CODING

Pattern Matching for instanceof in Java 14

Use pattern matching for instanceof to simplify the use of the instanceof operator in Java, thereby making your code safer and easier to write.

by Mala Gupta

May 18, 2020

Download a PDF of this article

Many Java developers use the `instanceof` operator to compare a reference variable to a type. If the result is `true`, the next obvious step is to explicitly cast it to the type they compared it with, to access its members. These steps have a repetition:

`compareTo`**`Type`** – `ifTrue` – `castTo`**`Type`**.

Here's an example of code that can be commonly found in codebases:

```
1. void outputValueInUppercase(Object obj) {
2.     if (obj instanceof String) {
3.       String str = (String) obj;
4.       System.out.println(str.toUpperCase()
5.     }
6. }
```

In the preceding code, the code in line 2 compares the reference variable `obj` to the type `String`. If the result is `true`, the code in line 3 defines a local variable `str`, explicitly casts `obj` to the type `String`, and assigns it to the variable `str`. The code in line 4 can access members of the `String` value referred to by `str`.

The following code shows how pattern matching for `instanceof` removes this redundant code by introducing a

pattern variable `str` with the `instanceof` operator (right after the type `String`):

```
1. void outputValueInUppercase(Object obj) {
2.     if (obj instanceof String str) {
3.         System.out.println(str.toUpperCase
4.     }
5. }
```

In the preceding code, if the `instanceof` condition is `true`, the pattern variable `str` binds to the instance referred to by the variable `obj`. This saves you from having to either define a new variable or explicitly cast it to `String` before you call the method `toUpperCase()` on it.

Pattern Variables

Pattern variables are `final` local variables that are declared *and* initialized at the same place. With other final local variables, it is possible to declare them and defer their assignment. Also, you cannot assign another value to a pattern variable since it is implicitly `final`.

The scope of the pattern variable is limited. If you try to access it in an `else` block, you'll receive an error.

This might seem confusing. In the following code, if the class `PatternMatching` defines an instance or static variable with the same name as the pattern variable (`s`), the code will compile. In this case, `s` in the `else` block would not refer to the pattern variable introduced in the `if` block:

```
public class PatternMatching {
    private String s = "initial value";
    void outputValueInUppercase(Object obj) {
        if (obj instanceof String s) {
            System.out.println(s.toUpperCase(
        } else {
            System.out.println(s.toLowerCase(
        }
    }
}
```

Simplifying the equals() Method

The simplicity of pattern matching can be deceptive. Here is an example of how developers usually override the `equals()` method in a class. In the following code, the class `Monitor` defines two fields—`model` (a `String` value) and `price` (a `double` value):

```
public class Monitor {
   String model;
```

```
    double price;

    @Override
    public boolean equals(Object o) {
        if (o instanceof Monitor) {
            Monitor other = (Monitor) o;
            if (model.equals(other.model) && p
                return true;
            }
        }
        return false;
    }
}
```

The following code shows how the preceding `equals()` method could be simplified by using pattern matching for `instanceof` and the further simplification of `if` statements:

```
public class Monitor {
    String model;
    double price;

    @Override
    public boolean equals(Object o) {
        return o instanceof Monitor other &&
            model.equals(other.model) &&
            price == other.price;
    }
}
```

Concise and Readable Code

Pattern matching with `instanceof` can be used at multiple places to simplify your code. Look at the method `isFeasible` in the following code:

```
class Project {
    Lang lang;
    Emp projManager;

    private boolean isFeasible(Project projec
        if (project.getLang() != Lang.PASCAL)
            return false;
        }
        if (!(project.getProjManager() instan
            return false;
        }
        return ceo.availableAt(location);
    }

    public Emp getProjManager() {
        return projManager;
    }
    public void setProjManager(Emp projManage
        this.projManager = projManager;
    }
    public Lang getLang() {
        return lang;
    }
```

```
    public void setLang(Lang lang) {
        this.lang = lang;
    }
}
Replace from here to the end of the code list

enum Lang {JAVA, PASCAL}
class Emp { }
class Location { }
class CEO extends Emp {
    Location loc;

    boolean availableAt(Location location) {
        return loc.equals(location);
    }
}
```

The following code shows how you can simplify the method `isFeasible` by using pattern matching with `instanceof`, which removes redundant casting and then simplifies its `if` statements:

```
1.    private boolean isFeasible(Project proj
2.        return project.getLang() == Lang.PA
3.            project.getProjManager() instan
4.            ceo.availableAt(location);
5.    }
```

In the preceding code, pattern matching with `instanceof` is used in line 3.

Using Pattern Matching for instanceof with the Stream API

Introduction of the pattern variable opens up various possibilities for improvements. Here is the definition of a method named `process`:

```
void process(Font font, int size) {
    final ArrayList<Node> list = modules.getC
    for (Iterator<Node> i = list.iterator();
        final Object o = i.next();
        if (o instanceof LetterNode) {
            final LetterNode letterNode = (Le
            if (letterNode.isLatin()) {
                if (!isLetterTrueFont(letterN
                    i.remove();
                }
            }
        }
    }
}
```

The following code shows how you can reduce the preceding code by passing code that uses pattern matching for `instanceof` with the Stream API:

```
void process(Font font, int size) {
    modules.getChildren().removeIf(o -> o ins
                                && letterNode.
                                && !isLetterTr
                font, size));
}
```

Generics and Multiple Uses of instanceof in a Code Block

Pattern matching for `instanceof` works with generics too.

To look for places where you can use pattern matching for `instanceof`, search for uses of the `instanceof` operator and explicit casting of variables. For instance, the following code has multiple occurrences of the `instanceof` operator with explicit casting:

```
void processChildNode(Tree tree) {
    if (tree.getChildNodes() instanceof Map)
        Map<?, Node> childNodes = (Map<?, Nod
        if (childNodes.size() == 1) {
            Node = childNodes.get("root");
            if (node instanceof LetterNode) {
                LetterNode = (LetterNode) nod
                System.out.println(letterNode
                }
            }
        }
    }
```

The preceding code block can be simplified to the following:

```
void processChildNode(Tree tree) {
    if (tree.getChildNodes() instanceof Map<?
                && childNodes.size() == 1
                && childNodes.get("root") in
        System.out.println(letterNode.isLatin
    }
}
```

If you are wondering about the unchecked cast in the preceding example, I'd like to share that method `getChildNodes()` returns a value of type `Map<String, Node>`. It is okay to cast from `Map<String, Node>` to `Map<?, Node>` in the preceding example since it is an upcast.

Pattern Matching with instanceof Is a Preview Language Feature

Pattern matching with `instanceof` has been released as a preview language feature in Java 14 under JEP 305. Being a preview feature essentially means that this feature is ready to be

used by developers, although its finer details could change in a future Java release depending on developer feedback.

With Java's new release cadence of six months, new language features are released as preview features. They are complete but not permanent. Unlike an API, language features cannot be deprecated in the future. So, if you have any feedback on pattern matching with `instanceof`, share it on the JDK mailing list.

To use preview language features, you must enable them when you compile and execute code that uses them. This ensures you do not use these features by mistake.

To compile a source file with pattern matching for `instanceof`, you must use the options `-enable-preview` and `-release 14`. Here is an example to compile a source file called `Java14.java` using the command line:

```
javac --enable-preview --release 14 Java14.ja
```

To reinforce that preview features are subject to change, you will get compiler warnings such as the one shown in **Figure 1** when you execute the preceding command:

```
C:\Users\Mala Gupta>javac --enable-preview --release 14 Java14.java

Note: Java14.java uses preview language features.
Note: Recompile with -Xlint:preview for details.
```

Figure 1. Compiler warning for code that uses preview language features

To execute the class `Java14`, you must use the option `–enable-preview`:

```
java --enable-preview Java14
```

Conclusion

A preview language feature in Java 14, pattern matching for `instanceof`, can simplify how you read and write your code every day. By adding a pattern variable to the `instanceof` operator, this feature makes your code concise and easier to read and write. In a future Java version, you might see its use extended to `switch` constructs and other statements.

Mala Gupta

Mala Gupta (@eMalaGupta) is a Java Champion and developer advocate at JetBrains. She is also the founder at eJavaGuru.com and an author of popular certification books. She co-leads the Delhi

Java User Group and is a director of the Delhi chapter of Women Who Code.

Share this Page