Accessible Formal Methods:
A Study of the Java Modeling Language

A project presented to the
Department of Computer and Information Sciences
State University of New York Polytechnic Institute at Utica
Utica, New York

In partial fulfillment
of the requirements of the
Master of Science Degree

Mike Rawding
May 2017
DECLARATION

I declare that this project is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Michael Rawding

Michael Rawding
SUNY POLYTECHNIC INSTITUTE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

Approved and recommended for acceptance
as a project in partial fulfillment of the requirements
for the degree of Master of Science in
Computer and Information Science

April 27, 2017

DATE

Bruno Andriamanalimanana
Dr. Bruno R. Andriamanalimanana
Advisor

Dr. Scott Spetka

Dr. Roopa Vishwanathan
DECLARATION

I declare that this project is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Michael Rawding
Abstract

While formal methods offer the highest level of confidence that software behaves as intended, they are notoriously difficult to use. The Java Modeling Language and the associated OpenJML tool aim to make formal specification and verification more accessible to Java developers. This report gives an overview of JML and assesses its current status and usability. Though many common Java features have been implemented, lack of standard library support is identified as an obstacle to using JML effectively. To help address that problem, this report documents the process of adding support for a new library to OpenJML.
Acknowledgements

I would like thank Dr. Bruno R. Andriamanalimanana for his help designing this project and for guiding me through the review process. I would also like to thank Dr. David R. Cok for answering questions, providing guidance, and writing bug fixes to OpenJML. Thanks to Dr. Scott Spetka and Dr. Roopa Vishwanathan for review and feedback. Finally, thank you to the handful of individuals who have contributed to JML/OpenJML over the years to get it to a point where projects like this are possible.

This work is dedicated to Chelsea.
Thank you for supporting me in all of my endeavors, often in ways unseen.
# Table of Contents

Chapter 1: Introduction ................................................................. 1  
  1.1 Formal Methods ................................................................. 1  
    1.1.1 Specification .............................................................. 1  
    1.1.2 Verification .............................................................. 1  
  1.2 Project Statement .............................................................. 2  

Chapter 2: An Overview of JML ..................................................... 3  
  2.1 Language Format .............................................................. 3  
  2.2 Visibility ................................................................. 4  
  2.3 Preconditions ............................................................. 4  
  2.4 Postconditions .............................................................. 5  
  2.5 Method Side Effects and Purity ........................................ 5  
  2.6 Specification Side Effects ............................................... 6  
  2.7 Class Invariants ............................................................ 7  

Chapter 3: Tools and Applications .............................................. 8  
  3.1 Runtime Assertion Checking ........................................... 8  
  3.2 Static Checking ........................................................... 9  
  3.3 Automated Unit Test Generation ...................................... 9  
  3.4 Automated Documentation Generation ................................ 9  
  3.5 OpenJML ................................................................. 10  
    3.5.1 Type-Checking ...................................................... 10  
    3.5.2 Runtime Assertion Checking ................................... 10  
    3.5.3 Extended Static Checking ...................................... 10  
    3.5.4 Limitations of OpenJML ........................................ 11  

Chapter 4: Design .................................................................. 12  
  4.1 Considerations When Specifying .................................... 12  
  4.2 Libraries Selected .......................................................... 12
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Layout</td>
<td>12</td>
</tr>
<tr>
<td>4.4</td>
<td>Develop</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>ment Env</td>
<td></td>
</tr>
<tr>
<td>Chapter 5:</td>
<td>Implement</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>ation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.1</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>Modeling StringBuilder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Specifying Interfaces</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.3</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Runtime Assertion Checking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.4</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Extended Static Checking</td>
<td></td>
</tr>
<tr>
<td>Chapter 6:</td>
<td>Conclusion</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Accomplishments</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Future Work</td>
<td></td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>Appendix</td>
<td></td>
<td>22</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

As software becomes increasingly complex, so does the struggle to confirm that programs behave as intended. Traditionally, methods such as automated unit and end-to-end testing have been used for this task. Indeed, it is not uncommon for a modern code base to be accompanied by tens of thousands of such tests. Although these methods can provide decent confidence under certain circumstances, only conditions that are anticipated by the developer can be checked. This means that such test suites rarely cover all possible behaviors. Furthermore, large test suites quickly become cumbersome and they must be maintained in addition to the original software. Formal methods aim to provide more concrete evidence that a program behaves as intended while eliminating the need to maintain thousands of individual tests.

1.1 Formal Methods

Formal methods are concepts that are used in an attempt to prove the correctness of software. In their purest form, formal methods involve rigorous mathematical proofs, or exhaustive searches over all possible program states. However, associated techniques that are not quite so rigorous are also used in practice [1]. While there are a few different ways that formal methods are applied, they all require a two-step process of specification then verification.

1.1.1 Specification

Specification entails using a formal language to precisely describe the intended behavior of a program. Specification languages must be expressive enough to comprehensively describe intended behavior, as well as formal enough to be translated into verifiable axioms.

Some specification languages are designed to be used alongside a traditional programming language. These languages are a formal representation of a particular software implementation. In other cases, a specification language may simply describe a detailed model of a program without being directly linked to any actual implementation.

1.1.2 Verification

Verification is the process of comparing a program’s behavior to its specification in an attempt to rule out discrepancies. One way to do this, known as model checking, is by running a simulation
of a program in an effort to demonstrate correct behavior for all reasonable paths. Another method, theorem proving, decomposes specifications into axioms and searches for counter examples that would invalidate these axioms.

1.2 Project Statement
The benefits of formal methods would be valuable to any code base, but they are notoriously difficult to use and their application is time intensive. For this reason, the use of formal methods has generally been limited to critical software like avionics and autonomous vehicles [2]. The Java Modeling Language (JML) and associated tools aim to make formal methods more accessible to developers [3]. The goal of this project is to aid the JML effort by making contributions to the specifications of the Java Standard Library. Since JML was designed, many common classes in the standard library have been specified. However, lack of library support is still a significant issue for JML. Since much of the Java Standard Library remains unspecified, JML users may receive compile errors or false violations while working with relatively simple software. Furthermore, there is no way to check if an unspecified library is misused or fails. JML is a mature language and the current toolkit, OpenJML, offers significant functionality. However, full specification of the Java Standard Library is needed to support real world applications.
Chapter 2: An Overview of JML

JML aims to provide a convenient mechanism to formally specify software written in Java. It is used to describe how methods behave and what guarantees they make. More plainly, JML formally defines the API of a program or part of a program [3].

The concept of design by contract (see appendix), first introduced with the Eiffel language, was a strong motivator during the creation of JML [4]. To this end, JML implements Hoare Logic (see appendix) by providing interfaces to define both preconditions and postconditions. Method specifications are rounded out by the additional ability to define the domain of a method’s side effects (i.e. frame axiom) or lack thereof [5]. In addition to design by contract, JML borrows ideas from the model-based approach to software verification by providing the capability to define attributes like model variables and invariant definitions. These attributes exist outside of the scope of the actual Java code and can only be used for verification [4].

2.1 Language Format

In order to make the language accessible to Java developers, JML uses familiar Java expressions and syntax where it is feasible. JML’s implementation of simple constructs like Boolean operators and semicolons, as well as more sophisticated features like for-loops, will be familiar to the average Java developer. Adding to its accessibility is the fact that JML annotations can be written in line with the Java code which they specify. This format provides a natural entry point to Java developers wishing to learn JML.

JML is defined within a .java file by containing the specification within Java comments and using the at-sign to further distinguish a specification from general comments. As such, JML specifications exist either after //@ or between /*@ and @*/. Method specifications are most commonly defined directly above a method declaration, similar to Javadoc annotations, though they can play a role throughout the method body as well. Other specifications, such as invariants, may be defined as static or instance attributes at the class level.

The JML specification of a Java class may also live in a parallel .jml file. Such files exist purely for specification purposes and are composed of class and method declarations, but not definitions. Method specifications are prepended to method definitions and class specifications are written independently of any method, just as is done in a Java source file [6]. This feature is useful if the author of the specifications does not have access to modify the source code of the software
that they are specifying. While writing specifications within Java source files provides a quick and easy way for new users to get started, some users may find that separate specification files declutter the page of implementation details and free them to focus on the interface they are working with.

2.2 Visibility
When specifying a method, incoming parameters are accessible with normal Java syntax and the method result can be accessed with the `\result` keyword. In addition, JML provides the `\old` keyword to reference the state of a variable before the method began execution. This is often useful when describing a mutation that occurs over the course of a method’s execution.

Class attributes are visible if their access modifier is equally or less restrictive than that of the specification in which they are used [5]. By default, a specification has equivalent visibility to the method which it specifies, so only attributes that are less restricted than the method may be accessed. This is because describing the behavior of a private variable in a public specification may cause a method’s implementation to become undesirably visible through its interface. While this constraint is well founded, in practice it is often necessary to describe variables that are therefore inaccessible. To resolve these competing interests, JML allows for variables and methods to have a second access modifier injected into their declaration that overrides the original for the purpose of specification. For example, the variable `number` has private visibility as far as Java is concerned, but public visibility to JML specifications:

```java
private /*@ spec_public */ int number;
```

This allows `number` to be used within the public specifications.

2.3 Preconditions
Most method specifications include some set of preconditions. Preconditions are required to hold in order for the method to provide the desired results. Generally, these requirements are constraints on class variables and/or the method’s parameters. These preconditions are written as Java-like predicates and qualified with the `requires` key word. One familiar example is a method that requires a parameter to not be null. In JML this precondition can be expressed as follows.
Another common use of the requires clause is bounds checking. For instance, consider the get(index) method of a Java Collection. Clearly, the method cannot be expected to return an element if the given index does not exist in the underlying Collection. The JML specification of this precondition can be expressed as follows.

```java
//@ requires 0 <= index && index < this.size;
public get(int index);
```

### 2.4 Postconditions

A method’s postconditions are predicates that are guaranteed to hold after the method completes execution provided that its preconditions were met at the beginning of execution. Postconditions are expressed with the ensures keyword in a similar fashion to preconditions. Postconditions are often used to describe what a method returns or the state that an object will be in after a method executes. Assuming the get(index) method referenced above is intended to return an element of an underlying array, a postcondition could be expressed as follows:

```java
//@ ensures \result == internalArray[index];
public get(int index);
```

When combined, preconditions and postconditions define a formal contract between a method and its caller.

### 2.5 Method Side Effects and Purity

Variables that may be altered by a method should be identified by the assignable clause. For instance, the add(Element) method of a collection probably has a side effect of incrementing the size of the collection and therefore, the specification may include:

```java
//@ assignable this.size;
public add(Object o);
```
However, the method is not expected to modify existing elements in the collection, so they would not be marked as assignable.

A method that has no side effects is considered pure and may be declared with the `pure` keyword. The `size()` method of a collection, for instance, may be marked pure because it is expected to return an integer without modification to the underlying data structure.

### 2.6 Specification Side Effects

Any specification or verification scheme must not alter the behavior of the software to which it is applied. In JML, specifications execute Java code as they are checked. Specifications are permitted to reference variables and methods in the usual way, even variables and methods which they specify. This allows specifications to easily build on one another, but also introduces the possibility of side effects. In order to guarantee that specifications do not alter the software that they specify, JML forbids specifications from modifying variables or executing methods with side effects [7] [5]. The following postcondition for the `add` method of a collection causes a side effect as it is checked and is therefore illegal.

```java
//@ ensures --size == \old(size);
public add(Object o);
```

This postcondition checks that `size` has been incremented, but it also decrements `size` back to its original value in the process. Therefore, executing this method with and without its specification will result in different behavior. The issue can be rectified as follows:

```java
//@ ensures size - 1 == \old(size);
public add(Object o);
```

To help ensure that specifications do not have side effects, strict JML requires that specifications only call methods that have been marked as pure (see 2.5). In the collection example, specifications may refer to `this.size` or alternatively call the pure method `this.size()`. However, a specification may never call `this.sort()` since it alters the object’s state.
2.7 Class Invariants

In addition to method specification, JML provides some features for describing the behavior of classes as a whole. One such feature is class invariants which are defined with the `invariant` keyword and not attached to any method. Class invariants are predicates defined in the usual JML fashion. Here we consider instance invariants which must be established when the class is constructed and hold for every visible state of an object. A visible state is loosely defined as the state of an object before or after method execution (see the JML Reference Manual [5] for full definition). That is, a method may temporally break an invariant, but it must be reestablished before execution completes. It should be noted that class invariants are expected to hold even if a method results in an exception.

Generally, class invariants may be thought of as universal preconditions and postconditions which apply to every method within a class. The exception to this rule is methods marked with the `helper` keyword which may only be applied to private methods or constructors. These methods are exempt from maintaining invariants in their post-state.
Chapter 3: Tools and Applications

JML is not itself a method of formal verification. Rather it is a language that describes the behavior of Java software precisely enough that tools may be used to test or verify that behavior. Some applications of JML include runtime checking, static checking, automated unit test generation, and automated documentation generation [3]. Various tools have been developed based on JML to implement these ideas. While some of these tools have matured more than others, it must be noted here that they are all experimental and are not often used outside of academia.

3.1 Runtime Assertion Checking

Perhaps the most intuitive use for JML specifications is runtime assertion checking. Runtime assertion checking (RAC) involves compiling specified classes with a special compiler. When a specified class is compiled, the bytecode includes checks which verify assertions extracted from the JML specifications [8]. For instance, before a method is run, the parameters being passed to it and relevant class variables are checked against the method’s preconditions. If a precondition is not satisfied, the program may log a warning or throw an exception. Likewise, assertions that relate to postconditions and invariants are constantly checked as the program runs. There is some obvious overhead incurred when running RAC compiled software due to this continuous checking.

Since JML specifications are forbidden from having side effects, a properly implemented RAC compiler will guarantee that the only difference between its output and that of javac is the warnings and errors which might be produced during execution. Therefore, source code can be compiled for RAC during the testing phase, then recompiled with a standard compiler for deployment without changing the behavior of a program [3]. This means that specifying a software package with JML and testing with RAC does not translate to a performance loss when it is released.

A disadvantage of RAC is that only cases which are exercised are checked. This means that it is up developers and testers to try all of the expected paths of execution in order to see if there will be an assertion failure. Perhaps some bad input will cause a method to violate an assertion and behave in an undesired way. If the method is never actually called with that bad input during testing, RAC will not give the tester any warning. This makes RAC similar to unit testing in that the tester must be able to anticipate likely points of failure that need to be checked.
3.2 Static Checking

Static checking attempts to verify specifications outside of actual program execution. This may take many different forms from basic type checking of JML specifications to mathematically proving that specifications hold.

In order to prove that software meets its specification, static checking tools must break specifications down to verification conditions. In the context of JML, //@requires clauses are translated to assumptions and //@ensures clauses are translated to assertions. These assumptions and assertions, along with the code they specify, are converted to verification conditions [8]. These verification conditions are then passed to an external SMT solver (see appendix for discussion on SMTs and SMT solvers). The SMT solver will either report that the verification condition holds, or return a counterexample which shows an inconsistency between a method’s implementation and its specification.

The effectiveness of static checking depends on the user in the sense that it relies on the quality and correctness of the JML specifications which they produce. It does not, however, rely on the user to exercise all specified methods. This is a significant advantage over runtime checking.

3.3 Automated Unit Test Generation

Automated unit testing is the process of generating and executing valid unit tests based on a methods specification. JML specifications contain all of the necessary context to generate these tests. First, tools generate a large set of tests based on what the user defines as acceptable data (e.g. ints {-1,0,1} and Strings {null, “”}) [2]. Then, tests are compared against a method’s preconditions. If the generated parameters do not meet the preconditions the test is deemed invalid. Otherwise, the test can be run as a standard JUnit test using invariants and postconditions as JUnit assert statements. This can be done on the scale of millions of tests and does not require the user to manually maintain the test suite [2].

3.4 Automated Documentation Generation

Good JML specifications and good JavaDocs should really say the same thing in different languages. This make generating human readable documentation a great use for JML specifications. Tools have been developed which parse JML into HTML in a similar manner to JavaDocs [3]. With this technique, the interface of a method only needs to be defined and
maintained in once place. This also promotes documentation that describes the interface more completely.

3.5 OpenJML
OpenJML is the current toolkit for working with JML. It has been built on top of previous JML tools and is actively maintained and enhanced. OpenJML is available through both a command line interface and a plugin for the Eclipse IDE.

3.5.1 Type-Checking
OpenJML includes a robust JML parser and type-checker. When run against a file containing JML, the type-checker will mark syntax mistakes, invalid keywords, and inaccessible variables. This feature is particularly well integrated with the Eclipse GUI.

3.5.2 Runtime Assertion Checking
The OpenJML project maintains a modified version of the OpenJDK compiler which is used to compile classes with embedded RAC assertions. Files may be compiled for RAC through the command line interface or marked for RAC within Eclipse. Files which have been marked are automatically compiled for RAC on save as is the normal Eclipse workflow. Upon execution, any violations are printed to the console.

3.5.3 Extended Static Checking
OpenJML also supports Extended Static Checking (ESC) and is the successor to the previous Java static checker ESC/Java2. Verification conditions produced by OpenJML are represented in SMT-LIB format which can be used by external SMT solvers. This feature is particularly well implemented in the GUI. Progress is shown while running ESC and methods are highlighted according to success or failure. Counterexamples are made available where possible.
3.5.4 Limitations of OpenJML

Unfortunately, some features that have been implemented in previous tools are not currently supported by the OpenJML toolkit. The previous documentation tool, jmldoc, is no longer maintained and this functionality was not carried over into OpenJML. This is, however, a planned feature for OpenJML. Additionally, there is no built-in functionality for unit test generation. OpenJML is compatible with the test generator JMLUnitNG, however that tool was last updated three years prior to this writing.
Chapter 4: Design

4.1 Considerations When Specifying
When determining if certain software can be specified, it must be understood that OpenJML, the most advanced JML toolkit, is still in development. Many Java features are not yet supported and may cause logic errors or even fail to compile. Even some basic Java features like floating point numbers are not fully supported.

The goal of this project was to select a standard library that had not been specified with JML, verify its functionality, and contribute the specification back to the central Git repository. Numerous errors were encountered while assessing the state of such libraries. For instance, a significant number of classes were unworkable because of an OpenJML bug that prevents any class which uses the Java 7 diamond operator (or inherits from any superclass that does) from being compiled. Additionally, many classes were ineligible because of insufficient support for the classes which they import or inherit. Many collections, for example, fail type-check because of issues in the specification for java.util.Vector.

4.2 Libraries Selected
java.lang.StringBuilder and the associated java.lang.AbstractStringBuilder were selected because they do not depend on complex objects or features. They are, however, intricate enough to demonstrate many of the capabilities of OpenJML. Furthermore, StringBuilder is a commonly used class, so being able to use it with OpenJML should be helpful to the community.

4.3 Layout
This project uses external .jml specifications (see 2.1) for java.lang.StringBuilder and java.lang.AbstractStringBuilder. External specifications are used in order to describe the interface of StringBuilder without being tied to any one implementation. Additionally, the use of private attributes is limited to model abstractions and only public methods are specified. This design means that the specifications correctly apply to the implementations in the official Oracle JDK as well as OpenJDK even though some private functionality is handled differently.

The model attributes which define the state of the class are defined publicly in AbstractStringBuilder.jml and are therefore accessible to public specifications throughout
StringBuilder as well. Class invariants imposed on these model variables are defined likewise and therefore apply to the StringBuilder specification as well.

Method specifications were written for AbstractStringBuilder first, then extended in StringBuilder as needed. Heavyweight (i.e. full) method specifications that account for all preconditions were used where possible. While the source of the specified classes is visible in this case, an effort was made to write specifications against the documentation and infer as little behavior as possible from the source. This is good practice to prevent specifying behavior that is not guaranteed and may change in the future.

In addition to the specifications, this project contains a driver program that was designed to test the libraries against both RAC and ESC. For RAC, use cases were selected for each method that represent fringe cases as well as exceptional behavior. Each behavior statement in the specification has an associated call in the driver program that meets its preconditions. For ESC, assert statements have been added to the driver to require that key conditions can be proven.

4.4 Development Environment
The most current version of OpenJML was used throughout this project, ranging from 0.8.6 to 0.8.11. Work was done in a Windows environment primarily using Eclipse Neon.3 and the OpenJML plugin. The library specifications were compiled for Java 1.8 and checked against the implantations in the Oracle Java SE JDK as well as OpenJDK. The SMT solver used for static checking was Z3 4.4.0.
Chapter 5: Implementation

5.1 Modeling StringBuilder

AbstractStringBuilder is a mutable representation of a String. This is accomplished by maintaining an underlying char[] value where the characters that make up the String are stored. Another attribute, int count, is used to mark the last index of value that is use. For specification purposes, equivalent attributes are added to the AbstractStringBuilder model. This is done by declaring a new model attribute and relating it to the class attribute. An additional model attribute representing the capacity of the object is added to help reason about behavior.

```java
//@ public model char[] modelValue;
//@ public model int modelCapacity;
/*@ spec_public @*/ char[] value; //@ in modelValue; in modelCapacity;
//@ represents modelValue = this.value;
//@ represents modelCapacity = this.value.length;

//@ public model int modelStringLength;
/*@ spec_public @*/ int count; //@ in modelStringLength;
//@ represents modelStringLength = this.count;
```

Next, ghost attributes are defined. These attributes are commonly used values throughout the specification:

```java
//@ public ghost int MAX_CAPACITY = Integer.MAX_VALUE;
//@ public ghost char[] nullChars = {'n', 'u', 'l', 'l'};
//@ public ghost char[] trueChars = {'t', 'r', 'u', 'e'};
//@ public ghost char[] falseChars = {'f', 'a', 'l', 's', 'e'};
```

Finally, class invariants are applied to the model attributes based on the class’s documentation. First, the object’s capacity must be nonnegative and is bound by MAX_CAPACITY:

```java
//@ public invariant 0 <= modelCapacity &&
//@ modelCapacity <= MAX_CAPACITY;
```

Next, the length of the string must be nonnegative and is bound by modelCapacity:

```java
//@ public invariant 0 <= modelStringLength &&
//@ modelStringLength <= modelCapacity;
```
Finally, the unused portion of the array must be null-filled:

```java
//@ public invariant (\forall int i; modelStringLength <= i &&
//@ i < modelCapacity; value[i] == '\0');
```

### 5.2 Specifying Interfaces

Ideally, a method’s JML specification is a formal translation of its documentation. This was done as much as possible throughout this project. Consider the following JavaDoc for the constructor `public StringBuilder(int capacity)`:

```java
/**
 * Constructs a string builder with no characters in it and an
 * initial capacity specified by the {code capacity} argument.
 *
 * @param capacity the initial capacity.
 * @throws NegativeArraySizeException if the {code capacity}
 *         argument is less than {code 0}.
 */
 public StringBuilder(int capacity);
```

Two normal postconditions are apparent from the first sentence. First, it is stated that there are no characters in the StringBuilder, so the postcondition `modelStringLength == 0` should hold. Second, the initial capacity of the StringBuilder is defined by the incoming argument `capacity`, so the postcondition `modelCapacity == capacity` should hold as well. The documentation does not mention any requirements for normal behavior to be applied as normal preconditions, but it does indicate an exceptional behavior with the `@throws` clause. The precondition `capacity < 0` should cause the exceptional postcondition `signals NegativeArraySizeException`. Since `capacity < 0` is the only documented precondition that does not exhibit normal behavior, the specification should negate that predicate to indicate that all other inputs will result in normal behavior. Finally, we arrive at a JML specification that fully defines the behavior of this constructor:
Note that no attributes have been marked as assignable because all class attributes are implicitly assignable for constructors.

Unfortunately, documentation is not always complete. Some cases may require the specifier to infer part of a method’s behavior. Consider the sparse documentation for the constructor `AbstractStringBuilder(int capacity)` which reads as follows:

```java
/**
 * Creates an AbstractStringBuilder of the specified capacity.
 */
```

This presents an obvious postcondition of `modelCapacity == capacity`, but does not include any requirements that can be translated into preconditions. In this case, it must be inferred that capacity must be nonnegative.
Consider the following JavaDoc and associated specification for the method public delete(int start, int end):

```java
/**
 * Removes the characters in a substring of this sequence.
 * The substring begins at the specified @code start} and extends to
 * the character at index @code end - 1} or to the end of the
 * sequence if no such character exists. If
 * @code start} is equal to @code end}, no changes are made.
 * @param start The beginning index, inclusive.
 * @param end The ending index, exclusive.
 * @return This object.
 * @throws StringIndexOutOfBoundsException if @code start} is negative, greater than @code length()}, or
 * greater than @code length{.}
 */
public AbstractStringBuilder delete(int start, int end);
```

The documentation obviously describes exceptional and normal behavior for this method. The exceptional precondition expressed on line 20 is easily extracted from the @throws clause of the documentation. In previous examples, the exceptional postcondition was negated to arrive at the
normal precondition. This can be seen here as well on lines 2 and 11. In this case, however, this is not the complete precondition because the documentation actually expresses two distinct normal behaviors. Under normal behavior, the deletion may extend to end – 1 or to modelStringLength if end is out of bounds. This bifurcation is expressed by the additional preconditions on lines 3 and 12.

Both normal behaviors have the same possible side effects, but slightly different postconditions. When comparing lines 6 and 15, we see that both behaviors should reduce modelStringLength by the number of elements deleted. Whether the number of elements deleted is equal to (end - start) or (modelStringLength - start) is determined by the additional preconditions mentioned earlier. Similarly, lines 7 and 16 both state that elements past the deleted segment should be shifted left by the number of elements deleted, but must use different expressions to represent the number of elements deleted.

5.3 Runtime Assertion Checking
Runtime Assertion Checking is accomplished by compiling the driver in RAC mode and executing it. As the program runs, each specified method is called with a series of carefully chosen arguments. Since RAC only checks conditions which are explicitly exercised, it is important to choose a wide range of test cases. This includes testing calling methods from different states as well as with different arguments. At a minimum, methods should be called in ways that satisfy the preconditions for each of the behaviors defined in their specification. Ideally, relevant edge cases are also checked as is done with traditional testing methods.

As methods are called, RAC checks to see which precondition is met. After the method has finished executing, the associated postconditions are checked against the actual object. Invariants are also checked for all visible states (see 2.7).

5.4 Extended Static Checking
Extended Static Checking is run against the driver as well. The various assert statements throughout the driver place additional burden on ESC. ESC must prove that postconditions and invariants are strong enough to satisfy these additional statements.
Chapter 6: Conclusion

6.1 Accomplishments

The primary accomplishments of this project are the specifications for java.lang.AbstractStringBuilder and java.lang.StringBuilder. These specifications add StringBuilder support to OpenJML. For instance, without the new specification, the following simple code is incorrectly marked as invalid by ESC citing an impossible null dereference:

```java
StringBuilder sb = new StringBuilder();
sb.append("StringBuilder")\toString();
```

This code generates a false positive citing an impossible negative array length:

```java
StringBuilder sb = new StringBuilder();
char[] copy = new char[sb.capacity()];
```

The same types of errors apply to the use of StringBuilder within specifications. This example is declared invalid because the initial value of `sb.length()` is undefined. OpenJML also logs a warning that `sb.length()` should not be used in specifications because it is not pure:

```java
StringBuilder sb = new StringBuilder();
//@ maintaining sb.length() < 4;
for (int i = 1; !sb.toString().equals("123"); i++) {
    sb.append(i);
}
```

These are precisely the types of nonintuitive errors that make it hard to work with unspecified libraries. All of these examples work correctly with RAC and ESC when the new StringBuilder specifications are included.

Additionally, this project demonstrates a real-world use case of JML by specifying and verifying a useful data structure. The techniques used are applicable to the specification of public
interfaces in general and could be expanded to future work. The project also establishes that there are not flaws within StringBuilder with a high degree of confidence.

6.2 Future Work
On its face, JML is an accessible language. The common Java syntax, ability to write specifications in-line with source code, and support for partially specified behavior with lightweight specifications make it very appealing. OpenJML has also done a great deal to enhance this by adding RAC and ESC features into an Eclipse plugin. However, there is still much work to be done before JML can be easily integrated into the workflow of most Java developers. While simple examples work well, unsupported Java features, unspecified libraries, and cryptic errors make real world specification more time consuming than expected. The original scope of this project needed to be narrowed considerably because of these factors.

More library specifications are needed. Without support for all of the standard libraries, it will be hard to integrate JML into most existing code bases. Furthermore, developers will expect consistent access to Java features like the diamond operator mentioned earlier. OpenJML is being actively improved at an impressive pace considering the number of contributors. Numerous updates with enhancements and bug fixes have been released during the course of this project. The recent integration with Java 8 has also been a significant improvement to the OpenJML workflow and will help keep JML accessible into the future.
References


Appendix

Design by Contract

The concept of Design by Contract states that software should be designed based on its interfaces and that these interfaces should be formally specified. It was first introduced with the Eiffel programming language and is the application of Floyd-Hoare logic to software design.

Floyd-Hoare Logic (Hoare Logic)

The center piece of Hoare logic is Hoare Triples. Hoare Triples describe a precondition, an event, and a postcondition. Logic rules can be used to manipulate these triples to describe complex behavior in software.

Satisfiability Modulo Theories (SMT)

SMTs are expressions with a set of constraints, a set of variables, and some target condition. To solve an SMT is to determine whether or not the variables can possibly be manipulated in a way that the expression will meet the target condition. If so, the SMT is said to be satisfiable. If not, the SMT is said to be unsatisfiable. SMT solvers are tools which attempt to determine whether or not an SMT is satisfiable. Doing this in efficient ways is an area of much research [9].
This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

package java.lang;

/* JML's specification of AbstractStringBuilder.
 * @author Mike Rawding
 */

abstract class AbstractStringBuilder implements Appendable, CharSequence {

    //@ public model char[] modelValue;
    //@ public model int modelCapacity;
    //@ spec_public @*/ char[] value; //@ in modelValue; in modelCapacity;
    //@ represents modelValue = this.value;
    //@ represents modelCapacity = this.value.length;

    //@ public model int modelStringLength;
    //@ spec_public @*/ int count; //@ in modelStringLength;
    //@ represents modelStringLength = this.count;

    //@ public model int MAX_CAPACITY = Integer.MAX_VALUE;
    //@ public model char[] nullChars = {'n', 'u', 'l', 'l'};
    //@ public model char[] trueChars = {'t', 'r', 'u', 'e'};
    //@ public model char[] falseChars = {'f', 'a', 'l', 's', 'e'};

    //@ public invariant 0 <= modelCapacity && modelCapacity <= MAX_CAPACITY;
    //@ public invariant 0 <= modelStringLength && modelStringLength <= modelCapacity;
    //@ public invariant (\forall int i; modelStringLength <= i && i < modelCapacity; modelValue[i] == '\0');

    //@ ensures modelStringLength == 0;
    AbstractStringBuilder(); //1

    //@ normal_behavior

AbstractStringBuilder.jml

@   requires 0 <= capacity;
@   ensures modelCapacity == capacity;
@   ensures modelStringLength == 0;
@ also
@   requires capacity < 0;
@   signals_only Exception;
@*/
/*@ pure helper@*/
AbstractStringBuilder(int capacity); //2
//@ ensures \result == modelStringLength;
public /*@ pure @*/ int length(); //3
//@ ensures \result == modelCapacity;
public /*@ pure @*/ int capacity(); //4
/*@  public normal_behavior
@    requires minimumCapacity > modelCapacity;
@    assignable modelValue;
@    ensures modelCapacity >= minimumCapacity;
@  also
@  public normal_behavior
@    requires minimumCapacity <= modelCapacity;
@    assignable \nothing;
@*/
public void ensureCapacity(int minimumCapacity); //5
/*@ public normal_behavior //NOTE: ESC precondition errors do to invariant bug
@    requires modelStringLength < modelCapacity;
@    assignable modelValue;
@    //may reduce size modelCapacity, but not required
@    also
@  public normal_behavior
@    requires modelStringLength == modelCapacity;
@    assignable \nothing;
@*/
public void trimToSize(); //9
/*@ public normal_behavior
@    requires modelStringLength <= newLength;
@    assignable modelValue;
@    assignable modelStringLength;
@    ensures modelStringLength == newLength;
@ also
@  public normal_behavior
@    requires 0 <= newLength && newLength < modelStringLength;
@ also
@  public normal_behavior
@    requires 0 <= newLength && newLength < modelStringLength;
AbstractStringBuilder.jml

@ assignable modelValue[newLength .. modelCapacity - 1];
@ assignable modelStringLength;
@ ensures modelStringLength == newLength;
@ ensures modelCapacity == old(modelCapacity);
@ also
@ public exceptional_behavior
@   requires newLength < 0;
@   signals_only StringIndexOutOfBoundsException;
@*/
public void setLength(int newLength); //10

/*@-ESC@ public normal_behavior
@   requires 0 <= index && index < modelStringLength;
@   ensures result == modelValue[index];
@*/
@*/
@ also
@ public exceptional_behavior
@   requires index < 0 || index >= modelStringLength;
@   signals_only StringIndexOutOfBoundsException;
@*/
public /*@ pure @*/ char charAt(int index); //11

public int codePointAt(int index); //12
public int codePointBefore(int index); //13
public int codePointCount(int beginIndex, int endIndex); //14
public int offsetByCodePoints(int index, int codePointOffset); //15

/*@ public normal_behavior
@   requires srcBegin >= 0 && srcEnd >= 0 && srcEnd <=
modelStringLength && srcBegin <= srcEnd;
@   requires srcEnd-srcBegin <= dst.length-dstBegin;
@   requires dst != null;
@   ensures (forall int i; srcBegin <= i && i < srcEnd; modelValue[i]
== dst[i + dstBegin]);
@ also
@ public exceptional_behavior
@   requires !(srcBegin >= 0 && srcEnd >= 0 && srcEnd <=
modelStringLength && srcBegin <= srcEnd);
@   signals_only StringIndexOutOfBoundsException;
@ also
@ public exceptional_behavior
@   requires !(srcEnd-srcBegin <= dst.length-dstBegin);
@   signals_only ArrayIndexOutOfBoundsException;
AbstractStringBuilder.jml

136     @
137     */
138     public  /*@ pure @*/  void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin); //16
139
140     /*@ public normal_behavior
141     @   requires 0 <= index && index < modelStringLength;
142     @   assignable modelValue[index];
143     @*/
144     //ESC@   ensures modelValue[index] == ch;
145     /*@ public exceptional_behavior
146     @   requires (index < 0 || index >= modelStringLength);
147     @   signals_only StringIndexOutOfBoundsException;
148     @*/
149     public void setCharAt(int index, char ch); //17
150
151     /*@ public normal_behavior
152     @   requires obj == null;
153     @   assignable modelValue;
154     @   assignable modelValue[modelStringLength - nullChars.length ..
155     modelStringLength];
156     @   assignable modelStringLength;
157     @   assignable modelCapacity;
158     @   ensures (\forall int i; 0 <= i && i < nullChars.length;
159     value[\old(modelStringLength)+i] == nullChars[i]);
160     @   ensures modelStringLength == \old(modelStringLength) +
161     nullChars.length;
162     @   ensures \result == this;
163     @ also
164     @ public normal_behavior
165     @   requires obj != null;
166     @   assignable modelValue;
167     @   assignable modelValue[modelStringLength -
168     String.valueOf(obj).length() .. modelStringLength];
169     @   assignable modelStringLength;
170     @   assignable modelCapacity;
171     @   //ensures (\forall int i; 0 <= i && i <
172     String.valueOf(obj).length(); modelValue[\old(modelStringLength)+i] ==
173     String.valueOf(obj).charAt(i));
174     @   //ensures modelStringLength == \old(modelStringLength) +
175     String.valueOf(obj).length();
176     @   ensures \result == this;
177     @*/
178     public AbstractStringBuilder append(/*@ nullable */ Object obj); //18
179
180     /*@ public normal_behavior
181     @   requires str == null;
182     @ also
183     @ public normal_behavior
184     @   requires str != null;
185     @*/
AbstractStringBuilder.jml

175   @   assignable modelValue;
176   @   assignable value[modelStringLength - modelCapacity ..
       modelStringLength];
177   @   assignable modelStringLength;
178   @   assignable modelCapacity;
179   @   ensures (\forall int i; 0 <= i && i < nullChars.length;
       modelValue[\old(modelStringLength)+i] == nullChars[i]);
180   @   ensures modelStringLength == \old(modelStringLength) +
       nullChars.length;
181   @   ensures \result == this;
182   @ also
183   @ public normal_behavior
184   @   requires str != null;
185   @   assignable modelValue;
186   @   assignable modelValue[modelStringLength - str.length() ..
       modelStringLength];
187   @   assignable modelStringLength;
188   @   assignable modelCapacity;
189   @   ensures (\forall int i; 0 <= i && i < str.length();
       modelValue[\old(modelStringLength)+i] == str.charAt(i));
190   @   ensures modelStringLength == \old(modelStringLength) + str.length();
191   @   ensures \result == this;
192   @*/
193   public AbstractStringBuilder append(/*@ nullable @*/ String str); //19
194
195   /*@ public normal_behavior
196   @   requires sb == null;
197   @   assignable modelValue;
198   @   assignable modelValue[modelStringLength - nullChars.length ..
       modelStringLength];
199   @   assignable modelStringLength;
200   @   assignable modelCapacity;
201   @   ensures (\forall int i; 0 <= i && i < nullChars.length;
       modelValue[\old(modelStringLength)+i] == nullChars[i]);
202   @   ensures modelStringLength == \old(modelStringLength) +
       nullChars.length;
203   @   ensures \result == this;
204   @ also
205   @ public normal_behavior
206   @   requires sb != null;
207   @   assignable modelValue;
208   @   //assignable modelValue[modelStringLength - sb.toString.length() ..
       modelStringLength];
209   @   assignable modelStringLength;
210   @   assignable modelCapacity;
211   @   //ensures (\forall int i; 0 <= i && i < sb.toString().length();
       modelValue[\old(modelStringLength)+i] == sb.toString().charAt(i));
AbstractStringBuilder.jml

212     @   //ensures modelStringLength == \old(modelStringLength) +
213           sb.toString().length();
214     @   ensures \result == this;
215     @*/
216     public AbstractStringBuilder append(StringBuffer sb); //20
217
218     AbstractStringBuilder append(/*@ nullable @*/ AbstractStringBuilder asb);
219     //21
220
221     @@ public normal_behavior
222     @   requires s == null;
223     @   assignable modelValue;
224     @   assignable modelValue[modelStringLength - nullChars.length ..
225             modelStringLength];
226     @   assignable modelStringLength;
227     @   assignable modelCapacity;
228     @   ensures (\forall int i; 0 <= i &\& i < nullChars.length;
229             modelValue[\old(modelStringLength)+i] == nullChars[i]);
230     @   ensures modelStringLength == \old(modelStringLength) +
231           nullChars.length;
232     @   ensures \result == this;
233     @ also
234     @@ public normal_behavior
235     @   requires s != null;
236     @   assignable modelValue;
237     @   assignable modelValue[modelStringLength - s.length() ..
238             modelStringLength];
239     @   assignable modelStringLength;
240     @   assignable modelCapacity;
241     @   ensures (\forall int i; 0 <= i &\& i < s.length();
242             modelValue[\old(modelStringLength)+i] == s.charAt(i));
243     @   ensures modelStringLength == \old(modelStringLength) + s.length();
244     @   ensures \result == this;
245     @*/
246     public AbstractStringBuilder append(/*@ nullable @*/ CharSequence s); //22
247
248     private AbstractStringBuilder appendNull(); //23
249
250     /*@ public normal_behavior
251     @   requires s == null;
252     @   requires 0 <= start && start <= end &\& end <= nullChars.length;
253     @   assignable modelValue;
254     @   assignable modelValue[modelStringLength - nullChars.length ..
255             modelStringLength];
256     @   assignable modelStringLength;
257     @   assignable modelCapacity;
258     @   ensures (\forall int i; 0 <= i &\& i < nullChars.length;
AbstractStringBuilder.jml

@   ensures modelStringLength == \old(modelStringLength) +
    nullChars.length;
252   @   ensures \result == this;
253   @ also
254   @ public normal_behavior
255   @   requires s != null;
256   @   requires 0 <= start && start <= end; //&& end <= s.length() //NOTE -
    marked as a possible null dereference on s. Probable OpenJML bug.
257   @   assignable modelValue;
258   @   assignable modelValue[modelStringLength - (end - start) ..
    modelStringLength];
259   @   assignable modelStringLength;
260   @   assignable modelCapacity;
261   @ ensures modelStringLength == \old(modelStringLength) + (end -
    start);
262   @ ensures (\forall int i; 0 <= i && i < end - start;
    modelValue[\old(modelStringLength)+i] == s.charAt(start + i));
263   @ ensures \result == this;
264   @ also
265   @ public exceptional_behavior
266   @   requires s != null;
267   @   requires start < 0 || start > end; // || end > s.length(); //NOTE -
    marked as a possible null dereference on s. Probable OpenJML bug.
268   @   signals_only IndexOutOfBoundsException;
269   @ also
270   @ public exceptional_behavior
271   @   requires s == null;
272   @   requires start < 0 || start > end || end > nullChars.length;
273   @   signals_only IndexOutOfBoundsException;
274   @*/
275   public AbstractStringBuilder append(/*@ nullable */CharSequence s, int
    start, int end); //24
276
277   public AbstractStringBuilder append(char[] str); //25
278
279   public AbstractStringBuilder append(char str[], int offset, int len); //26
280
281   /*@ public normal_behavior
282   @   requires b == true;
283   @   assignable modelValue;
284   @   assignable modelValue[modelStringLength - trueChars.length ..
    modelStringLength];
285   @   assignable modelCapacity;
286   @   assignable modelStringLength;
287   @   ensures modelStringLength == \old(modelStringLength +
    trueChars.length);
AbstractStringBuilder.jml

288    @   ensures (\forall int i; 0 <= i && i < trueChars.length;
    modelValue[\old(modelStringLength)+i] == trueChars[i]);
289    @   ensures \result == this;
290    @ also
291    @ public normal_behavior
292    @   requires b == false;
293    @   assignable modelValue;
294    @   assignable modelValue[modelStringLength - falseChars.length ..
    modelStringLength];
295    @   assignable modelCapacity;
296    @   assignable modelStringLength;
297    @   ensures modelStringLength == \old(modelStringLength +
    falseChars.length);
298    @   ensures (\forall int i; 0 <= i && i < falseChars.length;
    modelValue[\old(modelStringLength)+i] == falseChars[i]);
299    @   ensures \result == this;
300    @*/
301    public AbstractStringBuilder append(boolean b); //27
302
303    /*@
304    @ assignable modelValue;
305    @ assignable modelValue[modelStringLength - 1 .. modelStringLength];
306    @ assignable modelStringLength;
307    @ assignable modelCapacity;
308    @ ensures modelStringLength == \old(modelStringLength) + 1;
309    @ ensures \result == this;
310    @*/
311    /*-ESC@
312    @ ensures modelValue[\old(modelStringLength)] == c;
313    @*/
314    public AbstractStringBuilder append(char c); //28
315
316    /*@
317    @ assignable modelValue;
318    @ //assignable modelValue[modelStringLength - String.valueOf(i).length()..
    modelStringLength];
319    @ assignable modelStringLength;
320    @ assignable modelCapacity;
321    @ //ensures (\forall int j; 0 <= j && j < String.valueOf(i).length();
    modelValue[\old(modelStringLength)+j] == String.valueOf(i).charAt(j));
322    @ //ensures modelStringLength == \old(modelStringLength) +
    String.valueOf(i).length();
323    @ ensures \result == this;
324    @*/
325    public AbstractStringBuilder append(int i); //29
326
327    /*@
AbstractStringBuilder.jml

328     @ assignable modelValue;
329     @ //assignable modelValue[modelStringLength - String.valueOf(l).length() .. modelStringLength];
330     @ assignable modelStringLength;
331     @ assignable modelCapacity;
332     @ //ensures (\forall int i; 0 <= i && i < String.valueOf(l).length(); modelValue[\old(modelStringLength)+i] == String.valueOf(l).charAt(i));
333     @ //ensures modelStringLength == \old(modelStringLength) + String.valueOf(l).length();
334     @ ensures \result == this;
335     @*/
336     public AbstractStringBuilder append(long l); //30
337
338     /*@
339     @ assignable modelValue;
340     @ //assignable modelValue[modelStringLength - String.valueOf(obj).length() .. modelStringLength];
341     @ assignable modelStringLength;
342     @ assignable modelCapacity;
343     @ //ensures (\forall int i; 0 <= i && i < String.valueOf(obj).length(); modelValue[\old(modelStringLength)+i] == String.valueOf(obj).charAt(i));
344     @ //ensures modelStringLength == \old(modelStringLength) + String.valueOf(obj).length();
345     @ ensures \result == this;
346     @*/
347     public AbstractStringBuilder append(float f); //31
348
349     /*@
350     @ assignable modelValue;
351     @ //assignable modelValue[modelStringLength - String.valueOf(d).length() .. modelStringLength];
352     @ assignable modelStringLength;
353     @ assignable modelCapacity;
354     @ //ensures (\forall int i; 0 <= i && i < String.valueOf(d).length(); modelValue[\old(modelStringLength)+i] == String.valueOf(d).charAt(i));
355     @ //ensures modelStringLength == \old(modelStringLength) + String.valueOf(d).length();
356     @ ensures \result == this;
357     @*/
358     public AbstractStringBuilder append(double d); //32
359
360     /*@ public normal_behavior
361     @ requires 0 <= start && start <= end && end <= modelStringLength;
362     @ assignable modelValue[start .. modelStringLength];
363     @ assignable modelStringLength;
364     @ ensures modelStringLength == \old(modelStringLength) - (end - start);
AbstractStringBuilder.jml

```java
  @ ensures (\forall int i; start <= i && i < modelStringLength;
    modelValue[i] == \old(modelValue)[i + (end - start)]);
  @ ensures \result == this;
  @ also
  @ public normal_behavior
  @ requires 0 <= start && start <= end && start <= modelStringLength;
  @ requires end > modelStringLength;
  @ assignable modelValue[start .. modelStringLength];
  @ assignable modelStringLength;
  @ ensures modelStringLength == \old(modelStringLength) -
    (modelStringLength - start);
  @ ensures (\forall int i; start <= i && i < modelStringLength;
    modelValue[i] == \old(modelValue)[i + (modelStringLength - start)]);
  @ ensures \result == this;
  @ also
  @ public exceptional_behavior
  @ requires start < 0 || start > end || start > modelStringLength;
  @ signals_only StringIndexOutOfBoundsException;
  @*/
  public AbstractStringBuilder delete(int start, int end); //33

  public AbstractStringBuilder appendCodePoint(int codePoint); //34

 /*@ public normal_behavior
  @ requires 0 <= index && index < modelStringLength;
  @ assignable modelValue[index .. modelStringLength];
  @ assignable modelStringLength;
  @ ensures modelStringLength == \old(modelStringLength) - 1;
  @ ensures (\forall int i; index <= i && i < modelStringLength;
    modelValue[i] == \old(modelValue)[i + 1]);
  @ ensures \result == this;
  @ also
  @ public exceptional_behavior
  @ requires index < 0 || index >= modelStringLength;
  @ signals_only StringIndexOutOfBoundsException;
  @*/
  public AbstractStringBuilder deleteCharAt(int index); //35

 /*@ public normal_behavior
  @ requires 0 <= start && start <= end && start <= modelStringLength;
  @ assignable modelValue[start .. end - 1];
  @ ensures (\forall int i; start <= i && i < end; modelValue[i] ==
    str.charAt(i - start));
  @ ensures \result == this;
  @ also
  @ public exceptional_behavior
  @ requires start < 0 || start > end || start > modelStringLength;
  @*/
```
AbstractStringBuilder.jml

407   @   signals_only StringIndexOutOfBoundsException;
408   */
409   public AbstractStringBuilder replace(int start, int end, String str); //36
410
411   /*@ public normal_behavior
412      @ requires 0 <= start && start <= modelStringLength;
413      @ ensures result.length() == modelStringLength - start;
414      @ ensures (\forall int i; 0 <= i && i < result.length();
415        result.charAt(i) == modelValue[i + start]);
416      @ also
417      @ public exceptional_behavior
418      @ requires start < 0 || start > modelStringLength;
419      @ assignable StringIndexOutOfBoundsException;
420      */
421   public /*@ pure @*/ String substring(int start); //37
422
423   /*@ public normal_behavior
424      @ requires 0 <= start && start <= end && end <= modelCapacity;
425      @ ensures result.length() == end - start;
426      @ ensures (\forall int i; 0 <= i && i < result.length();
427        result.charAt(i) == modelValue[i + start]);
428      @ also
429      @ public exceptional_behavior
430      @ requires start < 0 || end > modelCapacity || start > end;
431      @ assignable StringIndexOutOfBoundsException;
432      */
433   public /*@pure @*/ CharSequence subSequence(int start, int end); //38
434
435   /*@ public normal_behavior
436      @ requires 0 <= start && start <= end && end <= modelCapacity;
437      @ ensures result.length() == end - start;
438      @ ensures (\forall int i; 0 <= i && i < result.length();
439      @ result.charAt(i) == modelValue[i + start]);
440      @ also
441      @ public exceptional_behavior
442      @ requires start < 0 || end > modelCapacity || start > end;
443      @ assignable StringIndexOutOfBoundsException;
444      */
445   public /*@ pure @*/ String substring(int start, int end); //39
446
447   /*@ public normal_behavior
448      @ requires index >= 0 && index <= modelStringLength && offset >= 0 &&
449      @ len >= 0 && offset <= str.length - len;
450      @ assignable modelValue;
451      @ assignable modelCapacity;
452      @ assignable modelValue[index .. modelCapacity - 1];
AbstractStringBuilder.jml

450  @ assignable modelStringLength;
451  @ ensures modelStringLength == old(modelStringLength) + len;
452  @ ensures (\forall int i; index <= i && i < index + len; modelValue[i] == str[i - offset]);
453  @ ensures (\forall int i; index + len <= i && i < modelStringLength; modelValue[i] == old(modelValue)[i + len]);
454  @ ensures \result == this;
455  @ also
456  @ public exceptional_behavior
457  @ requires index < 0 || index > modelStringLength || offset < 0 || len < 0 || offset > str.length - len;
458  @ signals_only StringIndexOutOfBoundsException;
459  @*/
460  public AbstractStringBuilder insert(int index, char[] str, int offset, int len); //40

461  /*@ public normal_behavior
462  @ requires 0 <= offset && offset <= modelStringLength;
463  @ assignable modelValue;
464  @ assignable modelCapacity;
465  @ assignable modelValue[offset .. modelCapacity - 1];
466  @ assignable modelStringLength;
467  @ ensures (\forall int i; offset <= i && i < offset + String.valueOf(obj).length(); modelValue[i] == String.valueOf(obj).charAt(i - offset));
468  @ ensures (\forall int i; offset + String.valueOf(obj).length() <= i && i < modelStringLength; modelValue[i] == old(modelValue)[i + String.valueOf(obj).length()]);
469  @ ensures \result == this;
470  @ also
471  @ public exceptional_behavior
472  @ requires offset < 0 || offset > modelStringLength;
473  @ signals_only StringIndexOutOfBoundsException;
474  @*/
475  public AbstractStringBuilder insert(int offset,/*@ nullable @*/ Object obj); //41

476  /*@ public normal_behavior
477  @ requires str != null;
478  @ requires 0 <= offset && offset <= modelStringLength;
479  @ assignable modelValue;
480  @ assignable modelCapacity;
481  @ assignable modelValue[offset .. modelCapacity - 1];
482  @ assignable modelStringLength;
483  @ ensures modelStringLength == old(modelStringLength) + str.length();
484  @ ensures (\forall int i; offset <= i && i < offset + str.length(); modelValue[i] == str.charAt(i - offset));
485  @ ensures (\forall int i; offset + str.length() <= i && i <

Page 12
AbstractStringBuilder.jml

modelStringLength; modelValue[i] == \old(modelValue)[i + str.length()]);
487   @   ensures \result == this;
488   @ also
489   @ public normal_behavior
490   @ requires str == null;
491   @ requires 0 <= offset && offset <= modelStringLength;
492   @ assignable modelValue;
493   @ assignable modelCapacity;
494   @ assignable modelValue[offset .. modelCapacity - 1];
495   @ ensures modelStringLength == \old(modelStringLength) +
nullChars.length;
496   @ ensures (\forall int i; offset <= i && i < offset +
nullChars.length; modelValue[i] == nullChars[i - offset]);
497   @ ensures (\forall int i; offset + nullChars.length <= i && i <
modelStringLength; modelValue[i] == \old(modelValue)[i + nullChars.length]);
498   @ ensures \result == this;
499   @ also
500   @ public exceptional_behavior
501   @ requires offset < 0 || offset > modelStringLength;
502   @ signals_only StringIndexOutOfBoundsException;
503   @*/
504   public AbstractStringBuilder insert(int offset, /*@ nullable @*/ String
str); //42
505   @@ public normal_behavior
506   @ requires str != null;
507   @ requires 0 <= offset && offset <= modelStringLength;
508   @ assignable modelValue;
509   @ assignable modelCapacity;
510   @ assignable modelValue[offset .. modelCapacity - 1];
511   @ ensures modelStringLength == \old(modelStringLength) + str.length;
512   @ ensures (\forall int i; offset <= i && i < offset + str.length;
modelValue[i] == str[i - offset]);
513   @ ensures (\forall int i; offset + str.length <= i && i <
modelStringLength; modelValue[i] == \old(modelValue)[i + str.length]);
514   @ ensures \result == this;
515   @ also
516   @ public normal_behavior
517   @ requires str == null;
518   @ requires 0 <= offset && offset <= modelStringLength;
519   @ assignable modelValue;
520   @ assignable modelCapacity;
521   @ assignable modelValue[offset .. modelCapacity - 1];
522   @ ensures modelStringLength == \old(modelStringLength) +
nullChars.length;
523   @ ensures (\forall int i; offset <= i && i < offset +
nulChars.length; modelValue[i] == nullChars[i - offset]);
AbstractStringBuilder.jml

525       @   ensures (\forall int i; offset + nullChars.length <= i && i <
      modelStringLength; modelValue[i] == \old(modelValue)[i + nullChars.length]);
526       @   ensures \result == this;
527       @ also
528       @ public exceptional_behavior
529       @   requires offset < 0 || offset > modelStringLength;
530       @   signals_only StringIndexOutOfBoundsException;
531       @*/
532
   public AbstractStringBuilder insert(int offset, char[] str); //43
533
534   /**< public normal_behavior
535       requires s != null;
536       requires 0 <= dstOffset && dstOffset <= modelStringLength;
537       assignable modelValue;
538       assignable modelCapacity;
539       assignable modelValue[dstOffset .. modelCapacity - 1];
540       ensures modelStringLength == \old(modelStringLength) + s.length();
541       ensures (\forall int i; dstOffset <= i && i < dstOffset +
542               s.length(); modelValue[i] == s.charAt(i - dstOffset));
543       @ ensures (\forall int i; dstOffset + s.length() <= i && i <
544               modelStringLength; modelValue[i] == \old(modelValue)[i + s.length()]);
545       @ also
546       @ public normal_behavior
547       @   requires s == null;
548       @   requires 0 <= dstOffset && dstOffset <= modelStringLength;
549       @   assignable modelValue;
550       @   assignable modelCapacity;
551       @   assignable modelValue[dstOffset .. modelCapacity - 1];
552       @   ensures modelStringLength == \old(modelStringLength) +
553               nullChars.length;
554       @   ensures (\forall int i; dstOffset <= i && i < dstOffset +
555               nullChars.length; modelValue[i] == nullChars[i - dstOffset]);
556       @   ensures (\forall int i; dstOffset + nullChars.length <= i && i <
557               modelStringLength; modelValue[i] == \old(modelValue)[i + nullChars.length]);
558       @ also
559       @ public exceptional_behavior
560       @   requires dstOffset < 0 || dstOffset > modelStringLength;
561       @   signals_only StringIndexOutOfBoundsException;
562       @*/
563
   public AbstractStringBuilder insert(int dstOffset, /*@ nullable @*/
      CharSequence s); //44
564
565   public AbstractStringBuilder insert(int dstOffset, CharSequence s, int
      start, int end); //45
AbstractStringBuilder.jml

564 /*@ public normal_behavior
565   @ requires b == true;
566   @ requires 0 <= offset && offset <= modelStringLength;
567   @ assignable modelValue;
568   @ assignable modelCapacity;
569   @ assignable modelValue[offset .. modelCapacity - 1];
570   @ ensures modelStringLength == old(modelStringLength) +
trueChars.length;
571   @ ensures (\forall int i; offset <= i && i < offset +
trueChars.length; modelValue[i] == trueChars[i - offset]);
572   @ ensures (\forall int i; offset + trueChars.length <= i && i <
modelStringLength; modelValue[i] == old(modelValue)[i + trueChars.length]);
573   @ ensures \result == this;
574   @ also
575   @ public normal_behavior
576   @ requires b == false;
577   @ requires 0 <= offset && offset <= modelStringLength;
578   @ assignable modelValue;
579   @ assignable modelCapacity;
580   @ assignable modelValue[offset .. modelCapacity - 1];
581   @ ensures modelStringLength == old(modelStringLength) +
falseChars.length;
582   @ ensures (\forall int i; offset <= i && i < offset +
falseChars.length; modelValue[i] == falseChars[i - offset]);
583   @ ensures (\forall int i; offset + falseChars.length <= i && i <
modelStringLength; modelValue[i] == old(modelValue)[i + falseChars.length]);
584   @ ensures \result == this;
585   @ also
586   @ public exceptional_behavior
587   @ requires offset < 0 || offset > modelStringLength;
588   @ signals_only StringIndexOutOfBoundsException;
589   @*/
590 public AbstractStringBuilder insert(int offset, boolean b); //46
591
592 /*@ public normal_behavior
593   @ requires 0 <= offset && offset <= modelStringLength;
594   @ assignable modelValue;
595   @ assignable modelCapacity;
596   @ assignable modelValue[offset .. modelCapacity - 1];
597   @ ensures modelStringLength == old(modelStringLength) + 1;
598   @ ensures modelValue[offset] == c;
599   @ ensures (\forall int i; offset + 1 <= i && i < modelStringLength;
modelValue[i] == old(modelValue)[i + 1]);
600   @ ensures \result == this;
601   @ also
602   @ public exceptional_behavior
603   @ requires offset < 0 || offset > modelStringLength;
AbstractStringBuilder.jml

604     @   signals_only StringIndexOutOfBoundsException;
605     @*/
606     public AbstractStringBuilder insert(int offset, char c); //47
607
608     /*@ public normal_behavior
609     @   requires 0 <= offset && offset <= modelStringLength;
610     @   assignable modelValue;
611     @   assignable modelCapacity;
612     @   assignable modelValue[offset..modelCapacity - 1];
613     @   ensures modelStringLength == \old(modelStringLength) +
       String.valueOf(i).length();
614     @   ensures (\forall int j; offset <= j && j < offset +
       String.valueOf(i).length(); modelValue[j] == String.valueOf(i).charAt(j -
       offset));
615     @   ensures (\forall int j; offset + String.valueOf(i).length() <= i &&
       i < modelStringLength; modelValue[j] == \old(modelValue)[j + 1]);
616     @   ensures \result == this;
617     @ also
618     @ public exceptional_behavior
619     @   requires offset < 0 || offset > modelStringLength;
620     @   signals_only StringIndexOutOfBoundsException;
621     @*/
622     public AbstractStringBuilder insert(int offset, int i); //48
623
624     /*@ public normal_behavior
625     @   requires 0 <= offset && offset <= modelStringLength;
626     @   assignable modelValue;
627     @   assignable modelCapacity;
628     @   assignable modelValue[offset..modelCapacity - 1];
629     @   ensures modelStringLength == \old(modelStringLength) +
       String.valueOf(l).length();
630     @   ensures (\forall int i; offset <= i && i < offset +
       String.valueOf(l).length(); modelValue[i] == String.valueOf(l).charAt(i -
       offset));
631     @   ensures (\forall int i; offset + String.valueOf(l).length() <= i &&
       i < modelStringLength; modelValue[i] == \old(modelValue)[i + 1]);
632     @   ensures \result == this;
633     @ also
634     @ public exceptional_behavior
635     @   requires offset < 0 || offset > modelStringLength;
636     @   signals_only StringIndexOutOfBoundsException;
637     @*/
638     public AbstractStringBuilder insert(int offset, long l); //49
639
640     /*@ public normal_behavior
641     @   requires 0 <= offset && offset <= modelStringLength;
642     @   assignable modelValue;
AbstractStringBuilder.jml

643  @  assignable modelCapacity;
644  @  assignable modelValue[off .. modelCapacity - 1];
645  @  ensures modelStringLength == \old(modelStringLength) +
       String.valueOf(f).length();
646  @  ensures (\forall int i; offset <= i && i < offset +
       String.valueOf(f).length(); modelValue[i] == String.valueOf(f).charAt(i -
       offset));
647  @  ensures (\forall int i; offset + String.valueOf(f).length() <= i &&
       i < modelStringLength; modelValue[i] == \old(modelValue)[i + 1]);
648  @  ensures \result == this;
649  @ also
650  @ public exceptional_behavior
651  @   requires offset < 0 || offset > modelStringLength;
652  @   signals_only StringIndexOutOfBoundsException;
653  @*/
654  public AbstractStringBuilder insert(int offset, float f); //50

655  /* //public normal_behavior
656  @   requires 0 <= offset && offset <= modelStringLength;
657  @   assignable modelValue;
658  @   assignable modelCapacity;
659  @   assignable modelValue[off .. modelCapacity - 1];
660  @   ensures modelStringLength == \old(modelStringLength) +
       String.valueOf(d).length();
661  @   ensures (\forall int i; offset <= i && i < offset +
       String.valueOf(d).length(); modelValue[i] == String.valueOf(d).charAt(i -
       offset));
662  @   ensures (\forall int i; offset + String.valueOf(d).length() <= i &&
       i < modelStringLength; modelValue[i] == \old(modelValue)[i + 1]);
663  @   ensures \result == this;
664  @ also
665  @ public exceptional_behavior
666  @   requires offset < 0 || offset > modelStringLength;
667  @   signals_only StringIndexOutOfBoundsException;
668  @*/
669  public AbstractStringBuilder insert(int offset, double d); //51

670  /* //public normal_behavior
671  @   //requires (\exists int i; 0 <= i && i < modelStringLength -
       str.length(); //NOTE - Compile error when nesting exist/forall expressions.  
       Possible OpenJML bug.
672  @   // (\forall int j; i <= j && j < i + str.length(); modelValue[j]
       == str.charAt(j)));
673  @ */
674  //@ ensures -1 <= \result && \result < modelStringLength;
675  public /*@ pure @*/ int indexOf(String str); //52
676
AbstractStringBuilder.jml

679  //@ ensures -1 <= \result && \result < modelStringLength;
680  public /*@ pure @*/ int indexOf(String str, int fromIndex); //53
681
682  //@ ensures -1 <= \result && \result < modelStringLength;
683  public int lastIndexOf(String str); //54
684
685  //@ ensures -1 <= \result && \result < modelStringLength;
686  public int lastIndexOf(String str, int fromIndex); //55
687
688  //@ assignable modelValue[*];
689  //@ ensures \result == this;
690  public AbstractStringBuilder reverse(); //56
691
692  //@ ensures \result != null;
693  //@ ensures \result.length() == modelStringLength;
694  //@ ensures (\forall int i; 0 <= i && i < modelStringLength; modelValue[i] == \result.charAt(i));
695  public /*@ pure @*/ abstract String toString(); //58
696
697  //@ spec_public @*/ final /*@ pure @*/ char[] getValue(); //59
698}
** This program is free software: you can redistribute it and/or modify
** it under the terms of the GNU General Public License as published by
** the Free Software Foundation, either version 3 of the License, or
** (at your option) any later version.
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
** GNU General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program. If not, see <http://www.gnu.org/licenses/>.
**
package java.lang;

/**
 * JML's specification of StringBuilder.
 *
 * @author Mike Rawding
 */

class StringBuilder extends AbstractStringBuilder implements java.io.Serializable, CharSequence {

    /*@ public normal_behavior
    @   ensures modelCapacity == 16;
    @   ensures modelStringLength == 0;
    */
    public StringBuilder(); //1

    /*@ public normal_behavior
    @   requires capacity >= 0;
    @   ensures modelCapacity == capacity;
    @   ensures modelStringLength == 0;
    @ also
    @ public exceptional_behavior
    @   requires capacity < 0;
    @   signals_only NegativeArraySizeException;
    */
    public StringBuilder(int capacity); //2

    /*@ public normal_behavior
    @   ensures modelCapacity == str.length() + 16;
    @   ensures modelStringLength == str.length();
    @ also
    @ ensures (forall int i; 0 <= i && i < modelStringLength;
    modelValue[i] == str.charAt(i));
    */
public StringBuilder(String str); //3

/*@ public normal_behavior
   @   ensures modelCapacity == seq.length() + 16;
   @   ensures modelStringLength == seq.length();
   @   ensures (\forall int i; 0 <= i && i < modelStringLength;
       modelValue[i] == seq.charAt(i));
   @*/
public StringBuilder(CharSequence seq); //4

}
StringBuilderDriver.java

1/**
2 * This program is free software: you can redistribute it and/or modify
3 * it under the terms of the GNU General Public License as published by
4 * the Free Software Foundation, either version 3 of the License, or
5 * (at your option) any later version.
6 *
7 * This program is distributed in the hope that it will be useful,
8 * but WITHOUT ANY WARRANTY; without even the implied warranty of
9 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
10 * GNU General Public License for more details.
11 *
12 * You should have received a copy of the GNU General Public License
13 * along with this program. If not, see <http://www.gnu.org/licenses/>.
14 */
15import javax.swing.text.Segment;
16
17/**
18 * A driver program to exercise the functionality of
19 * java.lang.AbstractStringBuilder and
20 * java.lang.StringBuilder
21 *
22 * @author Mike Rawding
23 */
24public class StringBuilderDriver {
25
26   public static void main(String[] args) {
27      testConstructorVoid(); //1
28
29      testConstructorInt(); //2
30
31      testConstructorString(); //3 (StringBuilder)
32      testConstructorCharSequence(); //4 (StringBuilder)
33
34      testLength(); //3
35
36      testCapacity(); //4
37
38      testEnsureCapacity(); //5
39
40      testTrimToSize(); //9
41
42      testSetLength(); //10
43
44      testCharAt(); //11
45
46      testGetChars(); //16
47   }
StringBuilderDriver.java

        testSetCharAt(); //17
49        testAppendObject(); //18
50        testAppendString(); //19
51        testAppendStringBuffer(); //20
52        testAppendCharSequence(); //22
53        testAppendCharSequenceStartEnd(); //24
54        testAppendCharArray(); //25
55        testAppendCharArrayStartEnd(); //26
56        testAppend_boolean(); //27
57        testAppend_char(); //28
58        testAppend_int(); //29
59        testAppend_Long(); //30
60        testDelete(); //33
61        testDeleteCharAt(); //35
62        testReplace(); //36
63        testSubstringStart(); //37
64        testSubSequence(); //38
65        testSubStringStartEnd(); //39
66        testInsertIndexCharArray(); //40
67        testInsertObject(); //41
68        testInsertString(); //42
69        testInsertCharArray(); //43
70            System.out.println("done");
private static void testConstructorVoid() { //1
    StringBuilder sb = new StringBuilder();
    //@ assert sb.capacity() == 16;
    //@ assert sb.length() == 0;
}

private static void testConstructorInt() { //2
    testConstructorInt(0);
    testConstructorInt(1);
    try {
        testConstructorInt(-1);
    } catch (NegativeArraySizeException ex) {
        System.out.println("StringBuilder(-1) correctly threw an exception.");
    }
}

public static void testConstructorInt(int i) { //2
    StringBuilder sb = new StringBuilder(i);
    //@ assert sb.capacity() == i;
    //@ assert sb.length() == 0;
}

private static void testConstructorString() { //3
    testConstructorString("\n");
    testConstructorString("Test String");
    try {
        /*@ nullable @*/String nullString = null;
        //@ testConstructorString(nullString);//ESC failure since null not allowed
    } catch (NullPointerException ex) {
        System.out.println("StringBuilder(nullString) correctly threw an exception.");
    }
}

public static void testConstructorString(String str) { //3 (StringBuilder)
    StringBuilder sb = new StringBuilder(str);
    //@ assert sb.capacity() == str.length() + 16;
    //@ assert sb.length() == str.length();
}

private static void testConstructorCharSequence() { //4
    char[] charArray = {'t', 'e', 's', 't'};
}
testConstructorCharSequence(new Segment());

testConstructorCharSequence(new Segment(new char[] {}, 0, 0));

testConstructorCharSequence(new Segment(charArray, 0,
charArray.length));

try {
    /*@ nullable @*/Segment nullSegment = null;
    //testConstructorCharSequence(nullSegment);//ESC failure since
null not allowed
    } catch (NullPointerException ex) {}}

public static void testConstructorCharSequence(CharSequence cs) {
    StringBuilder sb = new StringBuilder(cs);
    //@ assert sb.capacity() == cs.length() + 16;
    //@ assert sb.length() == cs.length();
}

private static void testLength() { //3
    testLength(new StringBuilder("Test String"));
    testLength(new StringBuilder(0));
}

public static void testLength(StringBuilder sb) { //3
    sb.length();
}

private static void testCapacity() { //4
    testCapacity(new StringBuilder("Test String"));
    testCapacity(new StringBuilder(0));
}

public static void testCapacity(StringBuilder sb) { //4
    char [] originalValue = new char[sb.length()];
    sb.getChars(0, sb.length(), originalValue, 0);
    int originalLength = sb.length();
    sb.capacity();
}

private static void testEnsureCapacity() { //5
    testEnsureCapacity(new StringBuilder(0), 1);
    testEnsureCapacity(new StringBuilder(0), 0);
    testEnsureCapacity(new StringBuilder(0), -1);
    testEnsureCapacity(new StringBuilder("Test String"), 1);
    testEnsureCapacity(new StringBuilder(""), 1);
public static void testEnsureCapacity(StringBuilder sb, int minCapacity) {
    //5
    sb.ensureCapacity(minCapacity);
    // assert sb.capacity() >= minCapacity;
}

private static void testTrimToSize() { //9
    testTrimToSize(new StringBuilder());
    testTrimToSize(new StringBuilder("Test String"));
    testTrimToSize(new StringBuilder(100));
}

public static void testTrimToSize(StringBuilder sb) { //9
    sb.trimToSize();
}

private static void testSetLength() { //10
    testSetLength(new StringBuilder("Test String"), 100);
    testSetLength(new StringBuilder(1000), 100);
    testSetLength(new StringBuilder("Test String"), 1);
    testSetLength(new StringBuilder("Test String"), 0);

    try {
        testSetLength(new StringBuilder("Test String"), -1);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("setLength(-1) correctly threw an exception.");
    }
}

private static void testSetLength(StringBuilder sb, int length) { //10
    sb.setLength(length);
    //@ assert sb.length() == length;
}

private static void testCharAt() { //11
    testCharAt(new StringBuilder("Test String"), 0);
    testCharAt(new StringBuilder("Test String"), 1);

    try {
        testCharAt(new StringBuilder(""), 0);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("charAt(0) on empty StringBuilder correctly threw an exception.");
    }
}
StringBuilderDriver.java

```
229     try {
230         testCharAt(new StringBuilder("Test String"), -1);
231     } catch (StringIndexOutOfBoundsException ex) {
232         System.out.println("charAt(-1) correctly threw an exception.");
233     }
234
235     try {
236         testCharAt(new StringBuilder("Test String"), 100);
237     } catch (StringIndexOutOfBoundsException ex) {
238         System.out.println("charAt(100) (too large) correctly threw an exception.");
239     }
240 }
241
242 public static void testCharAt(StringBuilder sb, int index) {
243     sb.charAt(index);
244     //assert sb.charAt(index) == sb.value[index];
245     //undefined index too large
246 }
247
248 private static void testGetChars() {
249     testGetChars(new StringBuilder("Test String"), 0, 5, new char[10], 0);
250     try {
251         testGetChars(new StringBuilder("Test String"), 0, 5, new char[10], 0);
252     } catch (ArrayIndexOutOfBoundsException ex) {
253         System.out.println("getChars() correctly threw an exception.");
254     }
255 }
256
257 public static void testGetChars(StringBuilder sb, int srcBegin, int srcEnd, char[] dst, int dstBegin) {
258     sb.getChars(srcBegin, srcEnd, dst, dstBegin);
259     //RAC-ESC@ assert (\forall int i; srcBegin <= i && i < srcEnd;
260     sb.value[i] == dst[i + dstBegin]);
261     //invalid assertion
262 }
263
264 private static void testSetCharAt() {
265     testSetCharAt(new StringBuilder("Test String"), 2, 'a');
266     try {
267         testSetCharAt(new StringBuilder("Test String"), 100, 'a');
268     } catch (StringIndexOutOfBoundsException ex) {
269         System.out.println("setChars() correctly threw an exception.");
270     }
271```
try {
    testSetCharAt(new StringBuilder("Test String"), -1, 'a');
} catch (StringIndexOutOfBoundsException ex) {
    System.out.println("setChars() correctly threw an exception.");
}

public static void testSetCharAt(StringBuilder sb, int index, char c) {
    char [] originalValue = new char[sb.length()];
    sb.getChars(0, sb.length(), originalValue, 0);
    sb.setCharAt(index, c);
}

private static void testAppendObject() {
    Integer testInteger = new Integer(1);
    /*@ nullable @*/Integer nullInteger = null;
    testAppendObject(new StringBuilder("Test String"), testInteger);
    testAppendObject(new StringBuilder("Test String"), nullInteger);
}

public static void testAppendObject(StringBuilder sb, /*@ nullable @*/Object o) {
    sb.append(o);
}

private static void testAppendString() {
    testAppendString(new StringBuilder("Test String"), "test");  
    testAppendString(new StringBuilder("Test String"), "");
    /*@ nullable @*/ String nullString = null;
    testAppendString(new StringBuilder("Test String"), nullString);
}

public static void testAppendString(StringBuilder sb, /*@ nullable @*/String string) {
    sb.append(string);
}

private static void testAppendStringBuffer() {
    /*@ nullable @*/StringBuffer nullStringBuffer = null;
}
StringBuilderDriver.java

    public static void testAppendStringBuffer(StringBuilder sb, StringBuffer stringBuffer) {
        sb.append(stringBuffer);
    }

    private static void testAppendCharSequence() {
        char[] testCharArray = {'t', 'e', 's', 't'};
        Segment testSequence = new Segment(testCharArray, 0, testCharArray.length);
        char[] emptyCharArray = {};
        Segment emptySegment = new Segment(emptyCharArray, 0, emptyCharArray.length);
        //@ nullable @*/ Segment nullSegment = null;
        StringBuilder stringBuilder = new StringBuilder(); 
        stringBuilder.append(testSequence);
        //stringBuilder.append(emptySegment);
        stringBuilder.append(nullSegment);
    }

    public static void testAppendCharSequence(StringBuilder sb, /*@ nullable @*/ CharSequence cs) {
        sb.append(cs);
    }

    private static void testAppendCharSequenceStartEnd() {
        char[] testCharArray = {'t', 'e', 's', 't'};
        char[] emptyCharArray = {};
        //@ nullable @*/ Segment nullSegment = null;
        StringBuilder sb = new StringBuilder();
        testAppendCharSequenceStartEnd(sb, new Segment(testCharArray, 0, testCharArray.length), 0, 4);
        testAppendCharSequenceStartEnd(sb, new Segment(testCharArray, 0, testCharArray.length), 1, 1);
        testAppendCharSequenceStartEnd(sb, new Segment(emptyCharArray, 0, emptyCharArray.length), 0, 0);
        testAppendCharSequenceStartEnd(sb, nullSegment, 0, 0);
        try {
            testAppendCharSequenceStartEnd(sb, new Segment(testCharArray, 0, testCharArray.length), 0, 4 + 1);
catch (IndexOutOfBoundsException ex) {
    System.out.println("append(charSequence, int, int) correctly threw an exception. end too large.");
}
    try {
        testAppendCharSequenceStartEnd(sb, new Segment(testCharArray, 0, testCharArray.length), -1, 4);
    } catch (IndexOutOfBoundsException ex) {
        System.out.println("setChars() correctly threw an exception. negative start.");
    }
    try {
        testAppendCharSequenceStartEnd(sb, new Segment(testCharArray, 0, testCharArray.length), 3, 1);
    } catch (IndexOutOfBoundsException ex) {
        System.out.println("setChars() correctly threw an exception. start > end");
    }
    public static void testAppendCharSequenceStartEnd(StringBuilder sb, /*@ nullable */ CharSequence cs, int start, int end) { //24
        sb.append(sb.append(cs, start, end));
    }
    private static void testAppendCharArray() { //25
        char[] testCharArray = {'t', 'e', 's', 't'};
        char[] emptyCharArray = {};
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append(testCharArray);
        stringBuilder.append(emptyCharArray);
    }
    private static void testAppendCharArrayStartEnd() { //26
        char[] testCharArray = {'t', 'e', 's', 't'};
        char[] emptyCharArray = {};
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append(testCharArray, 0, testCharArray.length);
        stringBuilder.append(emptyCharArray, 0, emptyCharArray.length);
private static void testAppend_boolean() { //27
    testAppend_boolean(new StringBuilder(), true);
    testAppend_boolean(new StringBuilder(), false);
    testAppend_boolean(new StringBuilder("Test String"), true);
    testAppend_boolean(new StringBuilder("Test String"), false);
}

public static void testAppend_boolean(StringBuilder sb, boolean b) { //27
    sb.append(b);
}

private static void testAppend_char() { //28
    StringBuilder sb = new StringBuilder();
    testAppend_char(sb, 'a');
    testAppend_char(sb, '1');
    testAppend_char(sb, '\0');
}

public static void testAppend_char(StringBuilder sb, char c) { //28
    sb.append(c);
}

private static void testAppend_int() { //29
    StringBuilder sb = new StringBuilder();
    testAppend_int(sb, -1);
    testAppend_int(sb, 0);
    testAppend_int(sb, 1);
}

public static void testAppend_int(StringBuilder sb, int i) { //29
    sb.append(i);
}

private static void testAppend_long() { //30
    StringBuilder sb = new StringBuilder();
    testAppend_long(sb, 1L);
    testAppend_long(sb, 0L);
    testAppend_long(sb, -1L);
}

public static void testAppend_long(StringBuilder sb, long l) { //30
    sb.append(l);
}
StringBuilderDriver.java

443
444 //    //NOTE - reals are not supported for ESC
445 //    private static void testAppend_float() { //31
446 //        StringBuilder sb = new StringBuilder();
447 //
448 //        testAppend_float(sb, -1.0f);
449 //        testAppend_float(sb, 0f);
450 //        testAppend_float(sb, -1.5f);
451 //    }
452 //    public static void testAppend_float(StringBuilder sb, float f) { //31
453 //        sb.append(f);
454 //    }
455 //
456 //     //NOTE - reals are not supported for ESC
457 //    private static void testAppend_double() { //32
458 //        StringBuilder sb = new StringBuilder();
459 //
460 //    testAppend_double(sb, 1.0);
461 //    testAppend_double(sb, 0);
462 //    testAppend_double(sb, -1.5);
463 //    }
464 //    public static void testAppend_double(StringBuilder sb, double d) { //32
465 //        sb.append(d);
466 //    }
467 //
468 private static void testDelete() { //33
469 
470        testDelete(new StringBuilder("Test String"), 0, 0);
471        testDelete(new StringBuilder("Test String"), 0, 1);
472        testDelete(new StringBuilder("Test String"), 0, 100);
473
474        try {
475            testDelete(new StringBuilder("Test String"), -1, 1);
476        } catch (StringIndexOutOfBoundsException ex) {
477            System.out.println("delete() with negative start correctly threw an exception");
478        }
479
480        try {
481            testDelete(new StringBuilder("Test String"), 2, 1);
482        } catch (StringIndexOutOfBoundsException ex) {
483            System.out.println("delete() with start > end correctly threw an exception");
484        }
StringBuilderDriver.java

```java
public static void testDelete(StringBuilder sb, int start, int end) {
    sb.delete(start, end);
}

private static void testDeleteCharArray() { //35
    testDeleteCharArray(new StringBuilder("Test String"), 0);
    testDeleteCharArray(new StringBuilder("Test String"), 1);

    try {
        testDeleteCharArray(new StringBuilder("Test String"), -1);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("delete() start > length correctly threw an exception");
    }
}

public static void testDeleteCharAt(StringBuilder sb, int index) { //35
    sb.deleteCharAt(index);
}

private static void testReplace() { //36
    testReplace(new StringBuilder("Test String"), 0, 1, "replace");
    testReplace(new StringBuilder("Test String"), 0, 0, "replace");
    testReplace(new StringBuilder("Test String"), 1, 5, "replace");

    try {
        testReplace(new StringBuilder("Test String"), -1, 1, "replace");
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("replace() with negative start correctly threw an exception");
    }
```
StringBuilderDriver.java

}  

try {
    testReplace(new StringBuilder("Test String"), 1, 0, "replace");
} catch (StringIndexOutOfBoundsException ex) {
    System.out.println("replace() with start > end correctly threw an exception");
}

try {
    testReplace(new StringBuilder("Test String"), 100, 101, "replace");
} catch (StringIndexOutOfBoundsException ex) {
    System.out.println("replace() with start > count correctly threw an exception");
}

public static void testReplace(StringBuilder sb, int start, int end, String str) {
    sb.replace(start, end, str);
}

private static void testSubstringStart() {
    testSubstringStart(new StringBuilder("Test String"), 0);
    testSubstringStart(new StringBuilder("Test String"), 1);
    testSubstringStart(new StringBuilder("Test String"), 11);
    try {
        testSubstringStart(new StringBuilder("Test String"), -1);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("substring() with negative start correctly threw an exception");
    }
    try {
        testSubstringStart(new StringBuilder("Test String"), 100);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("substring() with start too large correctly threw an exception");
    }
}

private static void testSubstringStart(StringBuilder sb, int start) {
    sb.substring(start);
}
private static void testSubSequence() { //38
    testSubSequence(new StringBuilder("Test String"), 0, 1);
    testSubSequence(new StringBuilder("Test String"), 1, 1);
    testSubSequence(new StringBuilder("Test String"), 0, 11);
    try {
        testSubSequence(new StringBuilder("Test String"), -1, 1);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("subSequence() with negative start correctly threw an exception");
    }
    try {
        testSubSequence(new StringBuilder("Test String"), 1, 0);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("subSequence() with start > end correctly threw an exception");
    }
    try {
        testSubSequence(new StringBuilder("Test String"), 0, 100);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("subSequence() with end too large correctly threw an exception");
    }
}

public static void testSubSequence(StringBuilder sb, int start, int end) {
    sb.subSequence(start, end);
}

private static void testSubStringLengthStartEnd() { //38
    testSubStringLengthStartEnd(new StringBuilder("Test String"), 0, 1);
    testSubStringLengthStartEnd(new StringBuilder("Test String"), 1, 1);
    testSubStringLengthStartEnd(new StringBuilder("Test String"), 0, 11);
    try {
        testSubStringLengthStartEnd(new StringBuilder("Test String"), -1, 1);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("subSequence() with negative start correctly threw an exception");
    }
    try {
        testSubStringLengthStartEnd(new StringBuilder("Test String"), 1, 0);
    }
StringBuilderDriver.java

```java
615        } catch (StringIndexOutOfBoundsException ex) {
616            System.out.println("subSequence() with start > end correctly threw an exception");
617        }
618
619        try {
620            testSubStringStartEnd(new StringBuilder("Test String"), 0, 100);
621        } catch (StringIndexOutOfBoundsException ex) {
622            System.out.println("subSequence() with end too large correctly threw an exception");
623        }
624    }
625
626    public static void testSubStringStartEnd(StringBuilder sb, int start, int end) {
627        sb.substring(start, end);
628    }
629
630    private static void testInsertIndexCharArray() {
631        testInsertIndexCharArray(new StringBuilder("Test String"), 0, new char[] {'t', 'e', 's', 't'}, 0, 3);
632        testInsertIndexCharArray(new StringBuilder("Test String"), 0, new char[] {'t', 'e', 's', 't'}, 0, 0);
633
634        try {
635            testInsertIndexCharArray(new StringBuilder("Test String"), -1, new char[] {'t', 'e', 's', 't'}, 0, 0);
636        } catch (StringIndexOutOfBoundsException ex) {
637            System.out.println("insert() with negative index correctly threw an exception");
638        }
639
640        try {
641            testInsertIndexCharArray(new StringBuilder("Test String"), 100, new char[] {'t', 'e', 's', 't'}, 0, 0);
642        } catch (StringIndexOutOfBoundsException ex) {
643            System.out.println("insert() with index > stringLength correctly threw an exception");
644        }
645
646        try {
647            testInsertIndexCharArray(new StringBuilder("Test String"), 0, new char[] {'t', 'e', 's', 't'}, -1, 0);
648        } catch (StringIndexOutOfBoundsException ex) {
649            System.out.println("insert() with negative offset correctly threw an exception");
```
```java
StringBuilderDriver.java

    try {
        testInsertIndexCharArray(new StringBuilder("Test String"), 0, new char[] {'t', 'e', 's', 't'}, 0, -1);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("insert() with negative len correctly threw an exception");
    }

    try {
        testInsertIndexCharArray(new StringBuilder("Test String"), 0, new char[] {'t', 'e', 's', 't'}, 0, 100);
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("insert() with len too large correctly threw an exception");
    }

    public static void testInsertIndexCharArray(StringBuilder sb, int index, char[] charArray, int offset, int len) {
        sb.insert(index, charArray, offset, len);
    }

    private static void testInsertObject() {
        testInsertObject(new StringBuilder("Test String"), 0, new Integer(1));
        testInsertObject(new StringBuilder("Test String"), 11, new Integer(1));

        /*@ nullable @*/ Object nullObject = null;
        testInsertObject(new StringBuilder("Test String"), 0, nullObject);

        try {
            testInsertObject(new StringBuilder("Test String"), -1, new Integer(1));
        } catch (StringIndexOutOfBoundsException ex) {
            System.out.println("insert() with negative offset correctly threw an exception");
        }

        try {
            testInsertObject(new StringBuilder("Test String"), 100, new Integer(1));
        } catch (StringIndexOutOfBoundsException ex) {
            System.out.println("insert() with offset too large correctly threw an exception");
        }
```
public static void testInsertObject(StringBuilder sb, int offset, /*@ nullable */ Object o) {
    sb.insert(offset, o);
}

private static void testInsertString() {
    testInsertString(new StringBuilder("Test String"), 0, "insert");
    testInsertString(new StringBuilder("Test String"), 11, "insert");
    String nullString = null;
    testInsertString(new StringBuilder("Test String"), 0, nullString);
    try {
        testInsertString(new StringBuilder("Test String"), -1, "insert");
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("insert() with negative offset correctly threw an exception");
    }
    try {
        testInsertString(new StringBuilder("Test String"), 100, "insert");
    } catch (StringIndexOutOfBoundsException ex) {
        System.out.println("insert() with offset too large correctly threw an exception");
    }
}

public static void testInsertString(StringBuilder sb, int offset, /*@ nullable */ String str) {
    sb.insert(offset, str);
}

private static void testInsertCharArray() {
    testInsertCharArray(new StringBuilder("Test String"), 0, new char[] {'t', 'e', 's', 't'});
    testInsertCharArray(new StringBuilder("Test String"), 11, new char[] {'t', 'e', 's', 't'});
    try {
        testInsertCharArray(new StringBuilder("Test String"), -1, new char[] {'t', 'e', 's', 't'});
    }
}
catch (StringIndexOutOfBoundsException ex) {
    System.out.println("insert() with negative offset correctly threw an exception");
}

try {
    testInsertCharArray(new StringBuilder("Test String"), 100, new char[] {'t', 'e', 's', 't'});
} catch (StringIndexOutOfBoundsException ex) {
    System.out.println("insert() with offset too large correctly threw an exception");
}

public static void testInsertCharArray(StringBuilder sb, int offset, char[] charArray) {
    sb.insert(offset, charArray);
}