

Refinements in Java - Syntax

This survey aims to assess the best syntax for the implementation of Refinement Types in Java. Refinement types have been proposed as an incremental approach for program verification that is directly embedded in the programming language. We propose the usage of Refinement Types within the Java programming language not only as a means for program verification but also for fault localization and efficient mutation in the context of software repair.

All responses to this questionnaire will be anonymous and the treatment of the information obtained will be used only and exclusively in the context [REDACTED]

It takes about 7 minutes to complete this survey. If you are using a mobile device, please use the landscape/horizontal mode to have a better experience.
We thank you in advance for your precious collaboration.

*Obrigatório

Familiarity with tools

This section serves as assessment of the knowledge ground on the different technologies used in the following sections.

1. How familiar are you with Java? *

[REDACTED]

- ☐ Very familiar
- ☐ Familiar
- ☐ Vaguely familiar
- ☐ Not familiar

2. How familiar are you with Functional Languages (e.g.: Haskell, Scala)? *

[REDACTED]

- ☐ Very familiar
- ☐ Familiar
- ☐ Vaguely familiar
- ☐ Not familiar

3. How familiar are you with JML (Java Modeling Language)? *

[REDACTED]

- ☐ Very familiar
- ☐ Familiar
- ☐ Vaguely familiar
- ☐ Not familiar

4. How familiar are you with Refinement Types? *

[REDACTED]

- ☐ Very familiar *Avançar para a pergunta 5*
- ☐ Familiar
- ☐ Vaguely familiar
- ☐ Not familiar

Introduction to Refinement Types

Refinement Types extend a language with predicates (boolean expressions) over the basic types.
A popular syntax is $\{v : T \mid p(v)\}$, of which $\{x : \text{Integer} \mid x > 0\}$ is an instance.

The example below represents a way to apply refinements in Java, where the variable `y` has the type `int` and a refinement that only allows `y` to have positive values which are lesser than 50. When the variable is assigned the value 10 no errors will be shown, but if the assigned value is 100 the compiler will send a refinement type error to the developer.

Simple Refinement Usage

```
@Refinement("y > 0 && y < 50")  
int y = 10; //okay  
int y = 100; //okay in Java, refinement type error
```

In the following sections you will be presented with several syntax options for the implementation of refinements in Java.

Wildcard Variables

To represent a variable without using its complete name, we can use a wildcard in the refinement that will be associated with the variable value.

Analyse the following syntax possibilities for wildcard variables.

A

```
@Refinement("\\v > 0")  
int biggerThanZero = 10;
```

B

```
@Refinement("? > 0")  
int biggerThanZero = 10;
```

C

```
@Refinement("_ > 0")  
int biggerThanZero = 10;
```

5. Evaluate your preference on each of the above syntaxes. *

	Not acceptable	Acceptable	Preferable
A	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
B	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

For the following sections we will be using "\v" as the wildcard variable.

Refinements in Variables

In this section we present syntax examples for refinements in variables. In the example, the refinements are related to a grading system from 0 to 20, and the following conditions are expressed:

- negativeGrade is an int smaller than 10;
- excellentGrade is an int equal to 19 or to 20;
- goodGrade is an int with a value between negativeGrade and excellentGrade

Analyse the following examples with syntax possibilities.

A

```
@Refinement("negativeGrade < 10")
int negativeGrade = 8;
@Refinement("excellentGrade == 19 || excellentGrade == 20")
int excellentGrade = 19;
@Refinement("goodGrade > negativeGrade && goodGrade < excellentGrade")
int goodGrade = 17;
```

B

```
@Refinement("{x | x < 10}")
int negativeGrade = 8;
@Refinement("{y | y == 19 || y == 20}")
int excellentGrade = 19;
@Refinement("{x | x > negativeGrade && x < excellentGrade}")
int goodGrade = 17;
```

6. Evaluate your preference on each of the above syntaxes. *

	Not acceptable	Acceptable	Preferable
A	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
B	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Refinements
in Methods

In this section we present syntax examples for refinements in methods, which includes refinements for the parameters and the return value.

These refinements express the following conditions:

- grade, the first parameter, is an int greater than or equal to 0;
- scale, the second parameter, is a positive int;
- the return value must be an int between 0 and 100.

Analyse each of the examples below.

A

```
@Refinement("\\v >= 0 && \\v <= 100")
public static int percentageFromGrade(@Refinement("grade >= 0") int grade,
                                       @Refinement("scale > 0") int scale){...}
```

B

```
@Refinement("{grade >= 0} -> {scale > 0} -> {\\v >= 0 && \\v <= 100}")
public static int percentageFromGrade(int grade, int scale){...}
```

7. Evaluate your preference on each of the above syntaxes. *

	Not acceptable	Acceptable	Preferable
A	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
B	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Alias for Refinement type

Several implementations of refinement types include alias for a group of predicates. In this section we present possible syntaxes for the creation of the alias and their usage in variable refinements.

In the example:

- PtGrade is a refinement alias that describes an int between 0 and 20 - grade range used in the Portuguese higher education system.

Analyse the following syntax examples.

A

```
@Refinement("PtGrade refines Integer where (\\v >= 0 && \\v <= 20)")
class MyClass{
    ...
    @Refinement("positiveGrade == PtGrade && positiveGrade >= 10")
    int positiveGrade = 12;
}
```

B

```
@Refinement("type PtGrade(int x) { x >= 0 && x <= 20}")
class MyClass{
    ...
    @Refinement("PtGrade(positiveGrade) && positiveGrade >= 10")
    int positiveGrade = 12;
}
```

C

File *PtGrade.java*

```
@Refinement("{int x | x >= 0 && x <= 20}")
@Retention(RetentionPolicy.CLASS)
@Target({ ElementType.METHOD, ElementType.FIELD,
          ElementType.LOCAL_VARIABLE,
          ElementType.PARAMETER, ElementType.TYPE })
public @interface PtGrade {}
```

File *MyClass.java*

```
class MyClass{
    ...
    @PtGrade @Refinement("positiveGrade >= 10")
    int positiveGrade = 12;
}
```

8. Evaluate your preference on each of the above syntaxes. *

	Not acceptable	Acceptable	Preferable
A	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
B	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Ghost Functions

To invoke functions inside the Refinements, these functions must be declared in the program as ghost functions, that are only relevant for the specification of the program properties. These functions work as uninterpreted functions, which means that only their signature is needed and not their implementation.

In this section we present possible placements for the declaration of ghost functions inside the class *MyList*. Note that all annotations need a target (a code element), so the refinements cannot be loose in the code.

len is the ghost function, that receives a *List*, and returns an *int* value. This ghost function is then used inside the refinements of the following Java functions:

- *createList()* : the refinement ensures that the *len* of the returned list is equal to 0;
- *append()*: the refinement ensures that the returned *List* has the same *len* as the given list, plus one.

A

```

@Refinement("ghost int len(List xs)")
class MyList{
    static final int MAX_VALUE = 50;

    @Refinement("len(\\v) == 0")
    public List createList(){...}

    @Refinement("len(\\v) == 1 + len(xs)")
    public List append(List xs, int k){...}
}

```

B

```

class MyList{
    static final int MAX_VALUE = 50;

    @Refinement("ghost int len(List xs)")
    @Refinement("len(\\v) == 0")
    public List createList(){...}

    @Refinement("len(\\v) == 1 + len(xs)")
    public List append(List xs, int k){...}
}

```

C

```

class MyList{
    @Refinement("ghost int len(List xs)")
    static final int MAX_VALUE = 50;

    @Refinement("len(\\v) == 0")
    public List createList(){...}

    @Refinement("len(\\v) == 1 + len(xs)")
    public List append(List xs, int k){...}
}

```


9. Evaluate your preference on each of the above syntaxes. *

	Not acceptable	Acceptable	Preferable
A	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
B	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Conclusion

10. Leave you suggestions or comments.

11. If you want to know the results of this study, please insert your email.

Thank you for the participation!
