RAOUL-GABRIEL URMA

RICHARD WARBURTON

varとJava 10で拡張された型推論

ローカル変数の型推論を使うためのベスト・プラクティス

Java 10 には、ローカル変数の型推論という華々しい言語機能が新しく導入されています。その主な目的は、定型挿入文を減らし、コードの可読性を向上させることです。ローカル変数を宣言する際に、型を var キーワードで置き換えると、コンパイラが、変数を初期化する記述から推論した適切な型を当てはめてくれます。次に例を示します。

```
Map<User, List<String>> userChannels = new HashMap<>();
```

Java 10 では上記のコードを次のように書くことができます。

```
var userChannels = new HashMap<User, List<String>>();
```

簡潔になること以外にも、この型推論によるメリットがいくつかあります。メリットについては本記事で詳しく説明しますが、その前に、より複雑な例を見てみます。

```
Path path = Paths.get("src/web.log");
try (Stream<String> lines = Files.lines(path)){
   long warningCount =
      lines.filter(line -> line.contains("WARNING"))
           .count();
   System.out.println(
      "Found " + warningCount + " warnings in the log file");
} catch (IOException e) {
   e.printStackTrace();
```

```
}
```

Java 10 では、次のようにリファクタリングすることができます。

```
var path = Paths.get("src/web.log");
try (var lines = Files.lines(path)){
    var warningCount =
        lines.filter(line -> line.contains("WARNING"))
             .count();
    System.out.println(
        "Found " + warningCount + " warnings in the log file");
} catch (IOException e) {
    e.printStackTrace();
}
```

コード中のそれぞれの式が、以下の静的な型（値に対して宣言されている型）を持つ点は変わりありません。

- ローカル変数 path は Path 型
- 変数 lines は Stream<String> 型
- 変数 warningCount は long 型

つまり、別の型の値を代入した場合、失敗します。たとえば、次のコードのような再代入を行った場合、コンパイル・エラーが発生します。

```
var warningCount = 5;
warningCount = "6";

| Error:
| incompatible types: java.lang.String cannot be converted to int
| warningCount = "6"
```

しかし、型推論には、多少の懸念事項もあります。たとえば、Vehicle クラスのサブクラスである Car クラスと Bike クラスが存在し、var v = new Car(); という宣言を行った場合、v は Car 型として宣言されるのでしょうか。それとも、Vehicle 型になるのでしょうか。

非常にわかりやすいのは、初期化に使われている型（この場合は Car）が、省略されているという単純な説明です。初期化しない限り var を使うことはできないことも、その裏付けになり得ます。

ただしこれは、その後の v = new Bike(); という代入が不可能になることを意味します。言い換えれば、var はポリモフィズム（多様性）のあるコードとは相性がよくありません。

ローカル変数の型推論を使えない場所

ローカル変数の型推論が機能しない場所はどこでしょうか。まず、これが機能するのはローカル変数のみです。フィールドやメソッドのシグネチャで使うことはできません。たとえば、次のようなことはできません。

```
public long process(var list) { }
```

また、明示的に初期化せずにローカル変数を宣言することもできません。すなわち、var 構文だけを使い、値なしで変数を宣言することはできません。

次のコード

```
var x;
```

はコンパイラ・エラーになります。

```
| Error:
| cannot infer type for local variable x
|   (cannot use 'var' on variable without initializer)
| var x;
| ^----^
```

var 変数を null で初期化することもできません。後ほど初期化することを意図してのものでしょうが、確かにこれでは型を何にするべきかを判断できません。

```
| Error:
| cannot infer type for local variable x
|   (variable initializer is 'null')
| var x = null;
| ^-----------^
```

さらに、var とラムダ式を併用することもできません。ラムダ式では、対象となる型が明示されている必要があるからです。次の代入は失敗します。

```
var x = () -> {}
```

そして、以下のエラー・メッセージが生成されます。

```
| Error:
| cannot infer type for local variable x
|   (lambda expression needs an explicit target-type)
| var x = () -> {};
| ^---------------^
```

しかし、奇妙に思えるかもしれませんが、次の代入は有効です。右辺で明示的に初期化を行っているからです。

```
var list = new ArrayList<>();
```

この場合、list の静的な型は何になるのでしょうか。推論される変数の型は ArrayList<Object> ですが、これではジェネリクスのメリットを活用できないため、あまり役には立ちません。そのため、こういったタイプの代入を記述するのは避けた方がよいでしょう。

無名型の型推論

Java には、いくつかの無名型が存在します。無名型とは、プログラム中に存在可能であるものの、その型の名前を明示的に記述する方法がない型を指します。わかりやすい無名型の例は、匿名クラスです。匿名クラスには、フィールドやメソッドを追加できますが、Java コードに匿名クラスの名前を記述することはできません。ダイヤモンド演算子を匿名クラスとともに使うことはできませんが、var の制限はそれよりも緩く、一部の無名型（具体的には、匿名クラスと交差型）とともに使うことができます。

また、var キーワードを利用すると、匿名クラスをさらに効率的に使うことや、その他の方法では記述できない型を指すこともできます。通常、匿名クラスを作成するときにフィールドを追加することはできますが、別の場所からそのフィールドを参照することはできません。フィールドを参照するためには、名前付きの型に代入する必要があるからです。

たとえば、次のコードはコンパイルできません。productInfo の型は Object であり、Object の name フィールドと total フィールドにアクセスすることはできないからです。

```
Object productInfo = new Object() {
   String name = "Apple";
   int total = 30;
};

System.out.println(
   "name = " + productInfo.name + ", total = " + productInfo.total);
```

しかし、var を使って、この制限を回避できます。var で型付けを行うローカル変数に匿名クラスを代入した場合、コンパイラは、その親の型ではなく、匿名クラスの型を推論します。つまり、次のコードで示すように、匿名クラスで宣言されているフィールドを参照できます。

```
var productInfo = new Object() {
   String name = "Apple";
   int total = 30;
};
```

```
System.out.println(
   "name = " + productInfo.name + ", total = " + productInfo.total);
```

一見したところ、この機能は、興味深い言語トリビアではあるものの、ほとんど役に立つことはないと思われるかもしれません。しかし、特定の状況では便利な場合があります。たとえば、何らかのメソッドの中で、中間結果としていくつかの値を返したい場合です。通常ならば、1 つのメソッドの中で使うためだけに、専用の新しいクラスを作って管理する必要があります。たとえば、Collectors.averagingDouble() の実装の中では、そのために double 値の小さな配列が使われています。

var を利用すると、中間値を格納するために匿名クラスを使うという、よりよいアプローチをとることができます。

いくつかの商品があり、それぞれに名前、在庫数、品目ごとの金銭的価値（すなわち、品目単価）が関連付けられている場合を考えてみます。ここで、各品目の合計価格、すなわち、在庫数と品目単価を掛けたものを計算したいとします。これだけの情報しかない場合、それぞれの商品を単純に価格にマップすることもできます。しかし、結果を使って何か有用なことをするために、商品名も併せて取得したい場合もあるでしょう。

次に示すのは、Java 10 の var を使ってこれを実現する方法の例です。

```
var products = List.of(
   new Product(10, 3, "Apple"),
   new Product( 5, 2, "Banana"),
   new Product(17, 5, "Pear"));
var productInfos = products
   .stream()
   .map(product -> new Object() {
      String name = product.getName();
      int total = product.getStock() * product.getValue();
   })
```

```
    .collect(toList());
productInfos.forEach(prod -> System.out.println(
    "name = " + prod.name + ", total = " + prod.total));
```

このコードからは、次の結果が出力されます。

```
name = Apple, total = 30
name = Banana, total = 10
name = Pear, total = 85
```

すべての無名型を var と併用できるわけではありません。サポートされているのは、匿名クラスと交差型です。しかし、Java プログラマーに対して、今まで以上にややこしい、ワイルドカード関連のエラー・メッセージが表示されることがないように、ワイルドカードで表される型の推論は行われません。無名型がサポートされているのは、推論される型に可能な限り多くの情報を保持させるため、そしてローカル変数を追加してより多くのコードをリファクタリングできるようにするためでした。この機能の元々の意図は、先ほどのようなコードを書くことではなく、単に var が無名型をどのように扱うべきかという問題を解決することでした。無名型とともに var を使うことがニッチなトリビアになるのか、一般的になるのかを予想するのは困難です。

推奨事項

型推論によって Java コードを迅速に書けるようになるのは明らかですが、可読性の面ではどうでしょうか。開発者は、ソース・コードを書くことよりも、読むことにずっと多くの時間を費やしています。そのため、書きやすさよりも読みやすさを優先して最適化を行うべきです。var によってどの程度可読性が向上するかというのは、主観的なものです。開発者の中にも、var を嫌う人もいれば、好む人もいます。したがって、どうすればチームメイトにとってコードが読みやすくなるかを常に考えるべきです。つまり、var を使っているコードをチームメイトが読んで満足するなら、使うべきです。そうでなければ、使うべきではありません。

場合によっては、明示的に型を含めることで、可読性が損なわれることがあります。たとえば、Map の entrySet についてループ処理を行う場合、Map.Entry オブジェクトの型パラメータを再度記述する必要があります。次に示すのは、Map のループ処理を行い、国名をその国内の都市名に変換するコードの例です。

```
Map<String, List<String>> countryToCity = new HashMap<>();
// ...
for (Map.Entry<String, List<String>> citiesInCountry :
   countryToCity.entrySet()) {
      List<String> cities = citiesInCountry.getValue();
   // ...
}
```

var を使い、上記のコードを次のように書き換えて、繰り返しと定型挿入文を削減できます。

```
var countryToCity = new HashMap<String, List<String>>();
// ...
for (var citiesInCountry : countryToCity.entrySet()) {
   var cities = citiesInCountry.getValue();
   // ...
}
```

これが優れているのは、可読性だけではありません。コードの保守性においても、メリットがあります。

たとえば、同じようなコードで、都市名を表す String を、都市についての追加情報を含めることができる City クラスで置き換えるとします。型を明示している場合、その型を使っているコードをすべて書き換える必要があります。

```
Map<String, List<City>> countryToCity = new HashMap<>();
// ...
for (Map.Entry<String, List<City>> citiesInCountry :
   countryToCity.entrySet()) {
      List<City> cities = citiesInCountry.getValue();
      // ...
}
```

しかし、var キーワードと型推論を使っていた場合は、コードの最初の行を次のように書き換えるだけで済みます。その他の行を変更する必要はありません。

```
var countryToCity = new HashMap<String, List<City>>();
// ...
for (var citiesInCountry : countryToCity.entrySet()) {
   var cities = citiesInCountry.getValue();
   // ...
}
```

この例は、var を使う際に守るべき重要な原則を示しています。すなわち、書きやすさや読みやすさを優先するのではなく、保守のしやすさを優先するということです。保守のしやすさを優先すれば、可読性と、やがてプログラムが進化する際に変更する必要があるコードの量とのバランスをとることができます。

型推論を追加することはコードにとって必ずプラスになると主張するのは危険でしょう。コードに明示的な型を含めることで、可読性が向上する場合もあります。特にそう言えるのは、式を見ても、そこから生成される型が明確でない場合です。次のコードの場合、getCities() というメソッド呼出しを見ただけでは、何が返されるかわからないため、型を明示的に記述する方がよいでしょう。

```
Map<String, List<City>> countryToCity = getCities();
var countryToCity = getCities();
```

可読性と var について突き詰めていけば、最終的に 1 つの推奨事項にたどりつきます。それは、変数名が重要だということです。var を使った場合、コードを読む人が変数の型を見ただけでコードの意図を推測することはできなくなります。そのため、開発者には、ローカル変数に適切な名前を与えることが今まで以上に求められます。理屈の上では、これは Java 開発者が今までも行ってきたはずのことです。しかし実際には、Java コードの可読性に関する問題の多くは、新しい言語機能というよりも、変数のネーミングなどの、既存の慣習に関係しています。

型推論と IDE

多くの IDE が提供しており、よく使われている機能の 1 つに、ローカル変数を抽出し、その変数の正しい型を推測して記述してくれるというものがあります。この機能には、Java 10 の var キーワードと重なる部分があります。この IDE 機能と var は、いずれも型を明示的に記述する手間を省くものですが、その点を除けば、それぞれに異なるトレードオフがあります。

ローカル変数抽出機能では、ローカル変数が生成される際に、完全に明示的な型がコードに記述されます。一方、var は、コードに明示的な型を記述する必要性自体をなくすものです。この 2 つの機能は、コードを簡単に書くという意味ではいずれも同じような価値を持つものですが、var は、ローカル変数抽出機能とは違った方法で可読性を変えることになります。前述のように、var は主に可読性に関するメリットをもたらすものですが、逆にその妨げになってしまう場合もあります。

Java と他のプログラミング言語の比較

変数の型推論を含む言語は Java だけではなく、Java が初めてというわけでもありません。実のところ、Java 10 の var で導入されたのは、かなり限定的な型推論です。これにより、アプローチがシンプルに保たれるとともに、var 宣言に関連するコンパイラのエラーが 1 つの文に限定されます。var の推論アルゴリズムが型の推論を行う際は、変数に代入される式のみを見ればよいからです。

Java 10のわずか半年後にリリースされる予定となっているJava 11では、ラムダ式のパラメータでvarキーワードを使用できるようになる見込みです。

まとめ

var は、Java 言語に追加された便利な機能で、生産性や可読性の向上に役立ちます。しかし、期待はそれだけにとどまりません。Java の今後のバージョンは、言語の着実な進化と最新化を続けていきます。たとえば、Java 10 のわずか半年後にリリースされ、長期的にサポートされる予定となっている Java 11 では、ラムダ式のパラメータで var キーワードを使用できるようになる見込みです。次の例のように、仮パラメータで型推論をしつつ、さらに Java アノテーションを付加できるため、この機能は便利です。

```
(@Nonnull var x, var y) -> x.process(y)
```

関数型プログラミング言語で実装されており、主流になりつつあるいくつかの機能も、今後のバージョンの Java での実装に向けた作業が進んでいます。パターン・マッチングや値型がその例です。こういった改善によって、今後の Java が、開発者に親しまれ愛されている Java でなくなってしまうわけではありません。今までよりも、さらに柔軟性と可読性が高まり、いっそう簡潔になるというだけのことです。</article>

Raoul-Gabriel Urma（@raoulUK）：イギリスのデータ・サイエンティストや開発者の学習コミュニティをリードする Cambridge Spark の CEO/ 共同創業者。若いプログラマーや学生のコミュニティである Cambridge Coding Academy の会長 / 共同創設者でもある。ベストセラーとなったプログラミング関連書籍『Java 8 in Action』（Manning Publications、2015 年）の共著者として執筆に携わった。ケンブリッジ大学でコンピュータ・サイエンスの博士号を取得している。

Richard Warburton（@richardwarburto）：Java Champion であり、ソフトウェア・エンジニアの傍ら、講師や著述も行う。ベストセラーとなった『Java 8 Lambdas』（O'Reilly Media、2014 年）の著者であり、Iteratr Learning と Pluralsight で開発者の学習に貢献し、数々の講演やトレーニング・コースを実施している。ウォーリック大学で博士号を取得している。