ESC/Java2 Warnings

David Cok, Joe Kiniry, and Erik Poll

Eastman Kodak Company, University College Dublin, and Radboud University Nijmegen
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible **runtime exceptions**: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)

  • These are the most common runtime exceptions caused by coding problems (that is, not by explicitly throwing an exception)
  • They do not include nearly all of the possible runtime exceptions
  • Most of the others are explicitly thrown by various library methods
The Cast warning occurs when ESC/Java2 cannot verify that a ClassCastException will not be thrown:

```java
public class CastWarning {
    public void m(Object o) {
        String s = (String)o;
    }
}
```

results in

```
---------------------------------------------------
CastWarning.java:3: Warning: Possible type cast error (Cast)
    String s = (String)o;
   ^
---------------------------------------------------
```

But this is OK:

```java
public class CastWarningOK {
    public void m(Object o) {
        if (o instanceof String) {
            String s = (String)o;
        }
    }
}
```
So is this:

```java
public class CastWarningOK2 {
    //@ requires o instanceof String;
    public void m( Object o ) {
        String s = (String) o;
    }
}
```
The Null warning occurs when ESC/Java2 cannot verify that a NullPointerException will not be thrown:

```java
public class NullWarning {
    public void m(Object o) {
        int i = o.hashCode();
    }
}
```

results in

```
NullWarning.java:3: Warning: Possible null dereference (Null)
    int i = o.hashCode();
   ^
```

But this is OK:

```java
public class NullWarningOK {
    public void m(/*@ non_null */ Object o) {
        int i = o.hashCode();
    }
}
```
ArrayStore Warning

The ArrayStore warning occurs when ESC/Java2 cannot verify that the assignment of an object to an array element will not result in an ArrayStoreException:

```java
class ArrayStoreWarning {
    public void m(Object o) {
        Object[] s = new String[10];
        s[0] = o;
    }
}
```

results in

```
ArrayStoreWarning.java:4: Warning: Type of right-hand side possibly not a subtype of array element type (ArrayStore)
    s[0] = o;
   ^
```

But this is OK:

```java
class ArrayStoreWarningOK {
    public void m(Object o) {
        Object[] s = new String[10];
        if (o instanceof String) s[0] = o;
    }
}
```
ZeroDiv, index Warnings

- **ZeroDiv** - issued when a denominator (integer division) may be 0
- **NegSize** - issued when the array size in an array allocation expression may be negative
- **IndexNegative** - issued when an array index may be negative
- **IndexTooBig** - issued when an array index may be greater than or equal to the array length

```java
class Index {
    void m() {
        int i = 0;
        int j = 8/i;  // Causes a ZeroDiv warning
        Object[] oo = new Object[i-1]; // NegSize warning
        oo = new Object[10];
        i = oo[-1].hashCode(); // IndexNegative warning
        i = oo[20].hashCode(); // IndexTooBig warning
    }
}
```
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)

- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)

  • These are all caused by violations of explicit user-written method specifications
These warnings occur in response to user-written preconditions (requires), postconditions (ensures, signals), or assert statements.

```java
public class PrePost {
    //@ requires i >= 0;
    //@ ensures \result == i;
    public int m(int i);

    //@ ensures \result > 0;
    public int mm() {
        int j = m(-1); // Pre warning - argument must be >= 0
        return j;
    }

    //@ ensures \result > 0;
    public int mmm() {
        int j = m(0);
        return j;
    } // Post warning - result is 0 and should be > 0
}
```
Frame conditions

- To reason (modularly) about a call of a method, one must know what that method might modify: this is specified by
  - **assignable** clauses
    ```java
    //@ assignable x, o.x, this.*, o.*, a[...], a[3], a[4..5];
    ```
  - **modifies** clauses (a synonym)
  - **pure** modifier
    ```java
    //@ pure
    public int getX() { return x; }
    ```

- Assignable clauses state what fields may be assigned within a method - this is the set of what might be modified

- The default assignable clause is **assignable \everything**; (but it is better to be explicit because \everything; is not fully implemented and ESC/Java2 can reason better with more explicit frame conditions)

- A **pure** method is **assignable \nothing**;
Frame conditions

- A **Modifies** warning indicates an attempt to assign to an object field that is not in a modifies clause.

- Note: Some violations of modifies clauses can be detected at typecheck time.

- Note also: Handling of frame conditions is an active area of research.
For example, in

```java
class ModifiesWarning {
    int i;

    // Assignable i;
    void m(/*@ non_null */ ModifiesWarning o) {
        i = 1;
        o.i = 2;  // Modifies warning
    }
}
```

we don’t know if \texttt{o} equals \texttt{this}; since only \texttt{this.i} may be assigned, ESC/Java2 produces

```java
---------------------------------------------------
ModifiesWarning.java:7: Warning: Possible violation of modifies clause (Modifies)
o.i = 2;  // Modifies warning
  ^
Associated declaration is "ModifiesWarning.java", line 4, col 6:
//@ assignable i;
  ^
---------------------------------------------------
```

David Cok, Joe Kiniry & Erik Poll - ESC/Java2 & JML Tutorial – p.12/?
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)

- warnings about possible specification violations: (Precondition, Postcondition, Modifies)

- non null violations (NonNull, NonNullInit)
  - These warnings relate to explicit non_null field or parameter specifications
Class fields declared non_null must be initialized to values that are not null in each constructor, else a NonNullInit warning is produced.

```java
class NonNullInit {
    /** non_null */ Object o;

    public NonNullInit() { }
}
```

produces

```
NonNullInit.java:4: Warning: Field declared non_null possibly not initialized (NonNullInit)
    public NonNullInit() { }
    ^

Associated declaration is "NonNullInit.java", line 2, col 6:
    /** non_null */ Object o;
    ^
```

David Cok, Joe Kiniry & Erik Poll - ESC/Java2 & JML Tutorial – p.14
NonNull warning

A NonNull warning is produced whenever an assignment is made to a field or variable that has been declared non_null but ESC/Java2 cannot determine that the right-hand-side value is not null.

```java
public class NonNull {
    /*@ non_null */ Object o;

    public void m(Object oo) { o = oo; } // NonNull warning
}
```

produces

```
NonNull.java:4: Warning: Possible assignment of null to variable declared non_null (NonNull)
    public void m(Object oo) { o = oo; } // NonNull warning
   ^

Associated declaration is "NonNull.java", line 2, col 6:
    /*@ non_null */ Object o;
   ^
```

David Cok, Joe Kiniry & Erik Poll - ESC/Java2 & JML Tutorial – p.15/?
**nonnull warning**

But this is OK

```java
public class NonNull {
    /*@ non_null */ Object o;
    public void m(/*@ non_null */ Object oo) { o = oo; }
}
```

So is this

```java
public class NonNull {
    /*@ non_null */ Object o;
    public void m(Object oo) {
        if (oo != null) o = oo;
    }
}
```

So is this

```java
public class NonNull {
    /*@ non_null */ Object o;
    public void m() {
        o = new Object();
    }
}
```

nonnull can be applied to
- a field
- a formal parameter
- a return value
- a local variable
- ghost and model variables
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)
- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)
- non null violations (NonNull, NonNullInit)
- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)

• These are caused by violations of specifications in a routine body
Body assertions

- **Assert**: warning occurs when an `assert` annotation may not be satisfied

- **Reachable**: not in JML, only in ESC/Java2; occurs with the `//@ unreachable;` annotation, which is equivalent to `//@ assert false;`

Example:

```java
class AssertWarning {
    //@ requires i >= 0;
    public void m(int i) {
        //@ assert i >= 0; // OK
        --i;
        //@ assert i >= 0; // FAILS
    }
    public void n(int i) {
        switch (i) {
            case 0,1,2: break;
            default: //@ unreachable; // FAILS
        }
    }
}
```
Loop assertions

- A **loop_invariant** assertion just before a loop asserts a predicate that is true prior to each iteration and at the termination of the loop (or a LoopInv warning is issued).

- A **decreases** assertion just before a loop asserts a (int) quantity that is non-negative and decreases with each iteration (or a DecreasesBound warning is issued).

- **Caution:** Loops are checked by unrolling a few times.

Example:

```java
public class LoopInvWarning {
    public int max(/* @ non_null */ int[] a) {
        int m=\text{Integer.MAX\_VALUE};
        //@ loop_invariant (\forall int j; 0<=j && j<i; a[j] <= m);
        //@ decreases a.length - i - 1;
        for (int i=0; i<a.length; ++i) {
            if (m < a[i]) m = a[i];
        }
        return m;
    }
}
```

In the scope of the loop variable
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)
- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)
- non null violations (NonNull, NonNullInit)
- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)
- warnings about possible class specification violations: (Invariant, Constraint, Initially)
Invariant and constraint clauses generate additional postconditions for every method. If they do not hold, appropriate warnings are generated:

```java
public class Invariant {
    public int i, j;
    //@ invariant i > 0;
    //@ constraint j > \old(j);

    public void m() {
        i = -1; // will provoke an Invariant error
        j = j-1; // will provoke a Constraint error
    }
}
```
An initially clause is a postcondition for every constructor:

```java
public class Initially {
    public int i; // @ initially i == 1;

    public Initially() { } // does not set i - Initially warning
}
```

produces

```
Initially.java:5: Warning: Possible violation of initially condition at constructor exit (Initially)
    public Initially() { } // does not set i - Initially warning
   ^
Associated declaration is "Initially.java", line 3, col 20:
    public int i; // @ initially i == 1;
   ^
```

Initially warning
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)

- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)

- non null violations (NonNull, NonNullInit)

- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)

- warnings about possible class specification violations: (Invariant, Constraint, Initially)

- exception problems (Exception)

  - These warnings are caused by undeclared exceptions
- Java **Errors** (e.g. OutOfMemoryError) can be thrown at any time
  - No declarations are needed in throws clauses
  - No semantics are implied by JML
  - No checking is performed by ESC/Java2
Checked Exceptions

- Java **checked** exceptions (e.g. FileNotFoundException) are Exceptions that are not RuntimeExceptions:
  - Declarations of exceptions mentioned in the body are required in throws clauses
  - ESC/Java2 checks during typechecking that throws declarations are correct (as a Java compiler does)
  - Typically specified in signals clauses in JML
  - ESC/Java2 checks via reasoning that signals conditions hold
  - Default specification is that *declared* exceptions may occur: signals (Exception) true;
  - ESC/Java2 presumes that checked exceptions not declared in a throws clause will not occur.
Unchecked Exceptions

- Java **unchecked** exceptions (e.g. NoSuchElementException) are RuntimeExceptions:
  - Java does not require these to be declared in throws clauses
  - ESC/Java2 is stricter than Java - it will issue an Exception warning if an unchecked exception might be *explicitly* thrown but is not declared in a throws declaration
  - Caution: currently ESC/Java2 will assume that an undeclared unchecked exception will not be thrown, even if it is specified in a signals clause - Declare all unchecked exceptions that might be thrown! (e.g. especially when there is no implementation to check).
So this

public class Ex {
    public void m(Object o) {
        if (!(o instanceof String)) throw new ClassCastException();
    }
}

produces

Ex.java:4: Warning: Possible unexpected exception (Exception)
    }
   ^

Execution trace information:
    Executed then branch in "Ex.java", line 3, col 32.
    Executed throw in "Ex.java", line 3, col 32.

Turn off this warning by

- declaring the exception in a throws clause
- using //@ nowarn Exception; on the offending line
- using a -nowarn Exception command-line option
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)

- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)

- non null violations (NonNull, NonNullInit)

- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)

- warnings about possible class specification violations: (Invariant, Constraint, Initially)

- exception problems (Exception)

- multithreading (Race, RaceAllNull, Deadlock)

  - These warnings are caused by potential problems with monitors
  - Multithreading problems caused by the absence of any synchronization are not detected.
Race conditions

- Java defines monitors associated with any object and allows critical sections to be guarded by synchronization statements.
- ESC/Java permits fields to be declared as monitored by one or more objects.
- To read a monitored field, at least one monitor must be held (or a Race warning is issued).
- To write a monitored field, all non-null monitors must be held (or a Race warning is issued).
- To write a monitored field, at least one of its monitors must be non-null (or a RaceAllNull warning is issued).
For example,

```java
public class RaceWarning {
    //@ monitored
    int i;

    void m() {
        i = 0; // should have a synchronization guard
    }
}
```

produces

```plaintext
RaceWarning.java:6: Warning: Possible race condition (Race)
    i = 0; // should have a synchronization guard
        ^
Associated declaration is "RaceWarning.java", line 2, col 6:
    //@ monitored
        ^
```

David Cok, Joe Kiniry & Erik Poll - ESC/Java2 & JML Tutorial – p.30/?
Deadlocks occur when each thread of a group of threads needs monitors held by another thread in the group.

One way to avoid this is to always acquire monitors in a specific order.

This requires

- the user state a (partial) order for monitors (typically using an axiom)
- that it be clear before acquiring a monitor that the thread does not hold any ‘larger’ monitors (typically a precondition)

Checking for Deadlock warnings is off by default but can be turned on with -warn Deadlock.
Deadlock warnings

For example:

```java
public class DeadlockWarning {
    /** non_null */ final static Object o = new Object();
    /** non_null */ final static Object oo = new Object();

    //@ axiom o < oo;

    //@ requires \max(\lockset) < o;
    public void m() {
        synchronized(o) { synchronized(oo) { }}
    }

    //@ requires \max(\lockset) < o;
    public void mm() {
        synchronized(oo) { synchronized(o) { }} // Deadlock warning
    }
}
```
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)
- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)
- non null violations (NonNull, NonNullInit)
- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)
- warnings about possible class specification violations: (Invariant, Constraint, Initially)
- exception problems (Exception)
- multithreading (Race, RaceAllNull, Deadlock)
- a few others (OwnerNull, Uninit, Unreadable, Writable)
Other warnings

- **Uninit**: used with the `uninitialized` annotation
- **OwnerNull**: see the ESC/Java User Manual for a description
- **Unreadable**: occurs with the `readable_if` annotation on shared variables. [JML’s change of syntax from `readable_if` to `readable` is not complete in ESC/Java2.]
- **Writable**: occurs with the `writable_if` annotation on shared variables. [JML’s change of syntax from `writable_if` to `writable` is not complete in ESC/Java2.]
For complicated bodies, the warning messages give some information about which if-then-else branches caused the warning:

```java
public class Trace {
    //@ ensures \result > 0;
    int m(int i) {
        if (i == 0) return 1;
        if (i == 2) return 0;
        return 4;
    }
}
```

produces

Trace.java:8: Warning: Postcondition possibly not established (Post)
  ^
Associated declaration is "Trace.java", line 2, col 6:
  //@ ensures \result > 0;
  ^

Execution trace information:
  Executed else branch in "Trace.java", line 4, col 4.
  Executed then branch in "Trace.java", line 5, col 16.
  Executed return in "Trace.java", line 5, col 16.
Sometimes when a specification is found to be invalid, ESC/Java2 will produce a counterexample context.

A full context will be produced with the -counterexample option.

These are difficult to read, but can give information about the reason for failure.

They state formulae that the prover believes to be true; if there is something you think should not be true, that is a hint about the problem.

Note also: Typically only one warning will be issued in a given run.