JEP 443: Unnamed Patterns and Variables (Preview)

Owner	Angelos Bimpoudis
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	specification / language
Discussion	amber dash dev at openjdk dot org
Effort	S
Duration	S
Relates to	JEP 456: Unnamed Variables & Patterns
Reviewed by	Alex Buckley
Endorsed by	Brian Goetz
Created	2022/09/26 08:00
Updated	2023/12/12 19:23
Issue	8294349

Summary

Enhance the Java language with *unnamed patterns*, which match a record component without stating the component's name or type, and *unnamed variables*, which can be initialized but not used. Both are denoted by an underscore character, _. This is a preview language feature.

Goals

- Improve the readability of record patterns by eliding unnecessary nested patterns.
- Improve the maintainability of all code by identifying variables that must be declared (e.g., in a `catch` clause) but will not be used.

Non-Goals

- It is not a goal to allow unnamed fields or method parameters.
- It is not a goal to alter the semantics of local variables, e.g., in definite assignment analysis.

Motivation

Unused patterns

Records (JEP 395) and record patterns (JEP 440) work together to streamline data processing. A record class aggregates the components of a data item into an instance, while code that receives an instance of a record class uses pattern matching with record patterns to disaggregate the instance into its components. For example:

```
record Point(int x, int y) { }
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) { }

... new ColoredPoint(new Point(3,4), Color.GREEN) ...

if (r instanceof ColoredPoint(Point p, Color c)) {
    ... p.x() ... p.y() ...
}
```

In this code, one part of the program creates a `ColoredPoint` instance while another part uses pattern matching with `instanceof` to test whether a variable is a `ColoredPoint` and, if so, extract its two components.

Record patterns such as `ColoredPoint(Point p, Color c)` are pleasingly descriptive, but it is common for programs to need only some of the components for further processing. For example, the code above needs only `p` in the `if` block, not `c`. It is laborious to write out all the components of a record class every time we do such pattern matching. Furthermore, it is not visually clear that the `Color` component is irrelevant; this makes the condition in the `if` block harder to read, too. This is especially evident when record patterns are nested to extract data within components, such as:

```
if (r instanceof ColoredPoint(Point(int x, int y), Color c)) {
    ... x ... y ...
}
```

We can use `var` to reduce the visual cost of the unnecessary component `Color c`, e.g., `ColoredPoint(Point(int x, int y), var c)`, but it would better to reduce the cost even further by omitting unnecessary components altogether. This would both simplify the task of writing record patterns and improve readability, by removing clutter from the code.

As developers gain experience with the data-oriented methodology of record classes and their companion mechanism, sealed classes (JEP 409), we expect that pattern matching over complex data structures will become commonplace. Frequently, the shape of a structure will be just as important as the individual data items within it. As a highly simplified example, consider the following `Ball` and `Box` classes, and a `switch` that explores the content of a `Box`:

```
sealed abstract class Ball permits RedBall, BlueBall, GreenBall { }
final  class RedBall   extends Ball { }
final  class BlueBall  extends Ball { }
final  class GreenBall extends Ball { }

record Box<T extends Ball>(T content) { }

Box<? extends Ball> b = ...
switch (b) {
    case Box(RedBall   red)   -> processBox(b);
    case Box(BlueBall  blue)  -> processBox(b);
    case Box(GreenBall green) -> stopProcessing();
}
```

Each `case` deals with a `Box` based on its content, but the variables `red`, `blue`, and `green` are not used. Since the variables are not used, this code would be more readable if we could elide their names.

Furthermore, if the `switch` were refactored to group the first two patterns in one `case` label:

```
case Box(RedBall red), Box(BlueBall blue) -> processBox(b);
```

then it would be erroneous to name the components: Neither of the names is usable on the right-hand side because either of the patterns on the left-hand side can match. Since the names are unusable it would be better if we could elide them.

Unused variables

Turning to traditional imperative code, most developers have encountered the situation of having to declare a variable that they did not intend to use. This typically occurs when the side effect of a statement is more important than its result. For example, the following code calculates `total` as the side effect of a loop, without using the loop variable `order`:

```
int total = 0;
for (Order order : orders) {
    if (total < LIMIT) {
        ... total++ ...
    }
}
```

The prominence of `order`'s declaration is unfortunate, given that `order` is not used. The declaration can be shortened to `var order`, but there is no way to avoid giving this variable a name. The name itself can be shortened to, e.g., `o`, but this syntactic trick does not communicate the semantic intent that the variable will go unused. In addition, static analysis tools typically complain about unused variables, even when the developer intends non-use and may not have a way to silence the warnings.

Here is an example where the side effect of an expression is more important than its result, leading to an unused variable. The following code dequeues data but only needs two out of every three elements:

```
Queue<Integer> q = ... // x1, y1, z1, x2, y2, z2 ..
while (q.size() >= 3) {
   int x = q.remove();
   int y = q.remove();
   int z = q.remove(); // z is unused
    ... new Point(x, y) ...
}
```

The third call to `remove()` has the desired side effect — dequeuing an element — regardless of whether its result is assigned to a variable, so the declaration of `z` could be elided. However, for maintainability, the developer may wish to consistently denote the result of `remove()` by declaring a variable, even if it is not currently used, and even if it leads to static analysis warnings. Unfortunately, in many programs, the choice of variable name will not come so easily as `z` in the code above.

Unused variables occur frequently in two other kinds of statements that focus on side effects:

- The `try`-with-resources statement is always used for its side effect, namely the automatic closing of resources. In some cases a resource represents a context in which the code of the `try` block executes; the code does not use the context directly, so the name of the resource variable is irrelevant. For example, assuming a `ScopedContext` resource that is `AutoCloseable`, the following code acquires and automatically releases a context:

  ```
  try (var acquiredContext = ScopedContext.acquire()) {
      ... acquiredContext not used ...
  }
  ```

  The name `acquiredContext` is merely clutter, so it would be nice to elide it.

- Exceptions are the ultimate side effect, and handling one often gives rise to an unused variable. For example, most Java developers have written `catch` blocks of this form, where the name of the exception parameter is irrelevant:

  ```
  String s = ...;
  try {
      int i = Integer.parseInt(s);
      ... i ...
  } catch (NumberFormatException ex) {
      System.out.println("Bad number: " + s);
  }
  ```

Even code without side effects must sometimes declare unused variables. For example:

```
...stream.collect(Collectors.toMap(String::toUpperCase,
                                   v -> "NODATA"));
```

This code generates a map which maps each key to the same placeholder value. Since the lambda parameter v is not used, its name is irrelevant.

In all these scenarios where variables are unused and their names are irrelevant, it would be better if we could simply declare variables with no name. This would free maintainers from having to understand irrelevant names, and would avoid false positives on non-use from static analysis tools.

The kinds of variables that can reasonably be declared with no name are those which have no visibility outside a method: local variables, exception parameters, and lambda parameters, as shown above. These kinds of variables can be renamed or made unnamed without external impact. In contrast, fields — even if they are `private` — communicate the state of an object across methods, and unnamed state is neither helpful nor maintainable.

Description

The *unnamed pattern* is denoted by an underscore character _ (U+005F). It allows the type and name of a record component to be elided in pattern matching; e.g.,

- `... instanceof Point(int x, _)`
- `case Point(int x, _)`

An *unnamed pattern variable* is declared when the pattern variable in a type pattern is denoted by an underscore. It allows the identifier which follows the type or `var` in a type pattern to be elided; e.g.,

- `... instanceof Point(int x, int _)`
- `case Point(int x, int _)`

An *unnamed variable* is declared when either the local variable in a local variable declaration statement, or an exception parameter in a `catch` clause, or a lambda parameter in a lambda expression is denoted by an underscore. It allows the identifier which follows the type or `var` in the statement or expression to be elided; e.g.,

- `int _ = q.remove();`
- `... } catch (NumberFormatException _) { ...`
- `(int x, int _) -> x + x`

In the case of single-parameter lambda expressions, such as `_ -> "NODATA"`, the unnamed variable that serves as the parameter should not be confused with an unnamed pattern.

A single underscore is the lightest reasonable syntax for signifying the absence of a name. Since it was valid as an identifier in Java 1.0, we initiated a long-term process in 2014 to reclaim it for unnamed patterns and variables. We started issuing compile-time warnings when underscore was used as an identifier in Java 8 (2014) and we turned those warnings into errors in Java 9 (2017, JEP 213). Many other languages, such as Scala and Python, use underscore to declare a variable with no name.

The ability to use underscore in identifiers of length two or more is unchanged, since underscore remains a Java letter and a Java letter-or-digit. For example, identifiers such as _age and MAX_AGE and __ (two underscores) continue to be legal.

The ability to use underscore as a digit separator is unchanged. For example, numeric literals such as 123_456_789 and `0b1010_0101` continue to be legal.

The unnamed pattern

The unnamed pattern is an unconditional pattern which binds nothing. It may be used in a nested position in place of a type pattern or a record pattern. For example,

- `... instanceof Point(_, int y)`

is legal, but these are not:

- `r instanceof _`
- `r instanceof _(int x, int y)`

Consequently, the earlier example can omit the type pattern for the `Color` component entirely:

```
if (r instanceof ColoredPoint(Point(int x, int y), _)) { ... x ... y ... }
```

Likewise, we can extract the `Color` component while eliding the record pattern for the `Point` component:

```
if (r instanceof ColoredPoint(_, Color c)) { ... c ... }
```

In deeply nested positions, using the unnamed pattern improves the readability of code that does complex data extraction. For example:

```
if (r instanceof ColoredPoint(Point(int x, _), _)) { ... x ... }
```

This code extracts the x coordinate of the nested `Point` while omitting both the y and `Color` components.

Unnamed pattern variables

An unnamed pattern variable can appear in any type pattern, whether the type pattern appears at the top level or is nested in a record pattern. For example, both of these appearances are legal:

- `r instanceof Point _`
- `r instanceof ColoredPoint(Point(int x, int _), Color _)`

By allowing us to elide names, unnamed pattern variables make run-time data exploration based on type patterns visually clearer, especially when used in `switch` statements and expressions.

Unnamed pattern variables are particularly helpful when a `switch` executes the same action for multiple cases. For example, the earlier Box and `Ball` code can be rewritten as:

```
switch (b) {
    case Box(RedBall _), Box(BlueBall _) -> processBox(b);
    case Box(GreenBall _)                -> stopProcessing();
    case Box(_)                          -> pickAnotherBox();
}
```

The first two cases use unnamed pattern variables because their right-hand sides do not use the Box's component. The third case, which is new, uses the unnamed pattern in order to match a Box with a `null` component.

A `case` label with multiple patterns can have a guard. A guard governs the case as a whole, rather than the individual patterns. For example, assuming that there is an `int` variable x, the first case of the previous example could be further constrained:

```
case Box(RedBall _), Box(BlueBall _) when x == 42 -> processBox(b);
```

Pairing a guard with each pattern is not allowed, so this is prohibited:

```
case Box(RedBall _) when x == 0, Box(BlueBall _) when x == 42 -> processBox(b);
```

The unnamed pattern is shorthand for the type pattern `var _`. Neither the unnamed pattern nor `var _` may be used at the top level of a pattern, so all of these are prohibited:

- `... instanceof _`
- `... instanceof var _`
- `case _`
- `case var _`

Unnamed variables

The following kinds of declarations can introduce either a named variable (denoted by an identifier) or an unnamed variable (denoted by an underscore):

- A local variable declaration statement in a block (JLS 14.4.2),
- A resource specification of a `try`-with-resources statement (JLS 14.20.3),
- The header of a basic `for` statement (JLS 14.14.1),
- The header of an enhanced `for` loop (JLS 14.14.2),
- An exception parameter of a `catch` block (JLS 14.20), and
- A formal parameter of a lambda expression (JLS 15.27.1).

(The possibility of an unnamed local variable being declared by a pattern, i.e., a pattern variable (JLS 14.30.1), was covered above.)

Declaring an unnamed variable does not place a name in scope, so the variable cannot be written or read after it has been initialized. An initializer must be provided for an unnamed variable in each kind of declaration above.

An unnamed variable never shadows any other variable, since it has no name, so multiple unnamed variables can be declared in the same block.

Here are the examples from above, modified to use unnamed variables.

- An enhanced `for` loop with side effects:

  ```
  int acc = 0;
  for (Order _ : orders) {
      if (acc < LIMIT) {
          ... acc++ ...
      }
  }
  ```

  The initialization of a basic `for` loop can also declare unnamed local variables:

  ```
  for (int i = 0, _ = sideEffect(); i < 10; i++) { ... i ... }
  ```

- An assignment statement, where the result of the expression on the right hand side is not needed:

  ```
  Queue<Integer> q = ... // x1, y1, z1, x2, y2, z2, ...
  while (q.size() >= 3) {
     var x = q.remove();
     var y = q.remove();
     var _ = q.remove();
     ... new Point(x, y) ...
  }
  ```

  If the program needed to process only the x1, x2, etc., coordinates then unnamed variables could be used in multiple assignment statements:

  ```
  while (q.size() >= 3) {
      var x = q.remove();
      var _ = q.remove();
      var _ = q.remove();
      ... new Point(x, 0) ...
  }
  ```

- A `catch` block:

  ```
  String s = ...
  try {
      int i = Integer.parseInt(s);
      ... i ...
  } catch (NumberFormatException _) {
      System.out.println("Bad number: " + s);
  }
  ```

  Unnamed variables can be used in multiple `catch` blocks:

  ```
  try { ... }
  catch (Exception _) { ... }
  catch (Throwable _) { ... }
  ```

- In `try`-with-resources:

  ```
  try (var _ = ScopedContext.acquire()) {
      ... no use of acquired resource ...
  }
  ```

- A lambda whose parameter is irrelevant:

  ```
  ...stream.collect(Collectors.toMap(String::toUpperCase, _ -> "NODATA"))
  ```

Risks and Assumptions

- We assume that very little existing and maintained code uses underscore as a variable name. Such code was almost certainly written for Java 7 or earlier and cannot have been recompiled with Java 9 or later. The risk to such code is a compile-time error when reading or writing a variable called _ and when declaring any other kind of entity (class, field, etc.) with the name _. We assume that developers can modify such code to avoid using underscore as the name of a variable or any other kind of entity by, e.g., renaming _ to _1.
- We expect developers of static analysis tools to realize the new role of underscore for unnamed variables and to avoid flagging the non-use of such variables in modern code.

Alternatives

- It is possible to define an analogous concept of *unnamed method parameters*. However, this has some interactions with specification (e.g., how do you write JavaDoc for unnamed parameters?) and overriding (e.g., what does it mean to override a method with unnamed parameters?). We will not pursue it in this JEP.
- JEP 302 (Lambda Leftovers) examined the issue of unused lambda parameters and identified the role of underscore to denote them, but also covered many other issues which were handled better in other ways. This JEP addresses the use of unused lambda parameters explored in JEP 302 but does not address the other issues explored there.