JEP 456: Unnamed Variables & Patterns

Owner	Angelos Bimpoudis
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	22
Component	specification / language
Discussion	amber dash dev at openjdk dot org
Effort	S
Duration	S
Relates to	JEP 443: Unnamed Patterns and Variables (Preview)
Reviewed by	Brian Goetz
Endorsed by	Brian Goetz
Created	2023/07/10 16:17
Updated	2024/01/04 20:25
Issue	8311828

Summary

Enhance the Java programming language with unnamed variables and unnamed patterns, which can be used when variable declarations or nested patterns are required but never used. Both are denoted by the underscore character, _.

History

Unnamed variables and unnamed patterns first previewed in JDK 21 via JEP 443, which was titled *Unnamed Patterns and Variables*. We here propose to finalize this feature without change.

Goals

- Capture developer intent that a given binding or lambda parameter is unused, and enforce that property, so as to clarify programs and reduce opportunities for error.
- Improve the maintainability of all code by identifying variables that must be declared (e.g., in `catch` clauses) but are not used.
- Allow multiple patterns to appear in a single `case` label, provided that none of them declare any pattern variables.
- Improve the readability of record patterns by eliding unnecessary nested type patterns.

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 in, e.g., definite assignment analysis.

Motivation

Developers sometimes declare variables that they do not intend to use, whether as a matter of code style or because the language requires variable declarations in certain contexts. The intent of non-use is known at the time the code is written, but if it is not captured explicitly then later maintainers might accidentally use the variable, thereby violating the intent. If we could make it impossible to accidentally use such variables then code would be more informative, more readable, and less prone to error.

Unused variables

The need to declare a variable that is never used is especially common in code whose side-effect is more important than its result. For example, this code calculates `total` as the side effect of a loop, without using the loop variable `order`:

```
static int count(Iterable<Order> orders) {
    int total = 0;
    for (Order order : orders)    // order is unused
        total++;
    return total;
}
```

The prominence of the declaration of `order` 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 intent that the variable never be used. 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.

For another example where the side effect of an expression is more important than its result, the following code dequeues data but needs only 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 author of this code may wish to consistently denote the result of `remove()` by declaring a variable. They currently have two options, both unpleasant:

- Do not declare the variable `z`, which leads to an asymmetry and possibly a static-analysis warning about ignoring a return value, or else
- Declare a variable that is not used and possibly get a static-analysis warning about an unused variable.

Unused variables occur frequently in two other 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 just 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 developers have written `catch` blocks of this form, where the exception parameter ex is unused:

  ```
  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.

Unused pattern variables

Local variables can also be declared by type patterns — such local variables are known as *pattern variables* — and so type patterns can also declare variables that are unused. Consider the following code, which uses type patterns in the `case` labels of a `switch` statement that switches over an instance of a `sealed` class `Ball`:

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

Ball ball = ...
switch (ball) {
    case RedBall   red   -> process(ball);
    case BlueBall  blue  -> process(ball);
    case GreenBall green -> stopProcessing();
}
```

The cases of the `switch` examine the type of the `Ball` using type patterns, but the pattern variables `red`, `blue`, and `green` are not used on the right-hand sides of the `case` clauses. This code would be clearer if we could elide these variable names.

Now suppose that we define a record class Box which can hold any type of `Ball`, but might also hold the `null` value:

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

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

The nested type patterns still declare pattern variables that are not used. Since this `switch` is more involved than the previous one, eliding the names of the unused variables in the nested type patterns would even further improve readability.

Unused nested patterns

We can nest records within records, leading to situations in which the shape of a data structure is as important as the data items within it. 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 a pattern `instanceof` to test whether a variable is a `ColoredPoint` and, if so, extract its two component values.

Record patterns such as `ColoredPoint(Point p, Color c)` are pleasingly descriptive, but it is common for programs to use only some of the component values for further processing. For example, the code above uses only `p` in the `if` block, not `c`. It is laborious to write out type patterns for all the components of a record class every time we do such pattern matching. Furthermore, it is not visually clear that the entire `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, as in:

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

We could use an unnamed pattern variable to reduce the visual cost, e.g. `ColoredPoint(Point(int x, int y), Color _)`, but the presence of the `Color` type in the type pattern is distracting. We could remove that by using `var`, e.g. `ColoredPoint(Point(int x, int y), var _)`, but the nested type pattern `var _` still has excessive weight. It would better to reduce the visual 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.

Description

An *unnamed variable* is declared by using an underscore character, _ (U+005F), to stand in for the name of 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.

An *unnamed pattern variable* is declared by using an underscore character to stand in for the pattern variable in a type pattern.

The *unnamed pattern* is denoted by an underscore character and is equivalent to the unnamed type pattern `var _`. It allows both the type and name of a record component to be elided in pattern matching.

A single underscore character is the lightest reasonable syntax for signifying the absence of a name. It is commonly used in other languages, such as Scala and Python, for this purpose. A single underscore was, originally, a valid identifier in Java 1.0, but we later reclaimed it for unnamed variables and patterns: We started issuing compile-time warnings when underscore was used as an identifier in Java 8 (2014), and we removed such identifiers from the language specification, thereby turning those warnings into errors, in Java 9 (2017, JEP 213).

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 also unchanged. For example, numeric literals such as `123_456_789` and `0b1010_0101` continue to be legal.

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),
- The resource specification of a `try`-with-resources statement (JLS §14.20.3),
- The header of a basic `for` loop (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).

Declaring an unnamed variable does not place a name in scope, so the variable cannot be written or read after it is initialized. An initializer must be provided for an unnamed variable declared in a local variable declaration statement or in the resource specification of a `try`-with-resources statement.

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 given above, rewritten to use unnamed variables.

- An enhanced `for` loop with side effects:

  ```
  static int count(Iterable<Order> orders) {
      int total = 0;
      for (Order _ : orders)    // Unnamed variable
          total++;
      return total;
  }
  ```

  The initialization of a simple `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();        // Unnamed variable
     ... 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();       // Unnamed variable
      var _ = q.remove();       // Unnamed variable
      ... new Point(x, 0) ...
  }
  ```

- A `catch` block:

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

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

  ```
  try { ... }
  catch (Exception _) { ... }                // Unnamed variable
  catch (Throwable _) { ... }                // Unnamed variable
  ```

- In `try`-with-resources:

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

- A lambda whose parameter is irrelevant:

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

Unnamed pattern variables

An unnamed pattern variable can appear in a type pattern (JLS §14.30.1), including `var` type patterns, regardless of whether the type pattern appears at the top level or is nested in a record pattern. For example, the `Ball` example can now be written:

```
switch (ball) {
    case RedBall _   -> process(ball);          // Unnamed pattern variable
    case BlueBall _  -> process(ball);          // Unnamed pattern variable
    case GreenBall _ -> stopProcessing();       // Unnamed pattern variable
}
```

and the `Box` and `Ball` example:

```
switch (box) {
    case Box(RedBall _)   -> processBox(box);   // Unnamed pattern variable
    case Box(BlueBall _)  -> processBox(box);   // Unnamed pattern variable
    case Box(GreenBall _) -> stopProcessing();  // Unnamed pattern variable
    case Box(var _)       -> pickAnotherBox();  // Unnamed pattern variable
}
```

By allowing us to elide names, unnamed pattern variables make run-time data exploration based on type patterns visually clearer, both in `switch` blocks and with the `instanceof` operator.

Multiple patterns in case labels

Currently, `case` labels are restricted to contain at most one pattern. With the introduction of unnamed pattern variables and unnamed patterns, it is more likely that we will have within a single `switch` block several `case` clauses with different patterns but the same right-hand side. For example, in the `Box` and `Ball` example the first two clauses have the same right-hand side but different patterns:

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

We could simplify matters by allowing the first two patterns to appear in the same `case` label:

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

We therefore revise the grammar for switch labels (JLS §14.11.1) to

```
SwitchLabel:
    case CaseConstant {, CaseConstant}
    case null [, default]
    case CasePattern {, CasePattern } [Guard]
    default
```

and define the semantics of a `case` label with multiple patterns as matching a value if the value matches any of the patterns.

If a `case` label has multiple patterns then it is a compile-time error for any of the patterns to declare any pattern variables.

A `case` label with multiple `case` patterns can have a guard. The 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);
```

Guards are properties of `case` labels, not individual patterns within a `case` label, so writing more than one guard is prohibited:

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

The unnamed pattern

The unnamed pattern is an unconditional pattern that matches anything but declares and initializes nothing. Like the unnamed type pattern `var _`, the unnamed pattern can be nested in a record pattern. It cannot, however, be used as a top-level pattern in, e.g., an `instanceof` expression or a `case` label.

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 value 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 making it clear that the `y` and `Color` component values are not extracted.

Revisiting the `Box` and `Ball` example, we can further simplify its final case label by using the unnamed pattern instead of `var _`:

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

Risks and Assumptions

- We assume that little if any actively-maintained code uses underscore as a variable name. Developers migrating from Java 7 to Java 22 without having seen the warnings issued in Java 8 or the errors issued since Java 9 could be surprised. They face the risk of dealing with compile-time errors when reading or writing variables named _ and when declaring any other kind of element (class, field, etc.) with the name _.
- We expect developers of static analysis tools to understand the new role of underscore for unnamed variables and 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 subtle interactions with the specification (e.g., what does it mean to override a method with unnamed parameters?) and tooling (e.g., how do you write JavaDoc for unnamed parameters?). This may be the subject of a future 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.