail and as such could be changed at any time e.g. an implementation may decide to store the balance using a new class called **Balance**. It is wise to avoid mentioning implementation details within a specification[LKP07].

<sup>7</sup><http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>

Implementation source code is not always available when writing specifications so it is desirable to keep specifications (like code) at an abstract level and this is one reason why specifications are added to the Account interface first in this example. Writing a generic specification for the interface benefits all implementing classes as they inherit those specifications.

```

1 com.hyland.moosikbox.impl.AccountImplSpecTest :
  testBalanceIsCorrectAfterInitialisation () ...
2
3 /Users/ralphhyland/CollegeWork/masters/Thesis/workspace/workspace_jck/
  exportedproject/ralph/test/com/hyland/moosikbox/impl/
  AccountImplSpecTest.java:12: Warning: Precondition possibly not
  established (Pre)
4     AssertUtil.assertEquals(account.getBalance(), 10L);
5
6 MARKER /Users/ralphhyland/CollegeWork/masters/Thesis/workspace/
  workspace_jck/exportedproject/ralph/test/com/hyland/moosikbox/impl/
  AccountImplSpecTest.java 12 Warning: Precondition possibly not
  established (Pre)
7 Associated declaration is "/Users/ralphhyland/CollegeWork/masters/
  Thesis/workspace/workspace_jck/exportedproject/ralph/test/com/hyland
  /testutils/AssertUtil.java", line 11, col 6:
8 // @requires actual == expected;
9
10 com.hyland.moosikbox.impl.AccountImplSpecTest :
  testInitialAccountBalances () ...
11
12 /Users/ralphhyland/CollegeWork/masters/Thesis/workspace/workspace_jck/
  exportedproject/ralph/test/com/hyland/moosikbox/impl/
  AccountImplSpecTest.java:18: Warning: Precondition possibly not
  established (Pre)
13     AssertUtil.assertEquals(account1.getBalance(), account2.getBalance
  ...
14
15 MARKER /Users/ralphhyland/CollegeWork/masters/Thesis/workspace/
  workspace_jck/exportedproject/ralph/test/com/hyland/moosikbox/impl/
  AccountImplSpecTest.java 18 Warning: Precondition possibly not
  established (Pre)
16 Associated declaration is "/Users/ralphhyland/CollegeWork/masters/
  Thesis/workspace/workspace_jck/exportedproject/ralph/test/com/hyland
  /testutils/AssertUtil.java", line 11, col 6:
17 // @requires actual == expected;

```

Listing 13: The ESC/Java2 results after checking AccountImplSpecTest.

Writing specifications on interfaces poses various challenges. For example, in the `Account`, it is beneficial if the specification contains rules governing an account's balance representation. The problem is that the `Account` interface does not contain any concept of an actual balance (mainly due to the fact that interfaces are not designed to have implementation detail, and doing so would defeat their intended purpose i.e. to define an abstraction). How can the notion of a balance be represented on the `Account` interface without leaking implementation detail?

It would be nice to say something like “the return value from `getBalance` method will equal the internal representation of the account balance”. With the use of `model` fields, JML makes this possible without exposing or jeopardising the abstract nature of the interface. It is common then to associate the `model` field with some runtime value. In this situation the `getBalance` method is perfect as it is abstract and so gives nothing away about potential subclass detail. The association is made courtesy of the `represents` clause that, in essence, says the value of our model field is represented (or realised) by the `getBalance` method. Listing 14 shows the specification for the `getBalance` method in addition to the class level specifications (lines 1-4 inclusive) that were added to portray the model field and representation discussed.

```

1 // @ public instance model JMLDataGroup balanceState;
2 // @ public instance model long _balance; in balanceState;
3 // @ public instance invariant 0 <= _balance;
4 // @ private represents _balance = getBalance();
5
6 /**
7  * Retrieve the current balance.
8  * @return The current balance which should never be negative.
9  */
10 // @ public normal_behavior
11 // @ ensures 0 <= \result;
12 // @ ensures \result == _balance;
13 // @ pure @*/ long getBalance();

```

Listing 14: The JML specified `getBalance` method.

Listing 14 includes some new ideas not yet discussed. The following sub sections give a brief explanation of each concept. After that ESC/Java2 is re-run on the test class again and the results are evaluated.

### 4.3.1 public normal\_behavior

Line 10 tells us that the normal behavior for this method is being outlined i.e. the flow of events that happen when the method returns normally. This is a heavyweight specification case. Such specification cases are useful when one wants to give a relatively complete specification, especially for the purposes of formal verification[LCC<sup>+</sup>05]. Heavyweight specification cases are introduced with a `normal_behavior` or `exceptional_behavior` keyword[CKLP06]. Heavyweight specification cases can contain exceptional behavior, which is explained more in Section 5.

Lightweight specification cases are applied when partial method behaviour is required or when the specification will only be used for runtime assertion checking or documentation and these are indicated by the absence of a behaviour keyword (like `normal_behaviour`).

### 4.3.2 The instance keyword

On line 1 an `instance model` field is created of the type `JMLDataGroup`, with the name `balanceState`. The `instance` keyword has the same connotations as it does in Java terminology i.e. this field will exist for each instance of this interface (or more precisely, each implementation of this interface)[Blo08]. The `JMLDataGroup` type is a JML specific class that represents a data group. It provides the user with a further abstraction layer. For example, one might add numerous model fields to a data group and then use the data group in assignable clauses of a specification. This abstracts away the actual fields that are assignable. It is sufficient to say that anything within the data group may be assigned to at some point during the method call. Model fields may be added or removed from the group without affecting the assignable clause in question plus specifications are shortened as a single assignable clause could potentially replace many individual clauses.

### 4.3.3 The model modifier

Model fields are only visible at the specification level. Model fields are used in JML as a way of abstracting away implementation details.

### 4.3.4 The invariant keyword

The invariants of the program are those pieces of data that hold true in all client visible states of the program i.e. after object construction and pre and post each method call. Line 3 contains an invariant that states the `_balance` field is always greater than or equal to zero.

### 4.3.5 The `represents` keyword

This keyword provides a means of associating model fields with concrete fields or methods. On line 4 the `_balance` field is associated with the value of the `getBalance` method. So now there is an invariant that says the `_balance` field must always be greater than zero and that the `_balance` field is represented by the value of the `getBalance()` method. This implies that the return value from the `getBalance` method must be greater than or equal to zero. The `ensures` clause post condition on line 12 states this fact.

### 4.3.6 The `AccountImpl` specifications

Before re-running ESC/Java2 on the tests, some JML specifications are required on the `AccountImpl` class. The tests construct new `AccountImpl(long)` objects therefore specifications are required to maintain the class invariant and to explicitly state what fields can be written to in the constructor. A new keyword `assignable` is introduced here. This keyword dictates what fields may be written to over the course of the constructor/method etc. By default in a non-pure method, everything is writable (`assignable \everything`) unless stated otherwise[LBR06]. ESC/Java2 will complain if it detects a field been written to that has not be declared in an `assignable` clause. In this case the `balance` field will be initialised with the value passed in via the constructor's parameter.

As the code stands right now, there is no way for a static analysis tool like ESC/Java2 to know that the `balance` field in `AccountImpl` is in any way related to the `model _balance` field specified in the `Account` interface. To accomplish this, another `represents` clause is used to associate the two.

There is one final annotation required. The `Account` interface contains a model data group field called `balanceState`. Data groups were explained in Section 4.3.2. The second model field `_balance` is specified to be `in` the `balanceState`. To recap, referral to the data group implicitly makes a reference to all the fields `in` the group. Now, the specifications on the `AccountImpl` state that the `balanceState` is `assignable` in the constructor. As a consequence, ESC/Java2 needs to be informed that the concrete `balance` field should be associated with the `balanceState` data group. This is achieved with an `in` clause after the `balance` declaration. Now the `balance` field is represented by the `_balance` model field and it is also `in` the `_balance` field which, in turn, is `in` the `balanceState` data group. See the next section for the full specifications on the constructor and member variables of the `AccountImpl` class.

## 4.4 Re-running the tests for `getBalance`.

```

1 public class AccountImpl implements Account {
2
3     private long balance = 0;  //@ in _balance;
4     //@ private represents _balance = balance;
5
6     public /*@ pure @*/ AccountImpl() {
7         this.balance = 0;
8     }
9
10    /*@ public normal_behavior
11       @ requires 0 <= openingBalance;
12       @ ensures getBalance() == openingBalance;
13       @ ensures 0 <= _balance;
14       @*/
15    public /*@ pure @*/ AccountImpl(final long openingBalance) {
16        if (0 <= openingBalance) {
17            this.balance = openingBalance;
18        } else {
19            this.balance = 0;
20        }
21    }
22 }

```

Listing 15: The AccountImpl with constructor and field specifications.

An important point to notice from Listing 15 is that although more specifications were added to the implementation class, the private field `balance` name is never divulged in the specifications. A layer of abstraction exists via `_balance` and `balanceState` in this case. This provides the ability to change the name of this field in future without affecting the specification in any way.

Now ESC/Java2 is re-run on the `AccountImplSpecTest` class.

```

1 Running Escjava checker
2 Specs project already exists - jmlspecs
3 Specs library already installed /jmlspecs
4 Using simplify executable at /Users/ralphhyland/CollegeWork/masters/
   Thesis/workspace/eclipse2/plugins/mobius.atp.simplify_1.5.11/
   Simplify-1.5.5.macosx
5 ESC/Java version ESCJava-CURRENT
6 [0.0030 s 73005992 bytes]
7
8 com.hyland.moosikbox.impl.AccountImplSpecTest ...
9 Prover started:0.057 s 78037008 bytes
10 [0.224 s 78275936 bytes]

```

```

11
12 com.hyland.moosikbox.impl.AccountImplSpecTest :
    testBalanceIsCorrectAfterInitialisation() ...
13     [0.201 s 75547096 bytes] passed
14
15 com.hyland.moosikbox.impl.AccountImplSpecTest :
    testInitialAccountBalances() ...
16     [0.219 s 79262056 bytes] passed
17
18 com.hyland.moosikbox.impl.AccountImplSpecTest: AccountImplSpecTest()
    ...
19     [0.0060 s 79496312 bytes] passed
20     [0.652 s 79498360 bytes total]
21 Escjava checker ended, status code 0

```

Listing 16: ESC/Java2 successful output after checking the `AccountImplSpecTest` class

The process demonstrated above is exactly how the rest of the Moosikbox application is developed, specified and tested. The next section details the `debit` and `credit` methods of the `Account` using the same steps as above. The full source code for the Moosikbox application can be found in [Appendix A](#).

#### 4.5 Adding specifications to the credit method.

The javadocs for the `credit` method state three things:

1. The account should be credited by the amount passed in
2. The amount passed in cannot be a negative value
3. An exception will be thrown if the amount is a negative value

```

1 /**
2  * Credits the account by the supplied amount.
3  * @param amount The amount to credit the account by. This can not be
4  * a negative value.
5  * @throws IllegalArgumentException if the {@code amount} is negative.
6  */
7 void credit(final long amount);

```

Listing 17: The `credit` method before JML specification.

The first and second are the postcondition and precondition respectively. The third specification states the required behaviour in an exceptional case i.e. a negative amount being passed.

In keeping with the process outlined above, a test is written for the `credit` method (Listing 18). Running ESC/Java2 against this test pinpoints a precondition error on the `AssertUtils.assertEquals` method. What looks obvious to the reader that the account's balance will be 15 after the `credit` method is executed, is not verifiable to ESC/Java2. It has no information as to what can possibly happen within the `credit` method and so can make no judgements about the state of the account.

The `credit` method is one that exhibits *normal* and *exceptional* behaviour. These behaviours are portrayed with `public normal_behavior` and `public exceptional_behavior` which, result in the two types of post conditions in JML; normal and exceptional. Normal postconditions are expressed by means of `ensures` clauses, that must hold when a method terminates normally. Exceptional postconditions are those expressed by means of `signals` clauses, that must hold when a method terminates with an exception [CKLP06]. All possible paths of the `credit` method are covered using normal and exceptional conditions.

#### 4.5.1 The *normal* path

In the normal execution of this method, once the precondition (`amount` greater than zero) is met, the balance of the account will be credited by the `amount`. The post condition ensures that the balance, after execution of `credit`, is equal to the old balance plus the `amount`. This is specified by the following clause:

```
@ensures \old(getBalance() + amount) == getBalance();
```

#### 4.5.2 The `\old` evaluator

This construct evaluates an expression to reveal its pre-state value. The line above states that the value of the `getBalance()` method prior to the call must equal the old value of `getBalance()` plus the `amount` i.e. the value of `getBalance()` prior to the invocation of the `credit` method.

#### 4.5.3 The *exceptional* path

The exceptional path also requires a precondition to hold if that path is to be taken. That precondition states that the `amount` is negative. The method will throw an exception once that precondition is adhered to and the postcondition will ensure that the balance remains as it was before the method was invoked. It is required to state the frame property of `assignable \nothing`; left unspecified this will default to `assignable \everything`;



```

1
2 public void testBalanceIsCorrectAfterCredit() {
3     Account account = new AccountImpl(10L);
4     account.credit(5L);
5     AssertUtil.assertEquals(15L, account.getBalance());
6 }

```

Listing 18: The test for the `credit` method.

#### 4.5.4 The signals clause

In JML, methods without an explicit throws clause have an implicit exceptional postcondition of `signals (Exception) false;`. Therefore, if ESC/Java2 finds code that throws an exception but there are no specifications to highlight that fact then it will complain because it was implicitly told not to expect any exceptions from the method. To specify a runtime exception such as `IllegalArgumentException`, that is not declared explicitly in the method signature, one must use the `signals_only` clause. Listing 19 shows the full heavyweight specification of the `credit` method.

```

1
2 /*@ public normal_behavior
3     @ requires 0 < amount;
4     @ ensures \old(getBalance() + amount) == getBalance();
5     @ assignable balanceState;
6     @ also
7     @ public exceptional_behavior
8     @ requires amount <= 0;
9     @ assignable \nothing;
10    @ signals_only IllegalArgumentException;
11    */
12 public void credit(final long amount) {
13 if (amount <= 0) {
14     throw new IllegalArgumentException("Amount can not be negative");
15 }
16     this.balance += amount;
17 }

```

Listing 19: The fully specified `credit` method.

## 5 Verification of JDK classes

### 5.1 Selecting the classes

JDK classes are selected for verification based on the following criteria:

1. The class is preferably in the top 100 on the specathon list<sup>8</sup>.
2. The JCK contains tests that utilize the class's important methods.

### 5.2 Set up a project

The verification process described here requires the creation of a Java project that will contain both the JDK classes to verify and the JCK test classes. From there the tools such as JML2 and ESC/Java2 can be employed. Eclipse is the chosen IDE in this thesis but any IDE or editor will suffice. One specific reason for choosing Eclipse is to benefit from the powerful ESC/Java2 verification plugin [MOB] written for the environment.

### 5.3 Target class: ResourceBundle

The first class selected is `java.util.ResourceBundle` as it appears at number 43 on the specathon list and there are adequate tests for the class in the JCK. According to the specathon list the most popular method in this class in terms of invocations is the `getString(String key)` method therefore that is focused upon here. The JCK contains a number of test classes for the methods in `ResourceBundle` and indeed for other classes too.

The test cases for the `getString(String key)` method are contained in a class called `javasoft.sqe.tests.api.java.util.ResourceBundle.castingGetterTests.java` while other test classes such as `getBundleTests.java` and `getObjectTests.java` test the methods `getBundle(String baseName)` and `getObject(String key)` respectively. This JCK approach of separating the tests for each method into specific classes is helpful to the process being applied here as it permits one to focus on the most important methods of a class in isolation.

*Note that the JCK class and package naming conventions permit test classes to have names starting with a lowercase letter and packages that begin with an uppercase letter.*

<sup>8</sup>The specathon is a Formal Methods Europe funded list compiled by Dan Zimmerman and Joe Kiniry that documents the most frequently invoked methods on the Java API.

After importing the source file for `ResourceBundle` into the working project the next step is to import the relevant test class or classes. For the moment, all that is required is the `castingGetterTests` file. This, like all of the JCK test classes, is buried in a deep package hierarchy. This particular package being `javasoft.sqe.tests.api.java.util.ResourceBundle`.

The JCK includes a concrete implementation of `ResourceBundle` called `PrimitiveBundle` that is used within it's test cases. This has no effect on the outcome of these tests and is used solely to facilitate the testing of the (abstract) `ResourceBundle` class API.

### 5.3.1 Run ESC/Java2 against the tests pre JML specification

When ESC/Java2 is executed against `castingGetterTests.java` two analysis warnings unfold. One of these warnings is of interest here as it pertains to the `getString(String key)` method. The other warning relates to the `getStringArray(String key)` method which is not examined here. The ESC/Java2 warning states that a possible null dereference can occur in the test i.e. a `NullPointerException`. ESC/Java2 emits this warning as it has no information about the intended behavior of the `getString(String key)` method to suggest otherwise.

ESC/Java2 flags the statement on line 25 (Listing 38) as a potential null dereference error as it cannot be sure if the `result` variable will have a value, or not, prior to the execution of `getString(String key)` on line 21. Notice the use of the `Status` class that was outlined in Section 1.

### 5.3.2 Writing JML specifications for the `getString(String key)` method

Section 2 noted that any JML specifications written in this process must be based solely on the Javadocs. This constraint is added to ensure that specifications are not written based on assumptions nor based on reading the source code of a method which, may be implementation specific. In this piece of work the source code happens to be available but this cannot be relied on to always be the case. Also, the approach shown here is one in which JML specifications are applied to existing classes.

Luckily the Javadocs for the `ResourceBundle` class are quite descriptive and therefore make the process of writing specifications a less ambiguous task.

As can be seen from Listing 20 the Javadocs for the `getString(String key)` method contain pre, post and exceptional conditions. When writing specifications for the Moosikbox application the JML annotations were added above the method signatures in a similar fashion to Javadoc placement. It was noted back then that specifications may also be contained in separate files to the source code once a naming convention is adhered to.

The specifications are applied to the `ResourceBundle` class using the latter of these approaches. A separate file called `ResourceBundle.refines-java` is created. This file will contain the class declaration and method signatures but will not contain any implementation code.

It is evident from the first line of the `getString(String key)` method's Javadocs (*Gets a string for the given key from this resource bundle or one of its parents.*) that some kind of internal storage is used for the data bundle and also that a reference to its parent is available. Given this information, two model fields are created in the specification of `ResourceBundle`. The first models the storage mechanism for this bundle's data and the second models the parent bundle.

```

1  /**
2   * Gets a string for the given key from this resource bundle or one of
3   * its parents.
4   * Calling this method is equivalent to calling
5   * <code>(String) { @link #getObject(java.lang.String) getObject}(key)
6   * </code>.
7   * </blockquote>
8   *
9   * @param key the key for the desired string
10  * @exception NullPointerException if <code>key</code> is <code>>null</code>
11  * @exception MissingResourceException if no object for the given key
12  * can be found
13  * @exception ClassCastException if the object found for the given key
14  * is not a string
15  * @return the string for the given key
16  */
17  public final String getString(String key);

```

Listing 20: The Javadocs for the `getString(String key)` method.

A heavyweight specification is added to the `getString(String key)` method and can be seen in Listing 21. This specification satisfies the warning that was previously raised by ESC/Java2 .

```

1  //@ public model Map bundle;
2  //@ public model ResourceBundle _parent;
3
4  /*@ public normal_behavior
5   * @ ensures \typeof(\result) == String.class;
6   * @ also
7   * @ public exceptional_behavior

```

```

8      @ requires key == null ||
9         \typeof(bundle.get(key)) != String.class ||
10         !bundle.containsKey(key) &&& !_parent.containsKey(key);
11     @ signals_only NullPointerException, ClassCastException,
12         MissingResourceException;
13     @ signals (NullPointerException) key == null;
14     @ signals (ClassCastException) \typeof(bundle.get(key)) != String.
15         class;
16     @ signals (MissingResourceException) !bundle.containsKey(key) &&& !
        _parent.containsKey(key);
17     @*/
18     public final /*@ pure non_null @*/ String getString(String key);

```

Listing 21: The specified getString(String key) method.

When ESC/Java2 is executed again the original warning is gone.

```

1
2 javasoft.sqe.tests.api.java.util.ResourceBundle.castingGetterTests :
   ResourceBundle0004() ...
3 [2.102 s 145449424 bytes] passed

```

Listing 22: ESC/Java2 successful results for the ResourceBundle0004() test method.

## 5.4 Target class: PrintStream

The next target class chosen is `java.io.PrintStream` and the method to test is `println(String)`. This class (and method) are number 9 on the specathon list and again there are adequate tests for this class within the JCK including the test class that is examined here:

```
javasoft.sqe.tests.api.java.io.PrintStream.ctorTests.java.
```

### 5.4.1 The existing specifications

The JML specifications for the `java.io.PrintStream` class are already present in the `esc1.4specs.jar`. Even still, it is used here to carry out an experiment to determine whether the JCK tests pass the ESC/Java2 checks against an already-specified class.

Basically, the same approach is taken as before but there is no requirement to write the JML specifications because they are already in place. So, the JCK related test class `ctorTests.java` is imported into the project and ESC/Java2 is ran against that. ESC/Java2 will first search the immediate project path for specifications related to the

`PrintStream` class (or other files of the same name but with a different suffix like `.refines-java`, `.jml`, `.spec`). When no specifications are found it continues its search down through the classpath until it either finds specifications or not. In this case it will find the specifications in a file called `PrintStream.refines-java` within the `esc1.4specs.jar`.

#### 5.4.2 Running ESC/Java2 against the tests pre JML specification

Running ESC/Java2 against the test class `ctorTests.java` reveals a precondition warning when an attempt is made to instantiate a `PrintStream` using the constructor that accepts an `OutputStream` (see Listing 24).

The first place to investigate is the specification file `PrintStream.refines-java` within the `esc1.4specs.jar`.

```

1 // @ public normal_behavior
2 // @ requires os != null;
3 // @ ensures outputText.equals("");
4 // @ ensures !endsInNewLine;
5 // @ ensures os.isOpen ==> isOpen;
6 // @ ensures underlyingStream == os;
7 // @ pure
8 public PrintStream(OutputStream os);

```

Listing 23: The specified `PrintStream(OutputStream os)` constructor.

From the precondition it is clear that the constructor expects a non null value for the parameter `os`. The precondition warning that occurs when ESC/Java2 is ran against the test class (as stated above) happens because prior to being used as the parameter to the `PrintStream` constructor, the `OutputStream` instance is explicitly set to null (as can be seen in Listing 24).

```

1 OutputStream os = null;
2
3 try {
4     y1 = new PrintStream(os); // Try to create object
5
6 } catch (NullPointerException e) { // Should not create invalid object
7     return Status.passed("OKAY");
8 }

```

Listing 24: Excerpt from the `ctorTest.java` JCK test class.

So it appears that there are discrepancies between what the JCK test expects to happen and what the JML specifications say. Before making a judgement on a course of action to

take, the Javadocs for `PrintStream` and its ancestors must be examined. The following points outline the results of that examination:

1. Neither the class summary nor constructor doc comments from `java.io.PrintStream` mention that a `NullPointerException` will be thrown if a null parameter is passed in.
2. The parent class `java.io.FilterOutputStream`, although it has a constructor that accepts an `OutputStream` as a parameter, has no mention of the expected exception.
3. The next class in the hierarchy is `java.io.OutputStream` and again this states nothing about null streams.

Given that the JCK test in `ctorTest.java` (Listing 24) will pass **only** if a `NullPointerException` is thrown from the constructor of `PrintStream`, the only logical suggestion to make is that there is a Javadoc bug in the `PrintStream` class hierarchy. The reasons behind this conclusion are:

1. The JML specification of the constructor shown in Listing 23 is as complete as it can be given the Javadocs available.
2. The test from the JCK has made an assumption (it is mentioned nowhere in the public documentation) that `PrintStream` must throw the exception.
3. Therefore the problem appears to lie with the quality of the doc comments for the `PrintStream` class.

The following extract is taken from the Sun Microsystems Javadoc writing guides:

*API spec bugs are bugs that are present in the method declaration or in the doc comment that affects the syntax or semantics. An example of such a spec bug is a method that is specified to throw a `NullPointerException` when null is passed in, but null is actually a useful parameter that should be accepted (and was even implemented that way).<sup>9</sup>*

Now, although different from the example demonstrated here, it is evident that the Javadocs for the `PrintStream` constructor would be more explicit if they stated that a `NullPointerException` is thrown when a null `OutputStream` is received. As far as the process in this document goes, there is no more that can be done to fix the ESC/Java2 specification warning from the test class. The Javadocs say nothing about the expected exception and therefore, according to the criteria of this process that says specifications may only be applied based on the Javadocs, the JML specifications that are currently added to the `PrintStream` constructor are as complete as they can be.

Although the Javadocs in question here have been in place for ten years or more and may not be of major concern to developers due to the fact that it might be obvious that

<sup>9</sup><http://java.sun.com/j2se/javadoc/writingdoccomments/>

the `OutputStream` passed in the constructor of a `PrintStream` can not be null, it does highlight inconsistencies in the Javadoc specifications. Coupled with the fact that there is a JCK test (Listing 24) that explicitly depends on a certain exception being thrown by a constructor in order to pass its test. It also makes the job of tools such as ESC/Java2 , and the theorem prover used by such a tool, that much more difficult. Correct application of a BISL like JML, coupled with ESC/Java2 , has the potential to eliminate this kind of ambiguity while also providing a basis on which secondary tools can make certain assertions about the class's intended behaviour.

## 5.5 Target class: Stack

The third and final target class shown in this document is `java.util.Stack`. Again this class is on the specathon list at number 43 and the JCK contains adequate test cases for it. The method `elementAt(int)` mentioned on the list is inherited from the class `java.util.Vector` and is fully specified within the `esc1.4specs.jar` therefore it is of no benefit to re-specify it here. To be thorough, the related test class from the JCK (`javasoft.sqe.tests.api.java.util.Vector.ElementAtTests.java`) is imported into the project anyway and ESC/Java2 is executed against it. Each of the tests pass and so it is assumed the JML specifications for the `elementAt(int)` method are correct (within the `esc1.4specs.jar`).

The `Stack` class is one of the more interesting classes on the specathon list and contains methods such as `push(Object item)`, `Object pop()`, and `Object peek()`. The `push(Object item)` method is examined here even though it does not show up on the specathon list.

### 5.5.1 Following the Javadocs

As shown in Section 5.4.2 there is a requirement to read the Javadocs throughout a class hierarchy, in addition to those on the target class, in order to glean as much relevant information as possible about the class or method in question. A good example of this is the `push(Object item)` method on the `Stack` class.

```
1 /**
2  * Pushes an item onto the top of this stack. This has exactly the same
3  * effect as: <blockquote>
4  *
5  * <pre>
6  * addElement(item)
7  * </pre>
8  *
```



```

9  * </blockquote>
10 *
11 * @param item
12 *         the item to be pushed onto this stack.
13 * @return the <code>item</code> argument.
14 * @see java.util.Vector#addElement
15 */
16 public Object push(Object item);

```

Listing 25: The `public Object push(Object item)` signature and Javadocs.

It is evident from the Javadocs in Listing 25 that no constraints are put on the `item` parameter in terms of whether it can be null etc. What it does state in the Javadoc method description is that “*This has exactly the same effect as `addElement(item)`*”. It also makes another reference to that method in the `@see` annotation. It is therefore, our obligation to check the referenced method to determine if it’s Javadocs state anything explicit about the `item` parameter. This might be misconstrued as cheating by delving into implementation but it is perfectly legitimate because the **public** Javadocs explicitly state the information and therefore it was intended to be followed.

The results from following the Javadoc trail were:

1. The docs for the `addElement(item)` method (Listing 34) uncovers nothing obvious **but** it states that the method is identical in functionality to the `add(Object)` method (which is part of the `Collection` interface). So that needs to be checked also.
2. The docs from the `add(Object)` method contains four separate `@throws` annotations (plus the normal description, `@params`, and `@returns` annotations). The `@throws` annotation of interest states that “a `NullPointerException` is thrown if the specified element (the object) is null and the collection does not permit null elements”.
3. The same method’s doc comments contain this statement: “Collection classes should clearly specify in their documentation any restrictions on what elements may be added.”
4. The `Stack` or `Vector` classes do not mention whether null elements are accepted.
5. The `ArrayList` class from the same package does honour the `Collection` interface docs, it states that “Implements all optional list operations, and permits all elements, including null.”.

### 5.5.2 Writing specifications for Stack

The JML specifications for `Vector` (inside the `esc1.4specs.jar`) state that the class can contain null. The specification writers obviously felt that a decision needed to be made about

how to continue with the specification without explicit knowledge of the type of elements handled and that decision was to assume that null elements are permitted. They most likely tested a piece of code to determine the inconclusive specification behaviour:

```
try {
    Vector v = new Vector();
    v.add(null);
    System.out.println("Allow Null Elements!");
} catch (Exception e) {
    System.out.println("Disallow null elements!");
}
```

Of course this is all seemingly obvious stuff but, just like writing software, it is never a good idea to be tied to a particular implementation. JML specifications that are written based on implementation behaviour are brittle and vulnerable to implementation changes.

Moving on with the specifications of `Stack`. All the methods of this class are specified here, except `pop()` (which proved too difficult to get past ESC/Java2 in the end), using the same semantics as `Vector` i.e. the specifications assume that the `Stack` class accepts null elements.

As stated above, the `push(Object)` method has exactly the same effect as the `addElement(item)` method from `Vector` so it makes sense to borrow the specifications from there as they already exist.

**An aside: New JML annotation.** *What would be extremely useful in this situation would be a specification annotation that states that “this method uses the same specification as the method `m()` in the class `C`” or written in JML syntax this might look something like:*

```
\\@uses Class.m();
public void otherMethod();
```

*This annotation may be researched as further work.*

Line 11 of the `addElement(item)` specification in Listing 35 states that the type of the object currently under execution must be equal to `java.util.Vector`:

```
\\@ requires \typeof(this) == \type(java.util.Vector);
```

It appears that there is a slight bug in that statement because everything after that statement i.e. pertaining to classes that are of the `Vector` type, should be applicable to subtypes of `Vector` also e.g. `Stack`. A correction is proposed here:

```
\\@ requires \typeof(this) <: \type(java.util.Vector);
```

With that caveat cleared up it was possible to borrow the relevant portion of specifications from `Vector.addElement(item)`. Coupled with the specifications created here that adhere to the Javadocs on the `push(Object)` method the resulting JML specifications for that method (and the rest of the `Stack` class) are shown in Listing 37. Note that the specifications were carried out in a file called `Stack.refines-java` as opposed to writing them inline with the source code.

### 5.5.3 Running ESC/Java2 against the Stack tests

The JCK test classes `SearchTests.java`, `PushTests.java`, `PeekTests.java`, and `EmptyTests.java` all passed the ESC/Java2 verification process. One small issue worth noting is that some of those test cases involved the use of the `pop` method in their set up and so ESC/Java2 struggled with those ones because it could make no guarantees about its behaviour due to lack of specification. Even still, it is clear to see the specified methods pass their associated tests.

## 6 Future Work

### 6.1 Continuing the process

The JDK is an extremely large collection of classes and the task of specifying each of those with JML is daunting but one that is already well underway as is evident from the `esc1.4specs.jar`. The specathon list, used in this document for selecting target classes for specification, is something that will potentially provide the specification community with an up-to-date list of the specified/unspecified classes from the JDK and will allow users to “checkout” a class of interest to them for specification and verification using the process put forward in this thesis. A continuous build system should be established in order to keep tight reign on the quality of specification checked into the repository. This system would check specifications using the JML suite of tools in addition to running ESC/Java2 against those specifications.

### 6.2 Narrowing the gap

Narrowing the gap between ESC/Java2, JML, and the version of Java is another major target for the JML specification community. As stated previously, the work in this paper was carried out using Java 1.4 as this is the version currently supported by ESC/Java2. Even at that the collection classes are not completely specified at this point.

### 6.3 OpenJML

OpenJML is the next release of the suite of JML tools, including RAC and ESC, that will work with the most recent versions of Java. OpenJML is built on Sun’s OpenJDK, the open source implementation of `javac`<sup>10</sup>.

### 6.4 New JML Annotation

In Section 5.5.2 the potential for a useful new JML annotation was mentioned. It is envisaged that this new annotation would be used to point a specification tool reader to an existing specification from a different method or class e.g. the idea was mentioned when the `push(Object)` method of `Stack` was being specified. It was noted that this method has the same semantics as the `addElement(Object)` method of `Vector`. Note that there is no inheritance between those two methods in terms of functionality. The `push(Object)` method delegates to the `addElement(Object)` method (which is in the parent `Vector`

<sup>10</sup><http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml>

class) and so is known to have the same pre and post conditions as that method. The proposed annotation would be along the lines of:

```
1 public Stack {  
2  
3     \\@ uses Vector.addElement(Object);  
4     public Object push(Object item);  
5 }
```

Listing 26: The proposed new JML annotation @uses

## 7 Conclusion

The subject of software analysis and verification is a highly complex area but one which holds great potential for enhancing the way software is written, tested, and ultimately delivered and maintained.

This paper described “*A Process for the Specification of Core JDK Classes*” which employs JML, ESC/Java2 , and the Java(TM) Compatibility Kit (JCK) test classes which are statically checked and verified. During this process, various aspects of software verification have been demonstrated. The Behavioural Interface Specification Language (BISL) JML was used to write specifications and the JML2 tools `jml`, `jmlc`, and `jmlrac` were used to check, compile, and execute Java code supplemented with JML specifications. The static analysis tool ESC/Java2 was used both as a standalone static analysis checker i.e. analysing Java code only, and for Java code with JML specifications applied.

All development and specification writing was undertaken against Java 1.4 as this is the version currently supported by ESC/Java2 . As mentioned in Section 6, OpenJML is the next generation of JML and work is currently ongoing for an updated ESC/Java2 that will be compatible with the latest versions of Java.

### 7.1 Observations from writing specifications.

Writing specifications has benefits beyond the obvious. An interesting side effect is that the developer is urged to look at an API through the eyes of both the client and supplier of the class. This becomes a natural process for a developer since specifications generally contain pre and post conditions. Sometimes the design of the API is driven by this process, as was the case while writing the example project for this thesis. General awareness of cause and effect has definitely been heightened by using the verification process. Heavy emphasis is placed on testing of all kinds (unit, functional, integration etc) and the author found writing specifications to be a really good way of improving one’s understanding of an API and about ‘what’ needs to be tested. In fact, discovering what needs to be tested falls naturally out of writing specifications. Properly specified and tested units of software can only lessen maintenance costs in terms of refactoring time, bug fixing, and the time required by new developers to understand the code in question.

#### 7.1.1 Combination of JML and ESC/Java2 .

It is advisable to always execute both the JML checker in addition to ESC/Java2 when writing specifications as there are certain cases where ESC/Java2 will not detect as much

as the checker. The MOBIUS plugin for the Eclipse development environment proved to be an invaluable tool during this process.

### 7.1.2 The verification of JDK classes using JCK test classes, JML specifications and ESC/Java2 .

The process itself is a powerful tool that has the potential to produce a high level of certainty about a piece of software and it's related unit tests. While it has been demonstrated that the process can work, it is really in it's infancy right now. In order to become a relevant development tool, the static analysis and verification tools must be brought inline with the latest JDK (as previously mentioned, this is already happening with OpenJML).

### 7.1.3 Standalone ESC/Java2

It was shown that ESC/Java2 used in isolation is a valuable static analysis tool and can be used to detect potential runtime errors like null deferences that may find their way into the runtime environment.

## 7.2 Findings

1. The goal of verifying JDK classes against their respective JCK tests using JML and ESC/Java2 for static analysis was acheived in some cases e.g. `ResourceBundle` (Section 5.3) and `Stack` (Section 5.5.2) classes.
2. The process itself uncovered interesting information about *some* of the Javadocs on the JDK classes in terms of them not adequately expressing the intentions of the functionality. Examples of this were shown in Section 5.4.2 and 5.5.1.
3. A suspected JML specification bug was uncovered in Section 5.5.1 regarding the specifications on the `addElement(Object)` method of the `Vector` class.
4. Currently, the `esc1.4specs.jar` is missing several classes and has some incomplete specifications.
5. The process highlights the importance of writing explicit specification documentation, whether that be Javadocs or JML. It also shows why specifications should be written at an abstract level where possible to provide the most flexibility and resilience to change in the future.
6. Javadocs are an excellent source of documentation when applied, and maintained, correctly. Using a BISL like JML provides additional tool support features like cross-

checking specifications, compilation, and execution of those specifications. Furthermore, unit tests can be auto-generated from these tool-readable specifications. As these tools are runnable the likelihood of them becoming out of sync with the source code, the way Javadocs tend to, is slim.

7. Using the process outlined in this thesis for specifying JDK classes, one is required to follow the Javadocs until a definite decision can be made about whether to add (or leave out) JML specifications. While this process has its benefits, as a greater understanding of the API is learned along the way, it is another reason why the use of Javadocs is limited in its usefulness.



## A Moosikbox source code including specifications.

```

1 package com.hyland.moosikbox;
2
3 /**
4  * Describes the behaviour of an account.
5  *
6  * @author ralphhyland
7  */
8 public interface Account {
9
10  // @ public instance model JMLDataGroup balanceState;
11  // @ public instance model long _balance; in balanceState;
12  // @ public instance invariant 0 <= _balance;
13  // @ private represents _balance = getBalance();
14
15  /**
16   * Retrieve the current balance.
17   * @return The current balance which should never be negative.
18   */
19  // @ public normal_behavior
20  // @ ensures 0 <= \result;
21  // @ ensures \result == _balance;
22  // @ pure @*/ long getBalance();
23
24  /**
25   * Credits the account by the supplied amount.
26   * @param amount The amount to credit the account by. This can not
27   *   be
28   * a negative value.
29   * @throws IllegalArgumentException if the {@code amount} is negative
30   */
31  // @ public normal_behavior
32  // @ requires 0 < amount;
33  // @ ensures \old(getBalance() + amount) == getBalance();
34  // @ assignable balanceState;
35  // @ also
36  // @ public exceptional_behavior
37  // @ requires amount <= 0;
38  // @ assignable \nothing;
39  // @ signals_only IllegalArgumentException;
40  // @*/
41  void credit(final long amount);

```

```

42  /**
43   * Debits the account by the supplied amount.
44   * @param amount The amount to debit the account by. This can not be
45   * a negative value.
46   * @throws IllegalArgumentException if the amount is negative.
47   * @throws IllegalStateException if {@code getBalance()} is less than
48   *   {@code amount}.
49   */
50  /*@ public normal_behavior
51   @ requires amount <= getBalance() && 0 < amount;
52   @ ensures \old(getBalance() - amount) == getBalance();
53   @ assignable balanceState;
54   @ also
55   @ public exceptional_behavior
56   @ requires (getBalance() < amount) || (amount <= 0);
57   @ modifies \nothing;
58   @ signals_only IllegalArgumentException, IllegalStateException;
59   @ signals (IllegalArgumentException) amount <= 0;
60   @ signals (IllegalStateException) getBalance() < amount;
61   @*/
62  void debit(final long amount);

```

Listing 27: The Account interface with JML specification.

## A.1 The AccountImpl class

```

1  package com.hyland.moosikbox.impl;
2
3  import com.hyland.moosikbox.Account;
4
5  public class AccountImpl implements Account {
6
7      private long balance = 0; //@ in _balance;
8      //@ private represents _balance = balance;
9
10     //@ assignable _balance;
11     //@ ensures 0 == _balance;
12     public AccountImpl() {
13         this.balance = 0;
14     }
15
16     //@ requires 0 <= openingBalance;
17     //@ assignable _balance;

```

```

18  //@ ensures getBalance() == openingBalance;
19  //@ ensures 0 <= _balance;
20  public AccountImpl(final long openingBalance) {
21      if (0 <= openingBalance) {
22          this.balance = openingBalance;
23      } else {
24          this.balance = 0;
25      }
26  }
27
28  public /*@ pure @*/ long getBalance() {
29      return balance;
30  }
31
32  //@ also assignable _balance;
33  public void credit(final long amount) {
34      if ((amount > 0) && (balance + amount) > 0) {
35          this.balance += amount;
36          return;
37      }
38  }
39
40  //@ also assignable _balance;
41  public void debit(final long amount) {
42      if ((amount > 0) && (balance - amount) >= 0) {
43          this.balance -= amount;
44          return;
45      }
46  }
47  }

```

Listing 28: The AccountImpl class.

## A.2 The Artist interface

```

1  package com.hyland.moosikbox;
2
3  public interface Artist {
4      //@ public instance model String _name;
5      //@ private represents _name = getName();
6      //@ public invariant _name != null;
7
8      //@ public instance model String _label;
9      //@ private represents _label = getLabel();

```

```

10  //@ public invariant _label != null;
11
12  //@ pure non_null @*/ String getName();
13
14  //@ assignable _name;
15  void setName( //@ non_null @*/ String name);
16
17  //@ pure non_null @*/ String getLabel();
18
19  //@ assignable _label;
20  void setLabel( //@ non_null @*/ String label);
21  }

```

Listing 29: The Artist interface.

### A.3 The Broker interface

```

1  package com.hyland.moosikbox;
2
3  public interface Broker {
4
5      //@ requires buyer.hasFunds(collectable.getPrice());
6      //@ requires seller.hasItemForSale(collectable);
7      void handleSale( //@ non_null @*/ final Collector seller , //@
8          non_null @*/ final Collector buyer, //@ non_null @*/ final
9          Collectable collectable);
10 }

```

Listing 30: The Broker interface.

### A.4 The Collectable interface

```

1  package com.hyland.moosikbox;
2
3  public interface Collectable {
4
5      //@ pure @*/String getCatalogueReference();
6
7      void setCatalogueReference( //@ non_null @*/ final String
8          catalogueReference);
9
10     //@ pure @*/ Artist getArtist();

```

```

10
11 void setArtist( /*@ non_null @*/ final Artist artist);
12
13 //@ ensures \result >= 0;
14 /*@ pure @*/ long getPrice();
15
16 //@ requires price >= 0;
17 //@ ensures getPrice() >= 0;
18 void setPrice(final long price);
19 }

```

Listing 31: The Collectable interface.

## A.5 The Collector interface

```

1 package com.hyland.moosikbox;
2
3 public interface Collector {
4
5     //@ public instance model JMLDataGroup accountState;
6     //@ public initially accountState != null;
7     //@ public instance model JMLDataGroup collectionState;
8     //@ public instance model Account _account; in accountState;
9     //@ public instance model Collectable [] _collection; in
10     collectionState;
11     //@ public represents _collection = getCollection();
12
13     /*@ pure non_null @*/ Collectable [] getCollection();
14
15     //@ public normal_behavior
16     //@ assignable collectionState;
17     void addItem(/*@ non_null @*/ final Collectable collectable);
18
19     /*@ pure @*/ boolean ownsItem(/*@ non_null @*/ final Collectable
20     collectable);
21
22     //@ public normal_behavior
23     //@ requires hasItemForSale(collectable);
24     //@ assignable collectionState;
25     void removeItem(/*@ non_null @*/ final Collectable collectable);
26
27     /*@ pure @*/ boolean hasItemForSale(/*@ non_null @*/ final
28     Collectable collectable);

```

```

27  //@ ensures \result <==> amount <= _account.getBalance();
28  /*@ pure @*/ boolean hasFunds(long amount);
29
30  //@ public normal_behavior
31  //@ assignable accountState;
32  //@ ensures _account.getBalance() == \old(_account.getBalance() -
    amount);
33  void deductFunds(final long amount);
34
35  //@ public normal_behavior
36  //@ requires 0 <= amount;
37  //@ assignable accountState;
38  //@ ensures _account.getBalance() == \old(_account.getBalance() +
    amount);
39  void increaseFunds(final long amount);
40  }

```

Listing 32: The Collector interface.

```

1  package com.hyland.testutils;
2
3  import com.sun.javatest.Status;
4
5  public class AssertUtil {
6
7      private AssertUtil() {
8      }
9
10     //@ public normal_behavior
11     //@ requires actual == expected;
12     public static /*@ pure @*/ void assertEquals(final long actual, final
        long expected) {
13         if (actual == expected) {
14             Status.passed("");
15             return;
16         }
17         Status.failed("");
18     }
19 }

```

Listing 33: The AssertUtils helper class for testing.

## A JDK Target class signatures and Javadocs.

### A.1 Vector addElement(Object obj)

```

1 /**
2  * Adds the specified component to the end of this vector,
3  * increasing its size by one. The capacity of this vector is
4  * increased if its size becomes greater than its capacity. <p>
5  *
6  * This method is identical in functionality to the add(Object) method
7  * (which is part of the List interface).
8  *
9  * @param   obj   the component to be added.
10 * @see     #add(Object)
11 * @see     List
12 */
13 public synchronized void addElement(Object obj)

```

Listing 34: Vector's addElement(Object obj) method signature and Javadocs.

### A.2 Vector addElement(Object obj) JML specifications.

```

1 /*@ public normal_behavior
2   @   requires \typeof(obj) <: elementType;
3   @   requires containsNull || obj!=null;
4   @   assignable objectState;
5   @   ensures \not_modified(containsNull, elementType);
6   @   ensures (obj == null & get(elementCount-1) == null)
7   @           || get(elementCount-1).equals(obj); // or == ?
8   @   ensures elementCount == \old(elementCount)+1;
9   @ also
10  @   public normal_behavior
11  @   requires \typeof(this) == \type(java.util.Vector);
12  @   {
13  @   requires elementCount < maxCapacity;
14  @   assignable objectState;
15  @   ensures \not_modified(maxCapacity) && \not_modified(
16  @   capacityIncrement);
17  @   also
18  @   requires elementCount == maxCapacity;
19  @   {
20  @   requires 0 < capacityIncrement

```

```

20      @          && maxCapacity <= Integer.MAX_VALUE -
           capacityIncrement;
21      @          assignable objectState;
22      @          ensures maxCapacity == \old(maxCapacity) +
           capacityIncrement;
23      @          also
24      @          requires capacityIncrement == 0 && maxCapacity == 0;
25      @          assignable objectState;
26      @          ensures maxCapacity == 1;
27      @          also
28      @          requires capacityIncrement == 0
29      @          && maxCapacity > 0
30      @          && maxCapacity < Integer.MAX_VALUE/2;
31      @          assignable objectState;
32      @          ensures maxCapacity == \old(maxCapacity) * 2;
33      @      |}
34      @  |}
35      @*/
36      public synchronized void addElement(Object obj);

```

Listing 35: Vector's addElement(Object obj) JML specifications.

### A.3 List add(Object o)

```

1  /**
2  * Appends the specified element to the end of this list (optional
3  * operation). <p>
4  *
5  * Lists that support this operation may place limitations on what
6  * elements may be added to this list. In particular, some
7  * lists will refuse to add null elements, and others will impose
8  * restrictions on the type of elements that may be added. List
9  * classes should clearly specify in their documentation any
   restrictions
10 * on what elements may be added.
11 *
12 * @param o element to be appended to this list.
13 * @return <tt>true</tt> (as per the general contract of the
14 *         <tt>Collection.add</tt> method).
15 *
16 * @throws UnsupportedOperationException if the <tt>add</tt> method is
   not
17 *         supported by this list.
18 * @throws ClassCastException if the class of the specified element

```



```

19 *           prevents it from being added to this list.
20 * @throws NullPointerException if the specified element is null and
    this
21 *           list does not support null elements.
22 * @throws IllegalArgumentException if some aspect of this element
23 *           prevents it from being added to this list.
24 */
25 boolean add(Object o);

```

Listing 36: List's add(Object o) signature and Javadocs.

```

1 public class Stack extends Vector {
2
3     //@ public instance invariant containsNull;
4     //@ public instance invariant elementType == \type(Object);
5
6     //@ ensures empty();
7     public Stack();
8
9     /*@ requires !empty();
10    @ ensures !empty();
11    @ also requires empty();
12    @ signals_only EmptyStackException;
13    @ signals (EmptyStackException) empty();
14    */
15    public synchronized Object peek() throws EmptyStackException;
16
17    /*@ public normal_behavior
18    @ requires \typeof(item) <: elementType;
19    @ requires containsNull || item!=null;
20    @ assignable objectState;
21    @ ensures \not_modified(containsNull, elementType);
22    @ ensures (item == null & get(elementCount-1) == null)
23    @           || get(elementCount-1).equals(item); // or == ?
24    @ ensures elementCount == \old(elementCount)+1;
25    @ ensures !isEmpty();
26    */
27    public Object push(Object item);
28
29    public /*@ pure */boolean empty();
30
31    public synchronized /*@ pure */ int search(Object o);
32 }

```

Listing 37: The JML specifications for the Stack class.

```

1  /**
2   * Equivalence class partitioning
3   * with input values orientation
4   * for String getString(String key),
5   * <br><b>key</b>: an existing key for a string value.
6   * <br><b>Expected results</b>: will return the string value
7   */
8  public Status ResourceBundle0004() {
9      ResourceBundle bundle = new PrimitiveBundle("stringKey", "
10         stringValue", null);
11
12     String result = null;
13     String expectedResult = null;
14
15     try {
16         expectedResult = (String)bundle.getObject("stringKey");
17     } catch(MissingResourceException e) {
18         return Status.failed("Wrong data; unexpected exception thrown
19             : " + e);
20     }
21
22     try {
23         result = bundle.getString("stringKey");
24     } catch(Exception e) {
25         return Status.failed("Unexpected exception thrown : " + e);
26     }
27
28     if (!result.equals(expectedResult)) {
29         return Status.failed(
30             "getString\"stringKey\") returned wrong result : " + result
31             +
32             "expected = " + expectedResult
33         );
34     } else {
35         return Status.passed("OK");
36     }
37 }

```

Listing 38: The ResourceBundle0004 test in the castingGetterTests class from the JCK.

## References

- [Blo08] Joshua Bloch, *Effective java (2nd edition) (the java series)*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.
- [CKC08] David Cok, Joe Kiniry, and Dermot Cochran, *Esc/java2 implementation notes*, 2008.
- [CKLP06] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll, *Beyond assertions: Advanced specification and verification with JML and ESC/Java2*, Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, vol. 4111, 2006, pp. 342–363.
- [CL02] Yoonsik Cheon and Gary T. Leavens, *A runtime assertion checker for the Java Modeling Language (JML)*, Tech. Report 02-05, Department of Computer Science, Iowa State University, March 2002, In SERP 2002, pp. 322-328.
- [FLL<sup>+</sup>99] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata, et al., *The Extended Static Checker for Java*, 1999, See <http://research.compaq.com/SRC/esc/>.
- [Hoa83] C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **26** (1983), no. 1, 53–56.
- [KC05] Joseph R. Kiniry and David R. Cok, *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system*, Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004, vol. 3362, January 2005.
- [LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby, *Preliminary design of JML: A behavioral interface specification language for Java*, no. 3, 1–38.
- [LC05] Gary T. Leavens and Yoonsik Cheon, *Design by contract with jml*, Draft, available from [jmlspecs.org](http://jmlspecs.org)., 2005.
- [LCC<sup>+</sup>05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok, *How the design of jml accommodates both runtime assertion checking and formal verification*, Sci. Comput. Program. **55** (2005), no. 1-3, 185–208.
- [LKP07] Gary T. Leavens, Joseph R. Kiniry, and Erik Poll, *A JML tutorial: Modular specification and verification of functional behavior for Java*, Computer Aided Verification 2007 (Berlin) (Werner Damm and Holger Hermanns, eds.), vol. 4590, July 2007, p. 37.

- [Mey92] Bertrand Meyer, *Applying "design by contract"*, Computer **25** (1992), no. 10, 40–51.
- [MOB] *The MOBIUS project.*