Elvis and Other Null-Safe Operators for Java

AUTHOR(S):

Neal Gafter

Note that the author does not specifically advocate adding these features to the Java programming language. Rather, this document is offered as an example of a language change proposal in a form suitable for consideration in the JDK7 small language changes JSR. Specifically, it is more like a specification than a tutorial or sales job.

OVERVIEW

FEATURE SUMMARY:

The `?.` null-safe member selection operator provides the same meaning as `.` (member selection), except when the left-hand-side evaluates to **`null`**, in which case any subexpressions on the right-hand-side are not evaluated and the `?.` expression yields **`null`**.

The `?:` binary "Elvis" operator results in the value of the left-hand-side if it is not **`null`**, avoiding evaluation of the right-hand-side. If the left-hand-side is **`null`**, the right-hand-side is evaluated and is the result.

The `?[]` indexing operator operates on a left-hand-side that is an array of object type. If the value of the left-hand operand is **`null`**, that is the result. If the left-hand-operand is not **`null`**, the index subexpression is evaluated and used to index the array, yielding the result.

These three operators always result in a value, not a variable.

MAJOR ADVANTAGE:

Simplifies and clarifies code in common patterns involving checks for **`null`**.

MAJOR BENEFIT:

Some common coding patterns are simplified. *[Neal asks: How is this section supposed to be different from the previous section?]*

MAJOR DISADVANTAGE:

Slightly more complex language specification with little increased expressiveness of the language. Encourages, rather than discourages, the use of **`null`** values in APIs.

ALTERNATIVES:

The most important alternative is to leave the language as it is and let

programmers do things the hard way. For programmers, there are library-based alternatives, but they are hardly more concise than doing things the hard way. See, for example, Stephan Schmidt's discussion of *Better Strategies for Null Handling in Java*[1].

EXAMPLES

SIMPLE EXAMPLE:

```
String s = mayBeNull?.toString() ?: "null";
```

ADVANCED EXAMPLE:

Given a Java class

```
class Group {
  Person[] members; // null if no members
}
class Person {
   String name; // may be null
}
Group g = ...; // may be null
```

we can compute the name of a member if the group is non-null and non-empty and the first member has a known (non-null) name, otherwise the string **"nobody"**:

```
final String aMember = g?.members?[0]?.name ?: "nobody";
```

Without this feature, we would have to write the roughly equivalent code

```
Person[] t1= g != null ? g.members : null;
Person t2 = t1 != null ? t1[0] : null;
Name t3 = t2 != null ? t2.name : null;
final String aMember = t3 != null ? t3 : "nobody";
```

DETAILS

SPECIFICATION:

Lexical:

We do not add any tokens to the language. Rather, we introduce new operators that are composed of a sequence of existing tokens.

Syntax:

The folllowing new grammar rules are added to the syntax

PrimaryNoNewArray:

NullSafeFieldAccess
NullSafeMethodInvocation
NullSafeClassInstanceCreationExpression
NullSafeArrayAccess

NullSafeFieldAccess:

PrimaryNoNewArray **?** **.** *Identifier*

NullSafeMethodInvocation:

PrimaryNoNewArray **?** **.** *NonWildTypeArguments*$_{opt}$ *Identifier* **(** *ArgumentList*$_{opt}$ **)**

NullSafeClassInstanceCreationExpression:

PrimaryNoNewArray **?** **.** new *TypeArguments*$_{opt}$ *Identifier* *TypeArguments*$_{opt}$ **(** *ArgumentList*$_{opt}$ **)** *ClassBody*$_{opt}$

NullSafeArrayAccess:

PrimaryNoNewArray **?** **[** *Expression* **]**

ConditionalExpression:

ElvisExpression

ElvisExpression:

ConditionalOrExpression **?** **:** *ConditionalExpression*

Semantics:

A *null-safe field access* expression **`e1?.name`** first evaluates the expression **`e1`**. If the result is **`null`**, then the null-safe field access expression's result is **`null`**. Otherwise, the result is the same as the result of the expression **`e1.name`**. In either case, the type of the result is the same as the type of **`e1.name`**. It is an error if this is not a reference type.

A *null-safe method invocation* expression **`e1?.name(args)`** first evaluates the expression **`e1`**. If the result is **`null`**, then the null-safe method invocation expression's result is **`null`**. Otherwise the arguments are

evaluated and the result is the same as the result of the invocation expression **`e1.name(args)`**. In either case, the type of the result is the same as the type of **`e1.name(args)`**. It is an error if this is not a reference type.

A *null-safe class instance creation expression* **`e1?.new name(args)`** first evaluates the expression **`e1`**. If the result is **`null`**, then the null-safe class instance creation expression's result is **`null`**. Otherwise, the arguments are evaluated and the result is the same as the result of the class instance creation expression **`e1.new name(args)`**. In either case, the type of the result is the same as the type of **`e1.new name(args)`**.

A *null-safe array access* expression **`e1?[e2]`** first evaluates the expression **`e1`**. If the result is **`null`**, then the null-safe array access expression's result is **`null`**. Otherwise, **`e2`** is evaluated and the result is the same as the result of **`e1[e2]`**. In either case, the type of the result is the same as the type of **`e1[e2]`**. It is an error if this is not a reference type.

An *Elvis expression* **`e1?:e2`** first evaluates the expression **`e1`**. It is an error if this is not a reference type. If the result is non-**`null`**, then that is the Elvis expression's result. Otherwise, **`e2`** is evaluated and is the result of the Elvis expression. In either case, the type of the result is the same as the type of **`(e1!=null)?e1:e2`**. *[Note: this section must mention bringing the operands to a common type, for example by unboxing when e2 is a primitive, using the same rules as the ternary operator]*

Exception Analysis:

JLS section 12.2.1 (exception analysis of expressions) is modified to read as follows. Additions are shown in bold.

A method invocation expression **or null-safe method invocation expression** can throw an exception type *E* iff either:

- The method to be invoked is of the form *Primary.Identifier* **or *Primary?.Identifier*** and the *Primary* expression can throw *E*; or
- Some expression of the argument list can throw *E*; or
- *E* is listed in the throws clause of the type of method that is invoked.

A class instance creation expression **or null-safe class instance creation expression** can throw an exception type *E* iff either:

- The expression is a qualified class instance creation expression **or a null-safe class instance creation expression** and the qualifying expression can throw *E*; or
- Some expression of the argument list can throw *E*; or
- *E* is listed in the throws clause of the type of the constructor that is invoked; or
- The class instance creation expression **or null-safe class instance creation expression** includes a *ClassBody*, and some instnance initializer block or instance variable initializer expression in the *ClassBody* can throw *E*.

For every other kind of expression, the expression can throw type *E* iff one of its immediate subexpressions can throw *E*.

Definite Assignment:

JLS section 16.1 (definite assignment and expressions) is augmented with the following new subsections

16.1.x Null-safe Method Invocation

- *v* is definitely assigned after **`e1?.name(args)`** iff *v* is definitely assigned after **`e1`**.
- *v* is definitely unassigned after **`e1?.name(args)`** iff *v* is definitely unassigned after **`args`**.
- in an expression of the form **`e1?.name(args)`**, *v* is [un]assigned before args iff *v* is [un]assigned after **`e1`**.

16.1.x Null-safe Class Instance Creation Expression

- *v* is definitely assigned after **`e1?.new name(args)`** iff *v* is definitely assigned after **`e1`**.
- *v* is definitely unassigned after **`e1?.new name(args)`** iff *v* is definitely unassigned after args.
- in an expression of the form **`e1?.new name(args)`**, *v* is [un]assigned before args iff *v* is [un]assigned after **`e1`**.

16.1.x Null-safe Array Access

- *v* is definitely assigned after **`e1?[e2]`** iff *v* is definitely assigned after **`e1`**.
- *v* is definitely unassigned after **`e1?[e2]`** iff *v* is definitely unassigned after **`e2`**.
- in an expression of the form **`e1?[e2]`**, *v* is [un]assigned before **`e2`** iff *v* is [un]assigned after **`e1`**.

16.1.x Elvis Operator

- *v* is definitely assigned after **`e1?:e2`** iff *v* is definitely assigned after **`e1`**.
- *v* is definitely unassigned after **`e1?:e2`** iff *v* is definitely unassigned after **`e2`**.
- in an expression of the form **`e1?:e2`**, *v* is [un]assigned before **`e2`** iff *v* is [un]assigned after **`e1`**.

COMPILATION:

These new expression forms can be desugared as follows:

- **`e1?.name`** is rewritten as **`(t != null ? t.name : null)`**
- **`e1?.name(args)`** is rewriten as **`(t != null ? t.name(args) : null)`**

- **`e1?.new name(args)`** is rewritten as **`(t != null ? t.new name(args) : null)`**
- **`e1?[e2]`** is rewritten as **`(t != null ? t[e2] : null)`**
- **`e1?:e2`** is rewritten as **`(t != null ? t : e2)`**

where **`t`** is a new temporary that holds the computed value of the expression **`e1`**.

TESTING:

This feature can be tested by exercising the various new expression forms, and verifying their correct behavior in erroneous and non-erroneous situations, with or without null as the value of the left-hand operand, and with respect to definite assignment and exception analysis.

LIBRARY SUPPORT:

No library support is required.

REFLECTIVE APIS:

No reflective APIs require any changes. However, the not-yet-public javac Tree APIs, which describe the syntactic structure of Java statements and expressions, should be augmented with new tree forms for these new expression types.

OTHER CHANGES:

No other platform changes are required.

MIGRATION:

No migration of existing code is recommended. These new language features are mainly to be used in new code. However, IDEs should provide refactoring advice for taking advantage of these new operators when existing code uses the corresponding idiom.

COMPATIBILITY

BREAKING CHANGES:

No breaking changes are caused by this proposal.

EXISTING PROGRAMS:

Because the changes are purely the introduction of new expression forms, there is no impact on the meaning of existing code.

REFERENCES

EXISTING BUGS:

4151957: Proposal: null-safe field access operator

URL FOR PROTOTYPE (optional):

No Java prototype exists at this time. However, Groovy (among others) has the Elvis operator.

OTHER REFERENCES

[1] Stephan Schmidt's discussion of Better Strategies for Null Handling in Java [http://www.slideshare.net/Stephan.Schmidt/better-strategies-for-null-handling-in-java]
[2] Groovy Operators
[3] Stephen Colebourne's glossy brief on null-safe operators
[4] Summary of three recent (but underspecified) language change polls