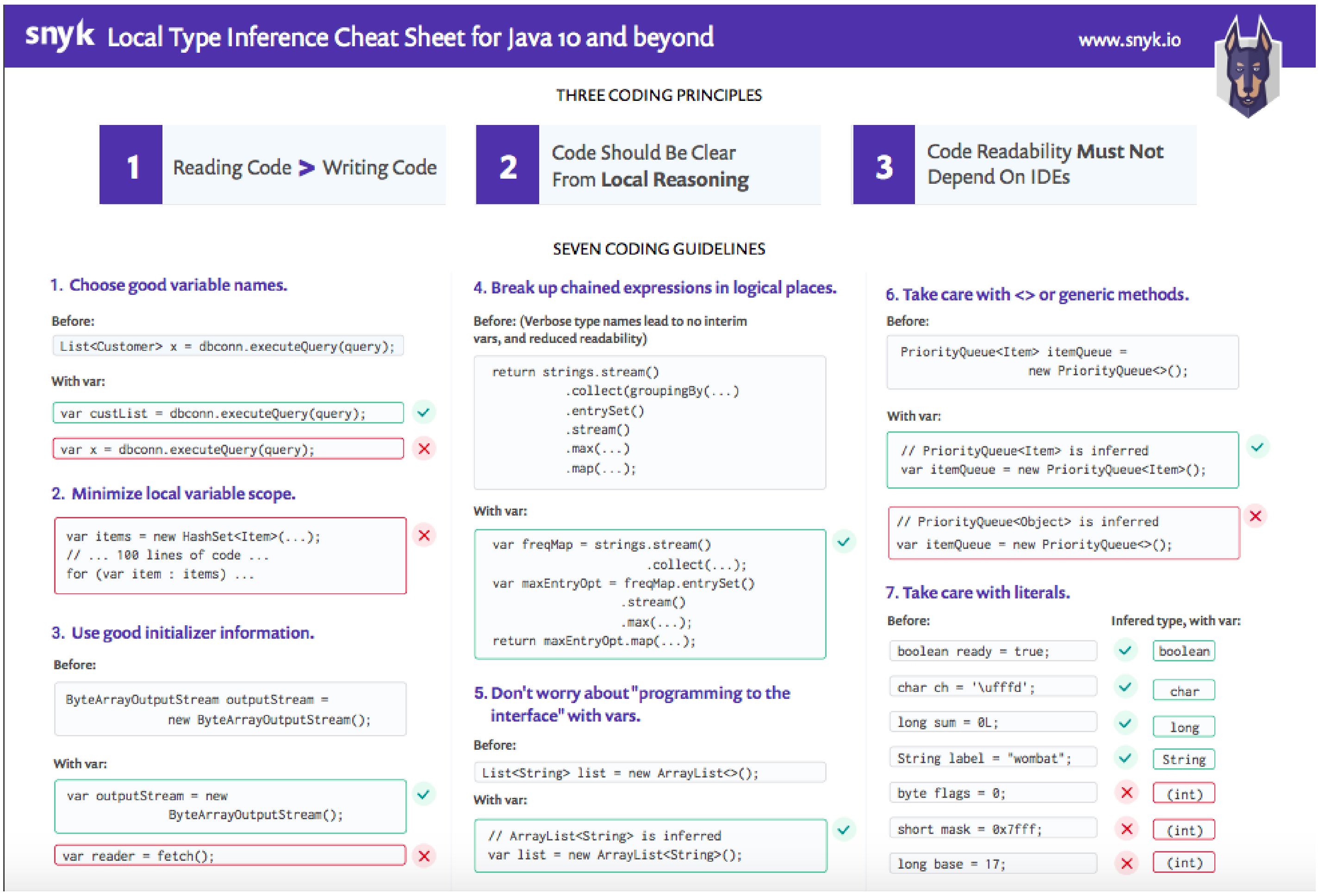
Local Type Inference Cheat Sheet for Java 10 and Beyond

by Liran Tal · May. 15, 19 · Java Zone · Presentation



snyk Local Type Inference Cheat Sheet for Java 10 and beyond www.snyk.io

THREE CODING PRINCIPLES

1 Reading Code > Writing Code

2 Code Should Be Clear From **Local Reasoning**

3 Code Readability **Must Not** Depend On IDEs

SEVEN CODING GUIDELINES

1. Choose good variable names.

Before:

```
List<Customer> x = dbconn.executeQuery(query);
```

With var:

```
var custList = dbconn.executeQuery(query);
```
✓

```
var x = dbconn.executeQuery(query);
```
✗

2. Minimize local variable scope.

```
var items = new HashSet<Item>(...);
// ... 100 lines of code ...
for (var item : items) ...
```
✗

3. Use good initializer information.

Before:

```
ByteArrayOutputStream outputStream =
                new ByteArrayOutputStream();
```

With var:

```
var outputStream = new
                ByteArrayOutputStream();
```
✓

```
var reader = fetch();
```
✗

4. Break up chained expressions in logical places.

Before: (Verbose type names lead to no interim vars, and reduced readability)

```
return strings.stream()
            .collect(groupingBy(...)
            .entrySet()
            .stream()
            .max(...)
            .map(...);
```

With var:

```
var freqMap = strings.stream()
                    .collect(...);
var maxEntryOpt = freqMap.entrySet()
                .stream()
                .max(...);
return maxEntryOpt.map(...);
```
✓

5. Don't worry about "programming to the interface" with vars.

Before:

```
List<String> list = new ArrayList<>();
```

With var:

```
// ArrayList<String> is inferred
var list = new ArrayList<String>();
```
✓

6. Take care with <> or generic methods.

Before:

```
PriorityQueue<Item> itemQueue =
                    new PriorityQueue<>();
```

With var:

```
// PriorityQueue<Item> is inferred
var itemQueue = new PriorityQueue<Item>();
```
✓

```
// PriorityQueue<Object> is inferred
var itemQueue = new PriorityQueue<>();
```
✗

7. Take care with literals.

Before:		Infered type, with var:
`boolean ready = true;`	✓	`boolean`
`char ch = '\ufffd';`	✓	`char`
`long sum = 0L;`	✓	`long`
`String label = "wombat";`	✓	`String`
`byte flags = 0;`	✗	`(int)`
`short mask = 0x7fff;`	✗	`(int)`
`long base = 17;`	✗	`(int)`

The main premise behind the local type inference feature is pretty simple. Replace the explicit type in the declaration with the new reserved type name 'var' and its type will be inferred. So we could replace:

```
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
```

with:

```
var outputStream = new ByteArrayOutputStream();
```

And the type of `outputStream` will be inferred as `ByteArrayOutputStream`. Hang on, are we saying that Java is now allowing dynamic typing? Absolutely not! *ALL* type inference occurs at compile time and explicit types are baked into the byte code by the compiler. At runtime, Java is as static as it's ever been. Given the usage is so simple, this cheat sheet will focus on the most important aspect of local type inference – its practical usage. It will give guidance when you should use explicit typing and when you should consider type inference.

should use explicit typing and when you should consider type inference.

Since wanting to write this cheat sheet, Stuart Marks, JDK engineer from Oracle, wrote the perfect article giving both coding principles and guidance of the usage of using type inference. So, when I wanted to create a cheat sheet, I headed straight over to Stuart to see if we could include his thoughts and condense them into a cheat sheet for developers to pin up and use daily! I would heavily recommend you read Stuart's article in full, It really is worth your time!

Principles

1. Reading Code > Writing Code

Whether it takes you 10 minutes or 10 days to write a line of code, you'll almost certainly be reading it for many years to come. Code is only maintainable and understandable in the future if it's clear, concise, and, most importantly, contains all the necessary information to understand its purpose. The goal is maximizing understandability.

2. Code Should Be Clear From Local Reasoning

Bake as much information as you can into your code to avoid a reader having to look through different parts of the code base in order to understand what's going on. This can be through method or variable naming.

3. Code Readability Shouldn't Depend on IDEs

IDE's can be great. I mean *really* great! They can make a developer more productive or more accurate with their development. Code must be readable and understandable without relying on an IDE. Often, the code is read outside an IDE. Or, perhaps, IDEs will differ in how much information they provide the reader. Code should be self-revealing. It should be understandable on its face, without the need for assistance from tools.

The Decision Is Yours

The choice of whether to give a variable an explicit type or to let the Java compiler work it out for itself is a trade-off. On one hand, you want to reduce clutter, boilerplate, ceremony. On the other hand, you don't want to impair understandability of the code. The type declaration isn't the only way to convey information to the reader. Other means include the variable's name and the initializer expression. We should take all the available channels into account when determining whether it's OK to mute explicit typing from the equation for each variable.

Guidelines

1. Choose Variable Names That Provide Useful Information

This is good practice, in general, but it's much more important in the context of var. In a var declaration, information about the meaning and use of the variable can be conveyed using the variable's name. Replacing an explicit type with var should often be accompanied by improving the variable name. Sometimes, it might be useful to encode the variable's type in its name. For example:

```java
List<Customer> x = dbconn.executeQuery(query);

 var custList = dbconn.executeQuery(query);
```

2. Minimize the Scope of Local Variables

Limiting the scope of local variables is described in Effective Java (3rd edition), Item 57. It applies with extra force if var is in use. The problem occurs when the variable's scope is large. This means that there are many lines of code between the declaration of the variable and its usage. As the code is maintained, changes to types, etc. may end up producing different behavior. For example, moving from a List to a Set might look OK, but does your code rely on ordering later on in the same scope? While types are always set statically, subtle differences in implementations using the same interface may trip you up. Instead of simply avoiding var in these cases, one should change the code to reduce the scope of the may trip you up. Instead of simply avoiding var in these cases, one should change the code to reduce the scope of the local variables, and only then declare them with var. Consider the following code:

```
var items = new HashSet<Item>(...);
items.add(MUST_BE_PROCESSED_LAST);
for (var item : items) { ... }
```

This code now has a bug, since sets don't have a defined iteration order. However, the programmer is likely to fix this bug immediately, as the uses of the items variable are adjacent to its declaration. Now, suppose that this code is part of a large method, with a correspondingly large scope for the items variable:

```
var items = new HashSet<Item>(...);

// ... 100 lines of code ...

items.add(MUST_BE_PROCESSED_LAST);
for (var item : items) { ... }
```

This bug now becomes much harder to track down as the line tries to add an item to the end of the set isn't close enough to the type declaration to make the bug obvious.

3. Consider Var When the Initializer Provides Sufficient Information to the Reader

Local variables are often initialized with constructors. The name of the class being constructed is often repeated as the explicit type on the left-hand side. If the type name is long, the use of var provides concision without loss of information:

```
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

 var outputStream = new ByteArrayOutputStream();
```

It's also reasonable to use var in cases where the initializer is a method call, such as `Files.newBufferedReader(…)` or `List stringList = List.of("a", "b", "c")`.

4. Use Var to Break Up Chained or Nested Expressions With Local Variables

Consider code that takes a collection of strings and finds the string that occurs most often. This might look like the following:

```
return strings.stream()
               .collect(groupingBy(s -> s, counting()))
               .entrySet()
               .stream()
               .max(Map.Entry.comparingByValue())
               .map(Map.Entry::getKey);
```

This code is correct but more readable over multiple statements. The problem with splitting over statements is shown below.

```
Map<String, Long> freqMap = strings.stream()
```

```
                                    .collect(groupingBy(s -> s, counting()));
Optional<Map.Entry<String, Long>> maxEntryOpt = freqMap.entrySet()
                                                    .stream()
                                                    .max(Map.Entry.comparingByValue());
return maxEntryOpt.map(Map.Entry::getKey);
```

But the author probably resisted doing so because the explicit typing looks extremely messy, distracting from the important code. Using var allows us to express the code more naturally without paying the high price of explicitly declaring the types of the intermediate variables:

```
var freqMap = strings.stream()
                     .collect(groupingBy(s -> s, counting()));
var maxEntryOpt = freqMap.entrySet()
                         .stream()
                         .max(Map.Entry.comparingByValue());
return maxEntryOpt.map(Map.Entry::getKey);
```

One might legitimately prefer the first snippet with its single long chain of method calls. However, in some cases, it's better to break up long method chains.

5. Don't Worry Too Much About "Programming to the Interface" With Local Variables

A common idiom in Java programming is to construct an instance of a concrete type but to assign it to a variable of an interface type. For example:

```
List<String> list = new ArrayList<>();
```

If var is used, however, the concrete type is inferred instead of the interface:

```
// Inferred type of list is ArrayList<String>.
 var list = new ArrayList<String>();
```

Code that uses the list variable can now form dependencies on the concrete implementation. If the variable's initializer were to change in the future, this might cause its inferred type to change, causing errors or bugs to occur in subsequent code that uses the variable.

This is less of a problem when adhering to guideline 2, as the scope of the local variable is small, the risks from "leakage" of the concrete implementation that can impact the subsequent code are limited.

6. Take Care When Using Var With Diamond or Generic Methods

Both var and the "diamond" feature allow you to omit explicit type information when it can be derived from information already present. However, if used together, they might end up omitting all the useful information the compiler needs to correctly narrow down the type you wish to be inferred.

Consider the following:

```
PriorityQueue<Item> itemQueue = new PriorityQueue<Item>();
PriorityQueue<Item> itemQueue = new PriorityQueue<>();
 var itemQueue = new PriorityQueue<Item>();

```

```
// DANGEROUS: infers as PriorityQueue<Object>
 var itemQueue = new PriorityQueue<>();
```

Generic methods have also employed type inference so successfully that it's quite rare for programmers to provide explicit type arguments. Inference for generic methods relies on the target type if there are no actual method arguments that provide sufficient type information. In a var declaration, there is no target type, so a similar issue can occur as with diamond. For example:

```
// DANGEROUS: infers as List<Object>
 var list = List.of();
```

With both diamond and generic methods, additional types of information can be provided by actual arguments to the constructor or method, allowing the intended type to be inferred. This does add an additional level of indirection but is still predictable. Thus:

```
// OK: itemQueue infers as PriorityQueue<String>
Comparator<String> comp = ... ;
 var itemQueue = new PriorityQueue<>(comp);
```

7. Take Care When Using Var With Literals

It's unlikely that using var with literals will provide many advantages, as the type names are generally short. However, var is sometimes useful, for example, to align variable names.

There is no issue with boolean, character, long, and string literals. The type inferred from these literals is precise, and so, the meaning of var is unambiguous. Particular care should be taken when the initializer is a numeric value, especially an integer literal. With an explicit type on the left-hand side, the numeric value may be silently widened or narrowed to types other than int. With var, the value will be inferred as an int, which may be unintended.

```
// ORIGINAL
boolean ready = true;
char ch = '\ufffd';
long sum = 0L;
String label = "wombat";
byte flags = 0;
short mask = 0x7fff;
long base = 17;

 var ready = true;
 var ch    = '\ufffd';
 var sum   = 0L;
 var label = "wombat";

// DANGEROUS: all infer as int
 var flags = 0;
 var mask = 0x7fff;
 var base = 17;
```

Conclusion

Using var for declarations can improve code by reducing clutter, thereby letting more important information stand out.

On the other hand, applying var indiscriminately can make things worse. Used properly, var can help improve good code, making it shorter and clearer without compromising understandability. When using var, ask yourself if you've made the code more ambiguous, or whether it's clearly understandable without too much investigation.

We invite you to visit the Snyk's blog for other cheat sheets or download this cheat sheet and pin it up!

Like This Article? Read More From DZone

Finally, Java 10 Has var to Declare Local Variables

Java 10's JEP 286: Local-Variable Type Inference

Using JDK 10's Local Variable Type Inference With jOOQ

Free DZone Refcard
Getting Started With Headless CMS

Topics: JAVA , WEB DEVELOPEMENT , JDK 10 , LOCAL TYPE INFERENCE , JAVA 10 , VAR , INTITIALIZER , CHEAT SHEET

Published at DZone with permission of Liran Tal . See the original article here.
Opinions expressed by DZone contributors are their own.

Your Guide to Java Streams [Tutorials and Articles]

by Peter Connelly STAFF MVB · Dec 09, 19 · Java Zone · Presentation

In this edition of "Best of DZone," we've compiled our best tutorials and articles on one of the most popular APIs in Java, Streams. Whether you're a beginner just looking to bring in some elements of functional programming into a Java application, or a Streams vet, we've got your back!

Before we begin, we'd like need to thank those who were a part of this article. DZone has and continues to be a community powered by contributors like you who are eager and passionate to share what they know with the rest of the world.

Let's get started!

Overview

- A Guide to Streams: In-Depth Tutorial With Examples by Eugen Paraschiv — Java 8 and streams: A match made in heaven, but it can be a little overwhelming. In this post, we take an in-depth look at this combination with some examples.

- Dipping Into Java 8 Streams by Dan Newton — With a working knowledge of lambda expressions and method references, you can see how streams enable you to work with Collections while efficiently adding logic.

- Java Streams Overview, Part 1 and Part 2 by Zoltan Raffai — Get started working with Streams and learn everything you need to get started with basic classes, reading and writing operations, and working with errors.

- A Java 8 Streams Cookbook by Martin Farrell — If you're looking for a quick breakdown of Streams, look no further. This cookbook covers Streams' chief advantages, its operations, and a comprehensive example.

Processing Collections

- Process Collections Easily With Stream in Java 8 by Leona Zhang — Learn how you can process collections easily with Stream in Java 8.

- How to Compare List Objects in Java 7 vs. Java 8 by Arpan Das — In this blast from the not-too-distant past, we compare how Java 8's Stream API changed how you can compare List objects.

- Mastering Java 8 Streams Part 1 and Part 2 by Marco Behler — Want to get your feet wet with Java streams? This introductory video will get you creating streams and performing simple tasks on them.

- Convert a List to a Comma-Separated String in Java 8 by Mario Pio Gioiosa — This quick tutorial shows you how to use streams in Java to convert the contents of a list to a comma-separated string in Java 8.

Working With Files

- Split a File as a Stream by Peter Verhas — Curious about when you would actually use the `splitAsStream` method? Here's a use case of splitting a file into chunks that can be processed as streams.

- Java 8 Stream and Lambda Expressions – Parsing File Example by Eyal Golan — See a common use case for Streams and Lambda Expressions in this article.

Parallel Streams

- Think Twice Before Using Java 8 Parallel Streams by Lukas Krecan — Parallelization was the main driving force behind lambdas, stream API, and others. Let's take a look at an example of stream API.

- Should I Parallelize Java 8 Streams? by Santhosh Krishnan — What do we need to consider before parallelizing Java streams?

Streams and Collectors

- Using Java Streams and Collectors by Jay Sridhar — This short guide covers how to use Java 8 Streams and Collectors to slice and dice lists, including computing sums, averages, and partitioning.

- How to Use Java Stream Collectors by Yogen Rai — Want to learn more about using Java Stream collectors? Check out this post on collectors and how to use them.

- How to Transform Elements In a Stream Using a Collector by Hubert Klein Ikkink — Learn more about how to transform elements in a Stream using Java Collectors.

- The Ultimate Guide to the Java Stream API groupingBy() Collector by Grzegorz Piwowarek — Still a bit puzzled by what the `groupingBy()` collector can do in the Java Stream API? Check out this guide on using the `groupingBy()` collector to its fullest potential.

Stream Bugs

- Java Stream API Was Broken Before JDK 10 by Grzegorz Piwowarek — Stream API bugs can affect anyone still residing on JDK 8 and JDK 9. Click here to learn more.

Lambda Streams

- Java Lambda Streams and Groovy Closures Comparisons by Alex Staveley — Want to learn more about the difference between in lambda streams in both Java and Groovy? Check out this post to learn more about the differences between them.

Advanced Topics

- Are Java 8 Streams Truly Lazy? Not Completely! by Lukas Eder — Is it always the case that streams in Java 8 are evaluated lazily? Check out this article and find a possibly surprising result.

- Overview of Java Stream API Extensions by Piotr Mińkowski — Check out the top Java Stream API extensions offered by popular Java libraries.

- Java Stream: Is a Count Always a Count? Part 1 and Part 2 by Per-Åke Minborg — Learn more about counts in Java streams.

- How to Reuse Java Streams by Miguel Gamboa — Need to use your streams over and over again? Let's cover three different approaches, their benefits, and their pitfalls when recycling Java streams.

- Become a Master of Java Streams, Part 1: Creating Streams, Part 2: Intermediate Operations, Part 3: Terminal Operations, and Part 4: Database Streams, Part 5: Turn Joined Database Tables Into a Stream, and Part 6: Creating a New Database Application Using Streams by Per-Åke Minborg and Julia Gustafsson — Want to become a Java Streams Master?

- Exception Handling in Java Streams by Brian Vermeer — When you want to use a method that throws a `checkedException`, you have to do something extra if you want to call it in a lambda.

- Peeking Inside Java Streams With Stream.peek by Dustin Marx — How to use the Stream.peek(Consumer) method in Java to help visualize stream operations.

Be a Part of the Conversation!

Think we missed something? Want to contribute? Let us know in the comments below... or, join the conversation by becoming a member of our community of thousands of developers eager to share their knowledge and passion for programming with others.

Further Reading

- The Best of Java Collections [Tutorials].
- The Complete Apache Spark Collection [Tutorials and Articles].
- Your Guide to Automated Testing [Articles and Tutorials].

Topics: JAVA, JAVA 8, STREAMS API, COLLECTOR, LAMBDA EXPRESSION, COLLECTIONS, PARALLEL STREAMS, TUTORIAL