

## 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](mailto: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 <b>package weight</b> is greater than 20 kg...	package weight > 20
The options for <b>marital status</b> are: <b>single, married, or, divorced</b> ...	marital status must be one of {single, married, divorced}
If the <b>agent status</b> is set to <b>available</b> at that time...	agent status must be "available"
The <b>minimum frequency</b> value is set to <b>20</b> 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  private static boolean nullOrEmpty(Object value) {
2      return value == null || "".equals(value);
3  }
```

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 assignment
```

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*.

**argouml-1**

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

*The current filename. If **no filename for the project is set** yet, then the titlebar shows "Untitled"*

Which contains the constraint:

*filename is not set*

And is implemented in the highlighted portion of this code:

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

(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())
```

**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-empty-check

equals-or-chain

binary-flag-check

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:

```
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

### Ant-5

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

*The optional description attribute can be used to provide a one-line description of this target, which is printed by the -projecthelp command-line option. **Targets without such a description** are deemed internal and will not be listed, unless either the -verbose or -debug option is used.*

Which contains the constraint:

*target does not have description*

And is implemented in the highlighted portion of this code:

```
1  final String targetDescription = currentTarget.getDescription();
2  // maintain a sorted list of targets
3  if (targetDescription == null) {
4      final int pos = findTargetPosition(subNames, targetName);
5      subNames.insertElementAt(targetName, pos);
```

(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-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)?

equals-or-chain

binary-flag-check

null-check

None of the above

rhino-1

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

The following characters are considered to be **white space**:

Code Point Value	Name	Formal Name
<b>\u0009</b>	Tab	<TAB>
<b>\u000B</b>	Vertical Tab	<VT>
<b>\u000C</b>	Form Feed	<FF>
<b>\u0020</b>	Space	<SP>

Which contains the constraint:

*character must be in {\u0009, \u000B, \u000C, \u0020}*

And is implemented in the highlighted portion of this code:

```
1  if (c <= 127) {
2      return c == 0x20 || c == 0x9 || c == 0xC || c == 0xB;
3  } else {
4      return c == 0xA0
```



```
5 || Character.getType((char)c) == Character.SPACE_SEPARATOR;  
6 }
```

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

Given the following pattern definitions:

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

```
value // Boolean variable  
  
obj.getValue() // Boolean return  
  
obj.value // Boolean field
```

**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
```

**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")
```

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

boolean-property

binary-flag-check

equals-or-chain

None of the above

### jEdit-7

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

*Different emblems are displayed next to buffer names in the list, depending the buffer's state; a red disk is shown for buffers with unsaved changes, a lock is shown for read-only buffers, and a spark is shown for **new buffers which don't yet exist on disk**.*

Which contains the constraint:

*buffer is new*

And is implemented in the highlighted portion of this code:

```
1  //{{{ getFlag() method
2  private boolean getFlag(int flag)
3  {
4      int mask = 1 << flag;
5      return (flags & mask) == mask;
6  } //}}}
```

(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
```

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

```
value // Boolean variable
obj.getValue() // Boolean return
obj.value // Boolean field
```

**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

boolean-property

binary-flag-check

None of the above

**Ant-37**

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

Attribute	Description	Required
...	...	...
<b>erroronmissingdir</b>	Specify what happens if the base directory does not exist. <b>If true</b> a build error will happen, if false, the dirset will be ignored/empty. Since Apache Ant 1.7.1	No [...]

Which contains the constraint:

*erroronmissingdir is true*

And is implemented in the highlighted portion of this code:

```

1    if (!dir.exists() && errorOnMissingDir) {
2        throw new BuildException(dir.getAbsolutePath()
3                                   + DirectoryScanner
4                                   .DOES_NOT_EXIST_POSTFIX);
5    }

```

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

Given the following pattern definitions:

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

```

value // Boolean variable

obj.getValue() // Boolean return

obj.value // Boolean field

```

**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)?

- boolean-property
- equals-or-chain
- null-check
- 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

Powered by Qualtrics