

This document contains a set of homework exercises for those that wish to learn JML and ESC/Java2. It was begun by Joseph R. Kiniry <kiniry@acm.org> in May 2004 for the tutorial “Design by Contract and Automatic Verification for Java with JML and ESC/Java2” presented at ECOOP 2004 in Oslo, Norway in June 2004. This is edition \$Revision: 1.13 \$.

This document is still very much a work in progress. Suggestions and input are welcome.

JML and ESC/Java2 Homework Exercises

Learning JML and ESC/Java2 Through Example

Edition \$Revision: 1.13 \$, May 2004

These exercises use at least JML version 5.0rc1 and ESC/Java2 version 2.0a7.

Joseph R. Kiniry <kiniry@acm.org>

The JML and ESC/Java2 Homework Exercises are licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.0 license. See <http://creativecommons.org/licenses/by-nc-sa/2.0/>

Copyright © 2004 Joseph R. Kiniry

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works

Under the following conditions:

- Attribution. You must give the original author credit.
- Noncommercial. You may not use this work for commercial purposes.
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

For any reuse or distribution, you must make clear to others the license terms of this work.

Any of these conditions can be waived if you get permission from the author.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code.

See <http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>

Table of Contents

Introduction	2
Core JML Constructs	3
Assertions	3
Assumptions	7
Ghost Variables	11
Lightweight Contracts	11
Preconditions	11
Postconditions	13
Exceptional (Abnormal) Postconditions	13
Frame Axioms	14
Data Groups	16
Heavyweight Specifications	17
Invariants	17
Reasoning about Loops	17
Aliasing	17
Models	17
Using JML and ESC/Java2	18
Specifying Abstract Classes and Interfaces	18
Annotating Preexisting Java Code	18
Annotating Preexisting APIs	18
Designing by Contract	18
Copying	19
Index	20

This set of homework problems focus on the use of JML in runtime assertion checking (with `jmlc` and `jmlrac`) and static verification with ESC/Java2.

Introduction

The Java Modeling Language (JML) is a behavioral interface specification language that can be used to specify the behavior of Java modules. It combines the “Design by Contract” approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages with some elements of the refinement calculus.

The JML tool suite comes with a typechecker (`jml`), a compiler (`jmlc`), a runtime assertion checker (`jmlrac`), a Javadoc-based documentation generator (`jmldoc`), a set of unit test tools (`jml-junit`, `jmlunit`, `jtest`), and a specification skeleton generator and specification comparison tool (`jmlspec`).

The JML tool suite is primarily the development of the JML project at Iowa State University, led by Gary Leavens. <http://www.jmlspecs.org/>

Several other tools understand (often subsets of) JML:

- The Extended Static Checker for Java (ESC/Java version 1). <http://research.compaq.com/SRC/esc/>
- The Open Source Extended Static Checker for Java (ESC/Java version 2, or ESC/Java2 for short) from the SoS Group at the University of Nijmegen and David Cok. <http://www.cs.kun.nl/sos/research/escjava/>
- The LOOP compiler from the SoS Group at the University of Nijmegen. <http://www.cs.kun.nl/sos/>
- The Daikon invariant detector from the Program Analysis Group at MIT. <http://pag.lcs.mit.edu/daikon/>
- The ChAsE frame axiom checker from Lemme project at INRIA Sophia-Antipolis. <http://www-sop.inria.fr/lemme/verificard/modifSpec/index.html>
- The JACK tool from Gemplus. http://www.gemplus.com/smart/r_d/trends/jack.html
- The Bogor tool from the SpEx project at Kansas State University. <http://spex.projects.cis.ksu.edu/>

This set of homework problems focus on the use of JML in runtime assertion checking (with `jmlc` and `jmlrac`) and static verification with ESC/Java2.

Core JML Constructs

The following sections discuss all of the core constructs of JML with examples and exercises. It is suggested that each exercise be completed in sequence, as the exercises build upon each other in the order presented. *I.e.*, [Problem two], page 5 builds on the knowledge that you gained in [Problem one], page 4.

It is assumed that the reader has a basic understanding of the following notions: *assertion*, *invariant*, *specification*, *signature*, *class*, and *type*.

It is also assumed that the reader has a good understanding of Java. If you want write a specification of a method written in Java, *you have to understand what the method does and what it is meant to do*. You cannot expect to write specifications for programs written in a language you do not understand.

When working on these problems, do not attempt to obtain full verification completeness at the cost of poor specifications. Understanding the limitations of ESC/Java2 and other JML-based tools is part of the challenge of writing specifications and performing verifications today.

In particular, several classes or methods are reused through several problems. As new constructs are introduced, and as your specifications become more rich (and thus, complete), more of your code will verify. You should not expect all of your answers to fully verify until you have covered all of the concepts discussed in this document.

Assertions

An assertion is a predicate that *must always* be true. Assertions are typically used for debugging and (localized) documentation purposes.

In JML, an assertion statement has the form

```
assert-statement ::= assert expression [ : expression ] ;
```

(Note that all specifications of JML grammar in this document are highly simplified from the actual JML grammar. Only the basic forms of each statement or clause are shown and are sufficient for the purpose of this homework.)

The first expression is the predicate that must be true. The second (optional) expression must be of type **String**. The value of this expression is printed by some tools when the assertion fails. (add xrefs)

Beginning in Java 1.4, an **assert** keyword was introduced. The syntax of the Java assert statement is the same as that of JML. (add xref)

Java's semantics does not restrict either subexpression in its assert statement, so you can write assert statements that have side-effects such as

```
int x;
x++;
assert (x++ == 2) : new String("This is a bad idea.");
```

This is a very bad idea, as it means that the behavior of your program changes whether you are testing your assertions or not.

JML's assert statement does not permit side effects in either subexpression.

Using runtime checking (with `jmlc` and `jmlrac`), JML and Java assert statements are tested at runtime. Using ESC/Java, JML assert statements are proven at compile-time.

In ESC/Java2, Java assert statements are treated one of two ways. They are either

- Treated as Java assert statements using the `-eajava` or `-javaAssertions` (synonymous) switches, or
- Treated as JML assert statements using the `-eajml` or `-jmlAssertions` (synonymous) switches.

In the former case, a failed assertion has the semantics of

```
throw new AssertionError(second expression);
```

Thus, you are **not** reasoning about another potential runtime exception, as all subtypes of `Error` are ignored by ESC/Java. (add xref)

Problem 1. Annotate the following code with Java assertions as indicated. The expression `P(a0,a1,...)` means “replace this expression with a predicate that is true of the variables `a0` and `a1` and ...”. Write the strongest predicate possible, given the context of the assertion. Execute the class to check your answers.

The provided Makefile in the `problems` directory can be used to compile, run, and verify all problems.

```
package problems;

class JavaAssertions extends Predicates
{
    static int i;
    static Object o;
    static String s;

    static void m() {
        i++;
        s = new String("foo");
        o = new Integer(1);
    }

    static void n(int j, Object p, String t) {
        assert O(p);
        if (p.equals(t))
            return;
        i |= 0x100;
        j |= 0xEFF;
        assert J(i, j);
        t += "bar";
    }

    static Object o(int i, Object o, String s) {
        assert O(o);
        assert S(s);
        if (s.equals(o))
            return new Integer(i);
        i ^= 0xFF;
        assert I(i);
        JavaAssertions.s += "piggie";
        return s;
    }

    static Object p(int i, Object o, String s) {
        assert O(o);
        assert S(s);
        switch (i) {
            case -255:
                s = "duck";
                return o;
        }
    }
}
```

```

        case -1:
            o = s;
            return new Short((short)i++);
        case 0:
            s = (String)o;
            return new Byte((byte)i--);
        case 1:
            i *= -2;
            return (String)o + s;
        case 255:
            i = o.hashCode();
            return s;
        case 257:
            s = ((Integer)o).toString();
            return s;
        default:
            return null;
    }
}

public static void main(String[] args) {
    assert I(i);
    assert S(s);
    assert O(o);
    m();
    assert I(i);
    assert S(s);
    assert O(o);
    n(i, o, s);
    assert I(i);
    assert S(s);
    assert O(o);
    Object p = p(-1 & i, o, (String)o(i, o, s));
    assert O(p);
    Object q = p(-1, o, (String)p);
    assert I(i);
    assert S(s);
    assert O(o);
    assert O(q);
}
}

```

Problem 2. Annotate the following code with JML assertions as indicated. Write the strongest predicate possible, given the context of the assertion. After you have written the assertions, first execute the class using `jmlrac` to check your answers, then attempt to verify your answers using `escjava2`.

```

package problems;

import problems.Predicates;

class JmlAssertions extends Predicates
{
    static int i;

```



```

static Object o;
static String s;

static void m() {
    i++;
    s = new String("foo");
    o = new Integer(1);
}

static void n(int j, Object p, String t) {
    //@ assert 0(p);
    if (p.equals(t))
        return;
    i |= 0x100;
    j |= 0xEFF;
    //@ assert J(i, j);
    t += "bar";
}

static Object o(int i, Object o, String s) {
    //@ assert 0(o);
    //@ assert S(s);
    if (s.equals(o))
        return new Integer(i);
    i ^= 0xFF;
    //@ assert I(i);
    JmlAssertions.s += "piggie";
    return s;
}

static Object p(int i, Object o, String s) {
    //@ assert 0(o);
    //@ assert S(s);
    switch (i) {
        case -255:
            s = "duck";
            return o;
        case -1:
            o = s;
            return new Short((short)i++);
        case 0:
            s = (String)o;
            return new Byte((byte)i--);
        case 1:
            i *= -2;
            return (String)o + s;
        case 255:
            i = o.hashCode();
            return s;
        case 257:
            s = ((Integer)o).toString();
            return s;
    }
}

```

```

        default:
            return null;
    }
}

public static void main(String[] args) {
    //@ assert I(i);
    //@ assert S(s);
    //@ assert O(o);
    m();
    //@ assert I(i);
    //@ assert S(s);
    //@ assert O(o);
    n(i, o, s);
    //@ assert I(i);
    //@ assert S(s);
    //@ assert O(o);
    Object p = p(-1 & i, o, (String)o(i, o, s));
    //@ assert O(p);
    Object q = p(-1, o, (String)p);
    //@ assert I(i);
    //@ assert S(s);
    //@ assert O(o);
    //@ assert O(q);
}
}

```

Assumptions

You will notice that nearly all of the assertions that you wrote for Problems [one], page 4 and [two], page 5 cannot be checked with ESC/Java2 as the `JavaAnnotations` and `JmlAnnotations` classes are currently written. While *you* might know that each assertion is true, given the context of the program and the body of the `main()` method, there is no information that ESC/Java2 can use to reach this same conclusion.

Because ESC/Java2 performs verification in a *modular* fashion, the *assumptions* you based your assertions upon in each method, which in turn were based upon what you read in the `main()` method's body, are unknown to ESC/Java2.

More precisely, to reason about the correctness of the method `n(int, Object, String)`, ESC/Java2 only uses the signature and body of `n()` and everything they depends upon. This set of dependencies is large. It including all the types mentioned in `n()`, the signature and specification of any methods `n()` might be overriding, and all invariants of the class `JmlAssertions` that relate to fields mentioned in `n()`. (More details about method specifications and class invariants will be covered shortly.)

Notice that, to reason about `n()`, *nothing* needs to be known about the body of the `main()` method of `JmlAssertions`. Thus, all of the assumptions you made by reading the body of `main()` are unknown to ESC/Java2.

The next two problems will focus on making assumptions explicit. The JML keyword `assume` is used to specify assumptions precisely. `assume` is used very much like `assert`, except instead of stating what *must* be true at a certain point in a program (as one does with an `assert`), an `assume` statement says what *is* true at a certain point in a program.

In JML, an `assume` statement has the form

$$\text{assume-statement} ::= \text{assume-keyword } \text{predicate} \text{ [: expression] ;}$$

The assume statement thus has the exact same structure of the assert statement, but means exactly the opposite.

Problem 3. Annotate your code from [Problem two], page 5 with JML assumptions. Try to write down the *minimal* set of assumptions that ensure all of your assertions verify correctly with ESC/Java2.

For the following discussion we will refer to the answer key found in the class `JmlAssertionsAndAssumptionsKey`, duplicated here for reference.

```
package problems;

import problems.Predicates;

class JmlAssertionsAndAssumptionsKey extends Predicates
{
    static int i;
    static Object o;
    static String s;

    //@ modifies i, s, o;
    static void m() {
        i++;
        s = new String("foo");
        o = new Integer(1);
    }

    //@ modifies i;
    static void n(int j, Object p, String t) {
        //@ assume p != null;
        //@ assert p != null;
        if (p.equals(t))
            return;
        //@ assume i == 1;
        //@ assume j == 1;
        i |= 0x100;
        j |= 0xFF;
        //@ assert (i & j) == 1;
        t += "bar";
    }

    //@ modifies JmlAssertionsAndAssumptionsKey.s;
    static Object o(int i, Object o, String s) {
        //@ assume o instanceof Integer;
        //@ assume ((Integer)o).intValue() == 1;
        //@ assume s != null;

        //@ assert ((Integer)o).intValue() == 1;
        //@ assert s != null;
        if (s.equals(o))
            return new Integer(i);
        //@ assume i == 257;
        i ^= 0xFF;
        //@ assert i == 510;
        JmlAssertionsAndAssumptionsKey.s += "piggie";
    }
}
```

```

    return s;
}

static Object p(int i, Object o, String s) {
    //@ assume o instanceof Integer;
    //@ assume s != null;

    //@ assert o instanceof Integer;
    //@ assert s != null;
    switch (i) {
        case -255:
            s = "duck";
            return o;
        case -1:
            o = s;
            return new Short((short)i++);
        case 0:
            s = (String)o;
            return new Byte((byte)i--);
        case 1:
            i *= -2;
            return (String)o + s;
        case 255:
            i = o.hashCode();
            return s;
        case 257:
            s = ((Integer)o).toString();
            return s;
        default:
            return null;
    }
}

public static void main(String[] args) {
    //@ assume i == 0;
    // assume s == null;
    // assume o == null;
    // assert i == 0;
    // assert s == null;
    // assert o == null;
    m();
    //@ assume i == 1;
    //@ assert i == 1;
    //@ assert s.equals("foo");
    //@ assert ((Integer)o).intValue() == 1;
    n(i, o, s);
    //@ assert i == 256 + 1;
    //@ assert s.equals("foo");
    //@ assert new Integer(1).equals(o);
    Object p = p(-1 & i, o, (String)o(i, o, s));
    //@ assert p.equals("1");
    Object q = p(-1, o, (String)p);
}

```

```

    //@ assert i == 257;
    //@ assert s.equals("foopiggie");
    //@ assert ((Integer)o).intValue() == 1;
    //@ assert ((Short)q).shortValue() == -1;
}
}

```

Note that the assumptions in this class are not the only ones that are correct.

Stronger assumptions (that is, logically stronger predicates) that imply these weaker assumptions are correct also, though are not as weak as possible.

For example, if one were to write

```
//@ assume p != null && t != null;
```

to provide an assumption to fulfill the assertion

```
//@ assert p != null;
```

the assertion would verify. But the assumption is stronger than necessary, since the information about the variable `t` is superfluous. The assumption can be made *weaker* by simply removing the term `t != null` because $A \ \&\& \ B \implies A$. I.e.,

```
//@ assert p != null && t != null ==> p != null;
```

is always a true assertion, regardless of the values of `p` and `t`.

The locations of the assumptions can also vary to some degree. Many cannot, in particular, those having to do with assertions located at the very beginning of method bodies cannot be moved any “earlier” because there is no earlier statement. Moving the assumption *outside* of the method body, for example into the `main()` method, has no effect due to the nature of modular verification, as discussed earlier. (add xref)

On the other hand, assumptions in the middle of a method can often be moved around with some latitude. For example, the statement

```
//@ assume i == 257;
```

in the method `o(int, Object, String)` could be moved earlier in the method. In fact, because the variable `i` is not referenced anywhere earlier in the method, the assumption could (in theory) be moved all of the way to the beginning of the method.

One problem with ESC/Java2 highlighted by this exercise is its incomplete support for arithmetic, and bitwise operations in particular. You will find that nearly all assertions having to do with the results of bitwise operations cannot be verified.

Consider the following method:

```

static void nm(Object p, Object t) {
    int i = 1, j = 1;
    i = i | 0x100;
    //@ assert i == 257;
    //@ assert 0 <= i && i <= 1000;
    //@ assert i == (1 | 0x100);
}

```

While all three assertions are true, only the third one (the most complicated one, in some respects!) can be verified by ESC/Java2. This weakness is due to the fact that the theorem prover used by ESC/Java2 (called “Simplify”) does not deal with bitwise operations (see Bug #965748 at SourceForge for more information). Thus, one cannot reason about any variables involved in bitwise operations at this time.

Assume statements are not to be used lightly. In fact, as a general rule, one should use assumes only as an absolute last resort. It is entirely too easy to introduce subtle inconsistencies to specifications with assume statements. If such an inconsistency exists, then the related methods will verify immediately because they have a false premise.

E.g., this method verifies:

```
void m() {
    //@ assume i == 0 && i == 1;
    //@ assert true == false;
}
```

Ghost Variables

Lightweight Contracts

A *specification* of the class and its methods describes the manner in which the class is correctly used. To write such a specification, one uses two primary concepts: *method contracts* and *class invariants*. The problems of this section will focus on the former; invariants will be dealt with later. (add xref).

A *method contract* consists of two basic assertions: a *precondition* which states under which circumstances the method is guaranteed to behave correctly, and a *postcondition* which states what is true when the method completes executing.

Preconditions

The assume statements written for the preceding problems that were “pushed” to the beginning of the method are preconditions; they state some of the conditions necessary for their method to terminate properly.

Preconditions are specified in JML using the **requires** clause which has the following form:

```
requires-clause ::= requires pred-or-not ;
pred-or-not ::= predicate | \not_specified
```

The value `\not_specified` says that you have a precondition in mind but you simply are not specifying it. Different tools can view such an unspecified precondition in different ways. For example, ESC/Java2 assumes a default precondition of **true**.

The process of writing preconditions is a tricky business. Preconditions must be ensured by the *caller* of the method, therefore everything stated in the precondition must be checkable by the caller, prior to making the call. Thus, one can only (legitimately) express assertions in preconditions that are “at least as visible” as the method they are guarding.

Problem 4. Annotate the following code with *preconditions* only. Consider what are the *necessary* conditions for the method bodies to (a) not fail abnormally (e.g., throw `NullPointerException`, `IndexOutOfBoundsException`, etc.), (b) pass all assertions in the code, and (c) perform their job correctly (e.g., not compute an incorrect value, given your intuition of the purpose of the method). Check the use of all of your method calls with ESC/Java2.

```
package problems;

class PrePostConditions
{
    byte b;
    int i;
    Object o;
    String s;
    static Object so;

    void m() {
        o = s.toString();
    }
}
```

```

void n() {
    b = (byte)i;
    if (b == i)
        s = new String();
    //@ assert s != null;
}

void o(int j) {
    if (j == b)
        o = null;
    //@ assert o != null;
}

void p(String t) {
    PrePostConditions.so = t.substring(3,6);
}

void q() {
    if (StaticPreconditions.o.hashCode() == 0) {
        StaticPreconditions.i = 1;
    }
    assert i == 1;
}

void r(Object o, byte b) {
    if (o == null)
        throw new IllegalArgumentException();
    if (b < 0)
        throw new IllegalArgumentException("bogus byte");
    i = 2;
}

public static void main(String[] args) {
    PrePostConditions ppc = new PrePostConditions();
    ppc.m();
    ppc.n();
    ppc.o(127);
    ppc.p("foobar");
    ppc.q();
}

class StaticPreconditions {
    static int i;
    static Object o;
}

```

Notice that many preconditions focus on variables other than the formal parameters of their method (as in the method `o(int)`). Some depend upon class attributes (as in the methods `m()` and `n()`), while others focus on “global” values like static fields of the enclosing class (as in the method `p(String)`) or static field of other classes (as in the method `q()`).

It is frequently argued that the more a method’s precondition depends upon variables other than `this` and its fields, `this`’s class and its ancestors and their fields, and the method itself’s parameters,

the more poorly designed the method is due to the complexity of its interdependencies and its consequent fragility in the face of “external” changes.

In this example in particular, the method `q()` is poorly designed due to its dependence on the (static) fields of the seemingly unrelated class `StaticPreconditions`.

Additionally, the more complex a precondition is, the harder it sometimes is to ensure it is fulfilled. This means that potentially more state has to be exposed as part of the design of related classes, as the client is responsible for ensuring the precondition of a method is fulfilled before calling the method. Consequently, complex preconditions place more burden on the client programmer than simple preconditions do.

This balancing act, between the complexity of preconditions and robustness of API methods in the face of bogus input, is supremely evident in the design of core reusable APIs in the Java platform. This topic will be discussed in more detail later in this document.

Postconditions

Postconditions are used to describe how a method should/must behave. If a method’s preconditions are fulfilled, then the (normal) postconditions *must* be true if the method terminates normally—that is, does not throw an exception or never terminates, perhaps by going into an infinite loop.

Normal postconditions are specified in JML using the **ensures** clause:

ensures-clause ::= **ensures** *pred-or-not* ;

Once again, an unspecified postcondition means “I have something in mind but I’m not writing it down.” The default postcondition for ESC/Java2 is **true**.

In a postcondition predicate one can reference any visible fields, the parameters of the method, the special keyword `\result` if the method has a non-void return type, and may contain expressions of the form `\old(E)`. `\result` is used to reference the value of the result of the method. The `\old()` expression is used to reference values in the pre-state of the method; that is, you can refer to the state of any values just *before* the method began executing.

Problem 5. Annotate the code of the previous problem with *postconditions* that match your preconditions. Consider what is known to be true when each method terminates *normally*. Check your postconditions with ESC/Java2.

Exceptional (Abnormal) Postconditions

Frequently Java APIs use exceptions and errors (all descendents of the class `Throwable`) to indicate errors or perform flow-control. Erroneous termination of a method is due to unexpected circumstances; changes in program flow are non-erroneous termination of a method due to expected, but not “normal” behavior in response to some particular expected circumstances. This “design confusion” of exceptions in Java is a primary criticism of the language.

Both situations are deemed *exceptional termination* in the JML vernacular because both use Java exceptions to signal termination of a method. To state what is known about the system when an exception is thrown, an *exceptional postcondition* is used.

Exceptional postconditions are written in JML using the **signals** clause:

signals-clause ::= **signals** (*reference-type* [*ident*]) [*pred-or-not*] ;

In many circumstances the value(s) of the exception object simply do not matter. In those circumstances the simpler form of the exceptional postcondition can be used:

```
//@ signals (java.io.IOException) aPredicate();
```

If one wants to refer to the exception object being thrown in the exceptional postcondition, a bound identifier can be included in the signals clause. E.g.,

```
//@ signals (java.io.IOException ioe) ioe.getMessage().equals("I/O Error!");
```

which is equivalent to the signals clause


```
//@ signals (java.lang.Exception e) (e instanceof IOException) ==>
//@                                     ioe.getMessage().equals("I/O Error!");
```

The default exceptional postcondition for ESC/Java2 is currently

```
//@ signals (java.lang.Exception) true;
```

Note that a **signals** clause specifies when a certain exception *may* be thrown, not when a certain exception *must* be thrown. To say that an exception must be thrown in some situation, one has to exclude that situation from other signals clauses and from all ensures clauses.

Also note that Java *errors* are *very* rarely specified and are not reasoned about at all using ESC/Java2. Also, Java “runtime exceptions” (descendents of the class `java.lang.RuntimeException`) are also rarely specified using exceptional postconditions.

Problem 6. Annotate the code of the previous problem with *exceptional postconditions*. Consider what is known to be true when each method terminates *abnormally*. If a method *cannot* terminate abnormally, annotate such using an exceptional postcondition with a **false** predicate. Check your exceptional postconditions with ESC/Java2.

Frame Axioms

Frame axioms are used to denote which fields *may* (*not must*) be modified by a method. Frame axioms are specified in JML using the *assignable* clause:

```
assignable-clause ::= assignable store-ref [ , store-ref ] ... ;
```

Special keywords `\everything` and `\nothing` are used to state that a method may modify *any* visible field (in the former case), or may modify *no* visible field (in the case of the latter). To specify that all visible field of an object *o* may be modified a globbing-like syntax is used, e.g., `assignable o.*`.

The default assignable clause for ESC/Java2 is

```
//@ assignable \everything;
```

Specifying the frame axioms of your methods is *vital* to correct reasoning with ESC/Java2. If your frame axioms are incorrect, verification is unsound. Consider the following example.

```
class BogusFrameConditions
{
    public int i;

    public void l() {
        //@ assume i == 0;
        s();
        //@ assert true == false;
    }

    public void m() {
        //@ assume i == 0;
        t();
        //@ assert true == false;
    }

    public void n() {
        //@ assume i == 0;
        u();
        //@ assert true == false;
    }

    public void o() {
```

```

    //@ assume i == 0;
    v();
    //@ assert true == false;
}

//@ requires i == 0;
//@ ensures i == 1;
public void s() {
    i++;
}

//@ requires i == 0;
//@ assignable \everything;
//@ ensures i == 1;
public void t() {
    i++;
}

//@ requires i == 0;
//@ assignable this.*;
//@ ensures i == 1;
public void u() {
    i++;
}

//@ requires i == 0;
//@ assignable i;
//@ ensures i == 1;
public void v() {
    i++;
}
}

```

ESC/Java2 produces the following output:

```

BogusFrameConditions: l() ...
    [TIME]    passed

BogusFrameConditions: m() ...
    [TIME]    passed

BogusFrameConditions: n() ...
-----
BogusFrameConditions.java:20: Warning: Possible assertion failure (Assert)
    // assert true == false;
        ^
-----

    [TIME]    failed

BogusFrameConditions: o() ...
-----
BogusFrameConditions.java:26: Warning: Possible assertion failure (Assert)
    // assert true == false;

```

[TIME] failed

Notice that the methods `l()` and `m()` *pass*, proving the bogus assertion `true == false`. The reason these methods pass is because the verification of methods whose frame axiom is `assignable \everything;` is incomplete.

The old SRC ESC/Java version 1 did not check frame conditions. This led to very specification bugs that exhibited themselves in the subtle behavior of ESC/Java verifying a (sometimes very complex) method very quickly.

ESC/Java2 now checks frame conditions, but its checking algorithm is not complete. Thus, as a general rule, one should avoid the use of `\everything` in frame axioms as much as possible at this point in time.

Problem 7. Annotate the code of the previous problem with *frame axioms*. Make your assignable clauses as precise as possible, given the body of the methods. Make sure to write a frame axiom for your `s()` method, but do not bother with the `main(String[])` method. Check your frame axioms with ESC/Java2.

Note that now that frame axioms are specified for all of your methods more specifications are verified by ESC/Java2.

Data Groups

The concept of a *data group* is a useful one, particularly with respect to specifying the frame axioms of a method and structuring one's specification, from both from a design and a verification point of view.

A data group can be thought of as a collection of fields that all relate to each other in some way: perhaps they are the key data values of a class, or set of classes; or perhaps they are all interdependent due to the design of your architecture.

One uses the `in` keyword to specify that a particular field is *in* (a part of) a specific data group. A field can be in many data groups, but data groups cannot be nested. Thus, all of the data groups of a system define a collection of (possibly overlapping) subsets of all fields.

Data groups are defined explicitly and implicitly, using the `JMLDataGroup` class in the former case, and using *implicit data groups* in the latter. `JMLDataGroup` is the JML class used to define a new data group, in lieu of a `datagroup` keyword at this point in time.

Every field implicitly defines a data group of the same name. For example, the Java field declarations

```
class C
    byte b;
    int i;
    Object o;
    ...
```

is equivalent to the following

```
class C
    // JMLDataGroup b;
    byte b; // in b;
    // JMLDataGroup i;
    int i; // in i;
    // JMLDataGroup o;
    Object o; // in o;
    ...
```

The data group most frequently used in JML specifications is an explicitly predefined data group found in the specification for the base class `java.lang.Object`. It is called the `objectState` data group. The `objectState` data group is used as a general data group that represents all non-private, transient state of an object. The `objectState` data group is only an convenience construct; it has no special significance to and of the JML tools.

A data group can be used within a frame axiom. For a data group `d`, writing `assignable d` means that all fields which are part of the data group `d` might be modified.

Problem 8. Annotate the code of the previous problem with *data groups*. Make your assignable clauses as broad as possible considering possible future extensions of the class you are specifying. Check your frame axioms with ESC/Java2.

Heavyweight Specifications

Invariants

Reasoning about Loops

Aliasing

Models

Using JML and ESC/Java2

Specifying Abstract Classes and Interfaces

Annotating Preexisting Java Code

Annotating Preexisting APIs

Designing by Contract

Copying

The JML and ESC/Java2 Homework Exercises are licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.0 license. See <http://creativecommons.org/licenses/by-nc-sa/2.0/>

Copyright © 2004 Joseph R. Kiniry

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works

Under the following conditions:

- Attribution. You must give the original author credit.
- Noncommercial. You may not use this work for commercial purposes.
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

For any reuse or distribution, you must make clear to others the license terms of this work.

Any of these conditions can be waived if you get permission from the author.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code.

See <http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>

Index

A

Abnormal Postconditions	13
Abstract Classes, Specifying	18
Aliasing	17
Annotating Existing APIs	18
Annotating Existing Java Code	18
API Annotation	18
Assertions	3
Assumptions	7

C

Code Annotation	18
Copying	19
Core JML Constructs	3

D

Data Groups	16
Design by Contract	18
Designing by Contract	18

E

Exceptional Postconditions	13
----------------------------------	----

F

Frame Axioms	14
--------------------	----

G

Ghost Variables	11
-----------------------	----

H

Heavyweight Specifications	17
----------------------------------	----

I

Interfaces, Specifying	18
Introduction	2
Invariants	17

L

Lightweight Contracts	11
-----------------------------	----

M

Models	17
--------------	----

O

objectState	17
-------------------	----

P

Postconditions	13
Preconditions	11

R

Reasoning about Loops	17
-----------------------------	----

S

Specifying Abstract Classes and Interfaces	18
--	----

U

Using JML and ESC/Java2	18
-------------------------------	----