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:

```
1    // Read WIN configuration file
2    if (configFile != null) {
3      FileReader fileReader = new FileReader(configFile);
4      BufferedReader reader = new BufferedReader(fileReader);
5      timeZone = reader.readLine();
6      channelInfo.clear();
7      ...
```

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:

```
1    if (projectFileName == null || "".equals(projectFileName)) {
2        if (ProjectManager.getManager().getCurrentProject() != null) {
3            projectFileName = ProjectManager.getManager()
4                .getCurrentProject().getName();
5        }
6    }
```

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 assignmer
```

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:

```
1    // Read WIN configuration file
2    if (configFile != null) {
3      FileReader fileReader = new FileReader(configFile);
4      BufferedReader reader = new BufferedReader(fileReader);
5      timeZone = reader.readLine();
6      channelInfo.clear();
7      ...
```

(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*.

ant-38

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

Property	Description	Required
erroronmissingdir	Specify what happens if the base directory does not exist.	No; **defaults to true**

Which contains the constraint:

erroronmissingdir = true

And is implemented in the highlighted portion of this code:

```
1    private boolean followSymlinks = true;
2    private boolean errorOnMissingDir = true;
3    private int maxLevelsOfSymlinks = DirectoryScanner.MAX_LEVELS_OF_SYMLINKS;
```

(If you need to, you can 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)
```

assign-constant: A literal value is assigned to a variable or field.

```
int val = 5000

public static final String STRING_VAL = "value"
```

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)?

constant-argument

assign-constant

binary-comparison

None of the above

argo-58

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

> *By default a new package has no name defined. The package will appear with the **name (Unnamed Package)** in the explorer.*

Which contains the constraint:

> *name = (Unnamed Package)*

And is implemented in the highlighted portion of this code:

```
1    misc.transitions-of-class = Transitions of Class
2    misc.type = Type
3    misc.unnamed = (Unnamed {0})
4    misc.untitled-model = untitledModel
5    misc.untitled-profile = untitledProfile
```

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

Given the following pattern definitions:

if-chain: A chain of ifs is used like a switch on a variable, checking against its possible values. Each if clause uses the == operator or equals method.

```
if (value == 1) {
    ...
} else if (value == 2) {
    ...
} else if (value == 3) {
    ...
```

properties-file: The default value for a variable is stored in a properties file.

```
value=100

value=true

value=default string
```

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)
```

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

if-chain

properties-file

constant-argument

None of the above

swarm-15

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

X is the number of minutes to display along the bottom of the helicorder. ***Default is 30 minutes****.*

Which contains the constraint:

time chunk = 30 minutes

And is implemented in the highlighted portion of this code:

```
1    useLargeCursor = StringUtils.stringToBoolean(config.getString("useLargeCurs
2
3    span = StringUtils.stringToInt(config.getString("span"), 24);
4    timeChunk = StringUtils.stringToInt(config.getString("timeChunk"), 30);
5
6    lastPath = StringUtils.stringToString(config.getString("lastPath"), "defaul
```

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

Given the following pattern definitions:
properties-file: The default value for a variable is stored in a properties file.

```
value=100

value=true

value=default string
```

if-chain: A chain of ifs is used like a switch on a variable, checking against its possible values. Each if clause uses the == operator or equals method.

```
if (value == 1) {
    ...
} else if (value == 2) {
    ...
} else if (value == 3) {
    ...
```

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)
```

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

properties-file

if-chain

constant-argument

None of the above

jedit-17-false

(Constraint 4/7) 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:

```
1    VFS vfs = VFSManager.getVFSForPath(getPath());
2    if (((vfs.getCapabilities() & VFS.WRITE_CAP) == 0) ||
3        !vfs.isMarkersFileSupported())
4    {
5     VFSManager.error(view, path, "vfs.not-supported.save",
6      new String[] { "markers file" });
```

```
7      return false;
8    }
```

(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

rhino-3

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

> *Because a single-line comment can contain **any character except a LineTerminator character**, and because of the general rule that a token is always as long as possible, a single-line comment always consists of all characters from the // marker to the end of the line.*

Which contains the constraint:

character must not be line terminator

And is implemented in the highlighted portion of this code:

```
1    private void skipLine() throws IOException
2    {
3        // skip to end of line
4        int c;
5        while ((c = getChar()) != EOF_CHAR && c != '\n') { }
6        ungetChar(c);
7    }
```

(If you need to, you can 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)
```

assign-constant: A literal value is assigned to a variable or field.

```
int val = 5000

public static final String STRING_VAL = "value"
```

properties-file: The default value for a variable is stored in a properties file.

```
value=100

value=true

value=default string
```

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

binary-comparison

assign-constant

properties-file

None of the above

swarm-52

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

> *When doing a P or S pick, users must traverse all the way down the menu tree to determine* ***onset (Emergent, Impulsive, or Questionable)****, polarity, and weight (0 to 4) of the pick.*

Which contains the constraint:

> *onset must be one of {Emergent, Impulsive, Questionable}*

And is implemented in the highlighted portion of this code:

```
 1  /**
 2     * Parse an Onset from a String.
 3     *
 4     * @param string onset
 5     * @return onset object
 6     * @throws ParseException when things go wrong
 7     */
 8    public static Onset parse(String string) throws ParseException {
 9      if ("emergent".equals(string)) {
10        return EMERGENT;
11      } else if ("impulsive".equals(string)) {
12        return IMPULSIVE;
13      } else if ("questionable".equals(string)) {
14        return QUESTIONABLE;
15      } else {
16        throw new ParseException("Cannot parse " + string, 12);
17      }
18
19    }
20  }
```

(If you need to, you can 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
```

if-chain: A chain of ifs is used like a switch on a variable, checking against its possible values. Each if clause uses the == operator or equals method.

```
if (value == 1) {
    ...
} else if (value == 2) {
```

```
    ...
} else if (value == 3) {
    ...
```

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-check

if-chain

binary-comparison

None of the above

argouml-77-false

(Constraint 7/7) 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:

```
1    if (comboText == null) {
2                Model.getCoreHelper().setMultiplicity(getTarget(), "1");
3    } else {
4                Model.getCoreHelper().setMultiplicity(getTarget(), comboText);
5    }
```

(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