i1-Welcome

Welcome to this fine-grained code patterns study. This survey is being conducted by Juan Manuel Florez and Andrian Marcus from the University of Texas at Dallas. We expect this survey to take **no more than 15 minutes** to complete.

You can email any questions or concerns to *jflorez@utdallas.edu*

i2-identifying-info

Please enter your four-digit participant number as it appears in the invitation email

What is the length of your experience with the Java programming language in a **professional** setting?

less than 1 year

Between 1 and 2 years

Between 2 and 3 years

Between 3 and 4 years

Between 4 and 5 years

5 or more years

Summary

Expected Time Allocation

The survey is divided in three parts. These are the estimated times for each:

1. **Background information:** approx. 5 minute reading.
2. **Sample question:** approx. 1 minute reading
3. **Survey questions:** 7 questions approx. 1 minute per question.

i3-intro-types

Background Information

Please read the following information carefully before proceeding with the rest of the survey

We are studying the implementations of *data constraints* in Java. We define a data constraint as *"any restriction of the possible values of a variable in the software domain".*

The following are some examples of data constraint descriptions as they would be found in software documents. The constraints are rewritten in a simplified language in the right column:

Constraint Description	**Simplified Constraint**
If **package *weight is greater than 20 kg***...	package weight > 20
The options for ***marital status*** *are:* ***single, married, or, divorced***...	marital status must be one of {single, married, divorced}
If the ***agent status is set to available*** at that time...	agent status must be "available"
The ***minimum frequency*** *value is set to* ***20*** by default...	minimum frequency = 20

We have identified a set of constraint implementation patterns (CIPs) that may be used to implement data constraints in Java code. The relevant definitions will appear alongside each question, so **you do not need to memorize them**. We present two definitions here for illustration.

> *In all the definitions, when we refer to a value, we mean any construct that returns a value, specifically a variable access, method call, or field access.*

Pattern name: null-check.
Pattern description: A value is checked for nullity using the == or != operators. The value may be the first or second operator.
Pattern generic form:

```
nullableValue == null

null != obj.nullableValue

obj.getNullableValue() == null
```

Pattern example: The portion of the following code highlighted in yellow is an example of the **null-check** pattern in real code:

```
// Read WIN configuration file
if (configFile != null) {
  FileReader fileReader = new FileReader(configFile);
  BufferedReader reader = new BufferedReader(fileReader);
  timeZone = reader.readLine();
  channelInfo.clear();
  ...
```

Pattern name: null-empty-check
Pattern description: A string value is checked for nullity using the == or != operators and immediately after compared to the empty string using the equals method. The two operations are combined using the && or || operators. The operands in each equality may be in any order.
Pattern generic form:

```
stringValue != null && !stringValue.equals("")

obj.stringValue == null || obj.stringValue.equals("")

null != obj.getStringValue() && !"".equals(obj.getStringValue())
```

Pattern example: The portion of the following code highlighted in yellow is an example of the **null-empty-check** pattern in real code:

```
if (projectFileName == null || "".equals(projectFileName)) {
    if (ProjectManager.getManager().getCurrentProject() != null) {
        projectFileName = ProjectManager.getManager()
            .getCurrentProject().getName();
    }
}
```

Note that the patterns that are defined on expressions can appear in any statement type where expressions are grammatically correct in Java, for example:

```
if( value != null ) // null-check in if statement

return value != null; // null-check in return statement

while( value != null ) // null-check in while statement

boolean res = value == null || "".equals(value); // null-empty-check in assignme
```

i4-instructions

Sample question

In the next section, you will be shown 7 constraints and their implementations, one by one.

You will be asked to select the pattern that the implementation matches from a list of options, or "None of the above" if it matches none of the options.

For example:

Consider the bold text in the following paragraph:

*If **configuration file is not available** or readable it will default to ‘UTC’.*

Which contains the constraint:

configuration file is not available

And is implemented in the highlighted portion of this code:

```
// Read WIN configuration file
if (configFile != null) {
  FileReader fileReader = new FileReader(configFile);
  BufferedReader reader = new BufferedReader(fileReader);
  timeZone = reader.readLine();
  channelInfo.clear();
  ...
```

(If you need to, you can see the full method, or see the full class)

Given the following pattern definitions:

null-check: A value is checked for nullity using the == or != operators. The value may be the first or second operator.

```
nullableValue == null

null != obj.nullableValue
```

null-empty-check: A string value is checked for nullity using the == or != operators and immediately after compared to the empty string using the equals method. The two operations are combined using the && or || operators. The operands in each equality may be in any order.

```
obj.stringValue == null || obj.stringValue.equals("")

null != obj.getStringValue() && !"".equals(obj.getStringValue())
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

- null-empty-check
- null-check
- None of the above

The correct answer in this example is *null-check*.

rh-60

(Constraint 1/7) Consider the bold text in the following paragraph:

> *The operator ToNumber converts its argument to a value of type Number: The result is 1 if* ***the argument is true****.*

Which contains the constraint:

> *argument must be true*

And is implemented in the highlighted portion of this code:

```
if (val instanceof Boolean)
    return ((Boolean) val).booleanValue() ? 1 : +0.0;
```

(If you need to, you can see the full method, or see the full class)

Given the following pattern definitions:

constant-argument: A literal value (string, integer, boolean) is passed as a parameter to a method call. The call can have other non-literal parameters.

```
obj.method("value")

obj.method(1, arg2, obj.arg3)
```

null-check: A value is checked for nullity using the == or != operators.

```
value == null

null != obj.value
```

boolean-property: A value of type Boolean is checked in a Boolean expression.

```
value // Boolean variable

obj.getValue() // Boolean return

obj.value // Boolean field
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

constant-argument

null-check

boolean-property

None of the above

jedit-17-false

(Constraint 2/7) Consider the bold text in the following paragraph:

> ***Files that you do not have write access to*** *are opened in read-only mode, where editing is not permitted.*

Which contains the constraint:

> *file is not accessible*

And is implemented in the highlighted portion of this code:

```
VFS vfs = VFSManager.getVFSForPath(getPath());
if (((vfs.getCapabilities() & VFS.WRITE_CAP) == 0) ||
    !vfs.isMarkersFileSupported())
{
 VFSManager.error(view, path, "vfs.not-supported.save",
  new String[] { "markers file" });
 return false;
}
```

(If you need to, you can see the full method, or see the full class)

Given the following pattern definitions:

null-empty-check: A string value is checked for nullity using the == or != operators and immediately after compared to the empty string using the equals method. The two operations are combined using the && or || operators. The operands in each equality may be in any order.

```
value != null && !value.equals("")

null != obj.getValue() && !"".equals(obj.getValue())
```

binary-flag-check: An integer value is operated with a bitwise AND operator (&) against an integer variable, and then the result is compared with == or != against another integer value (literal or variable).

```
value & FLAG == 0

obj.value & FLAG == res
```

binary-comparison: Two variables are compared using one of the relational operators (==, !=, >, <, >=, <=). Use of the equals method is considered an operator in this case. Neither of the operands may be the literal 'null'.

```
obj1.valueA <= obj2.valueB

obj.getValueA().equals(valueB)
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

null-empty-check

binary-flag-check

binary-comparison

None of the above

ant-4

(Constraint 3/7) Consider the bold text in the following paragraph:

> *In the first example, if the* ***property is set*** *(to any value, e.g. false), the target will be run.*

Which contains the constraint:

> *property must be set*

And is implemented in the highlighted portion of this code:

```
/**
 * Returns true if the object is null or an empty string.
 *
 * @param value Object
 * @return boolean
 * @since Ant 1.8.0
 */
private static boolean nullOrEmpty(Object value) {
    return value == null || "".equals(value);
}
```

(If you need to, you can see the full class)

Given the following pattern definitions:

null-empty-check: A string value is checked for nullity using the == or != operators and immediately after compared to the empty string using the equals method. The two operations are combined using the && or || operators. The operands in each equality may be in any order.

```
value != null && !value.equals("")
```

```
null != obj.getValue() && !"".equals(obj.getValue())
```

equals-or-chain: Equality expressions (using the == operator) or equals method calls are chained by “or” operators in an expression checking possible values of a variable.

```
value == 1 || value == 2 || value == 3

value.equals("val1") || value.equals("val2") || value.equals("val3")
```

boolean-property: A value of type Boolean is checked in a Boolean expression.

```
value // Boolean variable

obj.getValue() // Boolean return

obj.value // Boolean field
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

null-empty-check

equals-or-chain

boolean-property

None of the above

ant-9

(Constraint 4/7) Consider the bold text in the following paragraph:

> *Since Ant 1.8.0, you may instead use* ***property*** *expansion; a value of true (or on or yes) will enable the item, while* ***false (or off or no)*** *will disable it*

Which contains the constraint:

> *property must be one of {false, off, no}*

And is implemented in the highlighted portion of this code:

```
    if ("off".equalsIgnoreCase(s)
        || "false".equalsIgnoreCase(s)
        || "no".equalsIgnoreCase(s)) {
        return Boolean.FALSE;
    }
```

(If you need to, you can see the full method, or see the full class)

Given the following pattern definitions:

equals-or-chain: Equality expressions (using the == operator) or equals method calls are chained by "or" operators in an expression checking possible values of a variable.

```
value == 1 || value == 2 || value == 3

value.equals("val1") || value.equals("val2") || value.equals("val3")
```

binary-flag-check: An integer value is operated with a bitwise AND operator (&) against an integer variable, and then the result is compared with == or != against another integer value (literal or variable).

```
value & FLAG == 0

obj.value & FLAG == res
```

null-empty-check: A string value is checked for nullity using the == or != operators and immediately after compared to the empty string using the equals method. The two operations are combined using the && or || operators. The operands in each equality may be in any order.

```
value != null && !value.equals("")

null != obj.getValue() && !"".equals(obj.getValue())
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

equals-or-chain

binary-flag-check

null-empty-check

None of the above

argouml-77-false

(Constraint 5/7) Consider the bold text in the following paragraph:

> ***Multiplicity****: Editable drop down selector with checkmark. The* ***default value (1)*** *is that there is one instance of this attribute for each instance of the class, i.e. it is a scalar.*

Which contains the constraint:

> *multiplicity = 1*

And is implemented in the highlighted portion of this code:

```
    if (comboText == null) {
                Model.getCoreHelper().setMultiplicity(getTarget(), "1");
    } else {
                Model.getCoreHelper().setMultiplicity(getTarget(), comboText);
    }
```

(If you need to, you can see the full method, or see the full class)

Given the following pattern definitions:

binary-comparison: Two variables are compared using one of the relational operators (==, !=, >, <, >=, <=). Use of the equals method is considered an operator in this case. Neither of the operands may be the literal 'null'.

```
obj1.valueA <= obj2.valueB

obj.getValueA().equals(valueB)
```

equals-or-chain: Equality expressions (using the == operator) or equals method calls are chained by “or” operators in an expression checking possible values of a variable.

```
value == 1 || value == 2 || value == 3

value.equals("val1") || value.equals("val2") || value.equals("val3")
```

null-check: A value is checked for nullity using the == or != operators.

```
value == null

null != obj.value
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

binary-comparison

equals-or-chain

null-check

None of the above

rh-31

(Constraint 6/7) Consider the bold text in the following paragraph:

*If **the property has the ReadOnly attribute**, return false.*

Which contains the constraint:

property musst be read-only

And is implemented in the highlighted portion of this code:

```
int attr = attributeArray[id - 1];
if ((attr & READONLY) == 0) {
    if (start == obj) {
        if (value == null) {
            value = UniqueTag.NULL_VALUE;
        }
```

(If you need to, you can see the full method, or see the full class)

Given the following pattern definitions:

null-check: A value is checked for nullity using the == or != operators.

```
value == null

null != obj.value
```

binary-flag-check: An integer value is operated with a bitwise AND operator (&) against an integer variable, and then the result is compared with == or != against another integer value (literal or variable).

```
value & FLAG == 0

obj.value & FLAG == res
```

null-empty-check: A string value is checked for nullity using the == or != operators and immediately after compared to the empty string using the equals method. The two operations are combined using the && or || operators. The operands in each equality may be in any order.

```
value != null && !value.equals("")

null != obj.getValue() && !"".equals(obj.getValue())
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

null-check

binary-flag-check

null-empty-check

None of the above

ant-20

(Constraint 7/7) Consider the bold text in the following paragraph:

> *At least **one set of sources** and one set of targets **is required**.*

Which contains the constraint:

> *sets of sources > 0*

And is implemented in the highlighted portion of this code:

```
  if (sources == null) {
    throw new BuildException(
        "At least one set of source resources must be specified");
  }
```

(If you need to, you can see the full method, or see the full class)

Given the following pattern definitions:

null-check: A value is checked for nullity using the == or != operators.

```
value == null

null != obj.value
```

equals-or-chain: Equality expressions (using the == operator) or equals method calls are chained by “or” operators in an expression checking possible values of a variable.

```
value == 1 || value == 2 || value == 3

value.equals("val1") || value.equals("val2") || value.equals("val3")
```

binary-flag-check: An integer value is operated with a bitwise AND operator (&) against an integer variable, and then the result is compared with == or != against another integer value (literal or variable).

```
value & FLAG == 0

obj.value & FLAG == res
```

Which pattern does the highlighted portion of the code above exhibit (if any)?

null-check

equals-or-chain

binary-flag-check

None of the above