APPENDIX A

Local Variable Type Inference

In this chapter, you will learn:

- What type inference is
- What local variable type inference is
- What the restricted type name, `var`, is
- The rules to use `var` to declare local variables and indexes of `for-each` and `for` loops

What Is Type Inference?

Suppose you want to store a person ID, say 10, in a variable. You would declare the variable like so:

```
int personId = 10;
```

To store a list of person IDs, you would declare a variable like so:

```
ArrayList<Integer> personIdList = new ArrayList<Integer>();
```

How about rewriting these variable declarations as follows?

```
var personId = 10;
var personIdList = new ArrayList<Integer>();
```

You might say that this looks like JavaScript code, not Java code. That was correct until Java 10. In Java 10, you can use `var` to declare *local variables with an initializer*. The compiler will take care of inferring the correct type (sometimes, unintended incorrect type) for your variable. In Java 10, the previous snippet of code is the same as follows:

```
int personId = 10;
ArrayList<Integer> personIdList = new ArrayList<Integer>();
```

Does it mean that you do not need to use an explicit type to declare local variables in Java anymore? Does it mean that Java has become dynamically typed like JavaScript? No. None of these is true. Using `var` to declare local variable has limitations—some are technical and some are practical. Java is still a strongly and statically typed language. The examples you just saw are syntactic sugar added by the compiler. The compiler infers the type of the variables and generates the byte code, which is the same as before Java 10.

K. Sharan, *Java Language Features*, https://doi.org/10.1007/978-1-4842-3348-1

The Java runtime never sees var! The next few sections give you a little background of type inference and all the rules and exceptions on how and where to use var to declare local variables.

Everything in Java revolves around types. Either you are declaring types like classes and interfaces or you are using them. Every expression and value you use in your program has a type. For example, the value 10 is of int type, 10.67 is of double type, "Hello" is of String type, and the expression new Person() is of Person type.

Deriving the type of a variable or an expression from the context of its use is called *type inference.* Java already has type interference, for example, in lambda expressions and in object creation expression for a generic type. The following statement uses type inference, which infers the diamond operator (<>) on the right side as <Integer> by reading the ArrayList<Integer> from the left side:

```
ArrayList<Integer> personIdList = new ArrayList<>();
```

What Is var?

Is var a keyword in Java 10? No. It is a *restricted type name* or a *context sensitive keyword*. It is a type name, which means it denotes a type (primitive type or reference type) like int, double, boolean, String, List, etc. Because var is a restricted type name, you can still use it as an identifier, except to define another type. That is, you can use var as variable names, method names, and package names. However, you cannot use it as the name of a class or an interface. Java coding standards recommend that the names of classes and interfaces start with an uppercase letter. If you do not follow the coding standards and use var as a class or interface name, your code will break in Java 10.

You can use var to declare local variables in the following three contexts. There are many restrictions to this list. I cover them one by one.

- Local variables with initializers
- Indexes in for-each loops
- Indexes in for loops

The following are examples of using var to declare local variables with initializers:

```
// The type of birthYear is inferred as int
var birthYear = 1969;

// The type of promptMsg is inferred as String
var promptMsg = "Are you sure you want to proceed?";

// Assuming Person is a class, the type of john is inferred as Person
var john = new Person();

// Using var as a variable name is allowed. Here var will be inferred as int type
var var = 200;
```

The compiler infers the type of the variable declared using var. If the compiler cannot infer the type, a compile-time error occurs. The compiler uses the type of the expression used as the initializer to infer the type of the variable being declared. Consider the previous variable declarations. In the first case, the

compiler sees 1969, which is an integer literal of type `int` and infers type of the `birthYear` variable as `int`. When the source code is compiled, the class file contains the statement:

```
// The compiler inferred the type of birthYear from 1969 and replaced var with int
// in the class file. The runtime sees the following statement as it did before Java 10.
int birthYear = 1969;
```

■ **Tip** Type inference using `var` is performed by the compiler. Using `var` has no impact on the existing class file and the runtime.

The same argument goes for the other two statements in which the type of the `promptMsg` and `john` variables are inferred as `String` and `Person`, respectively.

Looking at the previous examples, you may argue that why would you not write these statements using explicit types as the following?

```
int birthYear = 1969;
String promptMsg = "Are you sure you want to proceed?";
Person john = new Person();
```

You have a point. Except for saving a few key strokes, you really made the job for the readers a little harder when you used `var`. When `var` is used, the reader must look at the initializer to know the type of the variable being declared. In simple expressions like the ones in these examples, it may not be very difficult to know the type of the initializer by just looking at it. One of the arguments made by the designer of the `var` type name is that using it to declare multiple variable at the same place makes the variable names easy to read because they all vertically align. Look at the following snippet of code:

```
var birthYear = 1969;
var promptMsg = "Are you sure you want to proceed?";
var john = new Person();
```

Compare the two previous snippets of code declaring the same set of variables. The latter is a little easier to read, but still a little harder to understand.

Let's consider the following variable declaration that uses a method call to initialize its value:

```
var personList = persons();
```

In this case, the compiler will infer the type of the `personList` variable based on the return type of the `persons()` method. For the readers of this statement, there is no way to tell the inferred type unless they look at the declaration of the `persons()` method. I gave the variable a good name, `personList`, to indicate its type. However, it is still not clear whether it is a `List<Person>`, a `Person[]`, or some other type.

Consider another example of a variable declaration:

```
Map<Integer,List<String>> personListByBirthMonth = new HashMap<Integer,List<String>>();
```

You can rewrite this statement as follows:

```
var personListByBirthMonth = new HashMap<Integer,List<String>>();
```

This time, the declaration looks much simpler and you may benefit from type inference offered by `var`, provided you keep the variable name intuitive enough to give a clue about its type.

You can mix `var` and explicit type names in the same section of the code (methods, constructors, static initializers, and instance initializers). Use `var` to declare variables whenever the types of the variables are clear to the readers, not just clear for the writer. If the reader of your code cannot figure out the variable types easily, use the explicit type names. Always choose clarity over brevity. It is very important that you use intuitive variable names when you use `var`. Consider the following variable declaration:

```
var x = 156.50;
```

The variable name `x` is terrible. If your method is a few lines long, using `x` as the variable name may be excused. However, if your method is big, the reader will have to scroll through and look at the variable declaration to find out what `x` means whenever `x` appears in the code. So, it is important to keep the variable name intuitive. Consider replacing the previous variable declaration with one such as follows:

```
var myWeightInPounds = 156.50;
```

No one reading the code that uses `myWeightInPounds` will have any doubt about its type.

A Quick Example

Listing A-1 contains a test class to show you how to use the `var` restricted type name to declare local variables. I cover many more examples shortly. The comments in the code and the output make it obvious as to what is going on during the type inference. I do not explain the code any further.

Listing A-1. Using the var Restricted Type Name to Declare Local Variables

```
// VarTest.java
package com.jdojo.java10.newfeatures;

import java.util.List;
import java.util.Arrays;

public class VarTest {
    public static void main(String[] args) {
        // The inferred type of personId is int
        var personId = 1001;
        System.out.println("personId = " + personId);

        // The inferred type of prompt is String
        var prompt = "Enter a message:";
        System.out.println("prompt = " + prompt);

        // You can use methods of the String class on prompt as you did
        // when you declared it as "String prompt = ..."
        System.out.println("prompt.length() = " + prompt.length());
        System.out.println("prompt.substring(0, 5) = " + prompt.substring(0, 5));

        // Use an explicit type name, double
        double salary = 1878.89;
        System.out.println("salary = " + salary);
```

```
        // The inferred type of luckyNumbers is List<Integer>
        var luckyNumbers = List.of(9, 19, 1969);
        System.out.println("luckyNumbers = " + luckyNumbers);

        // The inferred type of cities is String[]
        var cities = new String[]{"Atlanta", "Patna", "Paris", "Gaya"};
        System.out.println("cities = " + Arrays.toString(cities));
        System.out.println("cities.getClass() = " + cities.getClass());

        System.out.println("\nList of cities using a for loop:");

        // The inferred type of the index, i, is int
        for (var i = 0; i < cities.length; i++) {
            System.out.println(cities[i]);
        }

        System.out.println("\nList of cities using a for-each loop:");

        // The inferred type of the index, city, is String
        for (var city : cities) {
            System.out.println(city);
        }
    }
}
```

```
personId = 1001
prompt = Enter a message:
prompt.length() = 16
prompt.substring(0, 5) = Enter
salary = 1878.89
luckyNumbers = [9, 19, 1969]
cities = [Atlanta, Patna, Paris, Gaya]
cities.getClass() = class [Ljava.lang.String;

List of cities using a for loop:
Atlanta
Patna
Paris
Gaya

List of cities using a for-each loop:
Atlanta
Patna
Paris
Gaya
```

I am sure you have many questions about local variable type inference covering different use-cases. I attempt to cover most use-cases in the next section with examples. I found the JShell tool, which is a command-line tool that ships with the JDK starting from JDK9, invaluable to experiment with `var`. If you are not familiar with the JShell tool, refer to Chapter 23 of my book titled *Beginning Java 9 Fundamentals* (ISBN: 978-1484228432). I use JShell many times to show code snippets and results in next sections. Make sure to set the feedback mode to the JShell session to verbose, so JShell prints the variable declarations with inferred types when you use `var`. The following JShell session shows a few of the variable declarations used in the previous examples:

```
C:\Java10NewFeatures>jshell
|  Welcome to JShell -- Version 10-ea
|  For an introduction type: /help intro

jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> var personId = 1001;
personId ==> 1001
|  created variable personId : int

jshell> var prompt = "Enter a message:";
prompt ==> "Enter a message:"
|  created variable prompt : String

jshell> var luckyNumbers = List.of(9, 19, 1969);
luckyNumbers ==> [9, 19, 1969]
|  created variable luckyNumbers : List<Integer>

jshell> var cities = new String[]{"Atlanta", "Patna", "Paris", "Gaya"};
cities ==> String[4] { "Atlanta", "Patna", "Paris", "Gaya" }
|  created variable cities : String[]

jshell> /exit
|  Goodbye

C:\Java10NewFeatures>
```


Rules of Using var

The previous sections covered the basic rules of using the `var` restricted type name. In this section, I cover more detailed rules with examples. While reading these rules, keep in mind that there was only one objective in introducing `var` in Java 10—making developers' lives easier by keeping the rules of using `var` simple. I use JShell sessions in examples. I explain the following rules of using `var` to declare local variables:

- Variable declarations without initializers are not supported.
- The initializer cannot be of the `null` type.
- Multiple variables in a single declaration are not supported.
- The initializer cannot reference the variable being declared.
- Array initializers by themselves are not supported.

- No array dimensions can be specified while declaring arrays.
- No poly expressions such as lambda expressions and method references are allowed as initializers.
- Instance and static fields are not supported.
- `var` is not supported as the return type of a method. You cannot use it to declare method's arguments.

If you need to declare a variable that needs to use features not allowed according to these rules, use an explicit type instead of `var`.

No Uninitialized Variables

You cannot use `var` to declare uninitialized variables. In such cases, the compiler has no expression to use to infer the type of the variable.

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

This rule makes using `var` simple in your code and easy to diagnose errors. Assume that declaring uninitialized variables using `var` is allowed. You can declare a variable and assign it a value for the first time several lines later in your program, which would make the compiler infer the variable's type. If you get a compile-time error in another part, which tells you that you have assigned a value to your variable wrong type, you need to look at the first assignment to the variable that decided the variable's type. This is called "action at a distance" inference error in which an action at one point may cause an error in another part later—making the developer's job harder to locate the error.

No null Type Initializers

You can assign `null` to any reference type variable. The compiler cannot infer the type of the variable if `null` is used as the initializer using `var` type name.

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

No Multiple Variable Declarations

You cannot declare multiple variables in a single declaration using `var`.

```
jshell> var personId = 1001, days = 9;
|  Error:
|  'var' is not allowed in a compound declaration
|  var personId = 1001, days = 9;
|      ^
```

Cannot Reference the Variable in the Initializer

When you use `var`, you cannot reference the variable being declared in the initializer. For example, the following declaration is invalid:

```
jshell> var x = (x = 1001);
|  Error:
|  cannot infer type for local variable x
|    (cannot use 'var' on self-referencing variable)
|  var x = (x = 1001);
```

However, the following declaration is valid when you use an explicit type:

```
// Declares x as an int and initializes it with 1001
int x = (x = 1001);
```

No Array Initializers by Themselves

Array initializer such as `{10, 20}` is a poly expression and its type depends on the left side of the assignment. You cannot use an array initializer with `var`. You must use an array creation expression with or without an array initializer such as `new int[]{10, 20}` or `new int[2]`. In such cases, the compiler infers the type of the variable the same as the type used in array creation expression. The following JShell session shows a few examples:

```
jshell> var evens = {2, 4, 6};
|  Error:
|  cannot infer type for local variable evens
|    (array initializer needs an explicit target-type)
|  var evens = {2, 4, 6};
|  ^--------------------^

jshell> var evens = new int[]{2, 4, 6};
evens ==> int[3]{ 2, 4, 6 }
|  created variable evens : int[]

jshell> var evens = new int[2];
evens ==> int[2] { 0, 0 }
|  modified variable evens : int[]
|    update overwrote variable evens : int[]
```

No Array Dimensions

While using `var` to declare a variable, you cannot use brackets ([ ]) after `var` or a variable's name to specify the dimension of the array. The dimension of the array being declared is inferred from the initializer.

```
jshell> var[] cities = new String[3];
|  Error:
|  'var' is not allowed as an element type of an array
|  var[] cities = new String[3];
|        ^

jshell> var cities = new String[3];
cities ==> String[3] { null, null, null }
|  modified variable cities : String[]
|    update overwrote variable cities : String[]

jshell> var points3D = new int[3][][];
points3D ==> int[3][][] { null, null, null }
|  created variable points3D : int[][][]
```

No Poly Expressions as Initializers

Poly expressions need a target type to infer their types. No poly expressions such as lambda expressions and method references are allowed in initializers for variable declaration using `var`. Array initializers are poly expressions, which are also not allowed. You need to use explicit types with poly expressions. The following JShell session shows a few examples in which I use `var` to declare variables with poly expressions initializers, which generate errors. I also show using explicit types instead of `var`, which do not generate errors.

```
jshell> var increment = x -> x + 1;
|  Error:
|  cannot infer type for local variable increment
|    (lambda expression needs an explicit target-type)
|  var increment = x -> x + 1;
|  ^-----------------------^

jshell> Function<Integer,Integer> increment = x -> x + 1;
increment ==> $Lambda$13/2081853534@2a2d45ba
|  created variable increment : Function<Integer, Integer>

jshell> var next = increment.apply(10);
next ==> 11
|  created variable next : Integer

jshell> var intGenerator = new Random()::nextInt;
|  Error:
|  cannot infer type for local variable intGenerator
|    (method reference needs an explicit target-type)
|  var intGenerator = new Random()::nextInt;
|  ^---------------------------------------^
```

```
jshell> Supplier<Integer> intGenerator = new Random()::nextInt;
intGenerator ==> $Lambda$14/1734161410@51565ec2
|  created variable intGenerator : Supplier<Integer>

jshell> int nextInteger = intGenerator.get();
nextInteger ==> -2061949196
|  created variable nextInteger : int
```

Inferring Types on Both Sides

In most cases, you get an error when the compiler cannot infer the type of the initializer and the variable. In some cases, `Object` is inferred as the type. Consider the following:

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

The initializer uses a diamond operator, which makes the compiler infer the type for `ArrayList<>`, whereas using `var` on the left side makes the compiler infer the type in `ArrayList<>`. You might expect an error in this case. However, the compiler replaces the previous declaration with the following:

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

Consider the following snippet of code, which is also asking for inferring types on both sides:

```
public class NumberWrapper<E extends Number> {
    // More code goes here
};

var wrapper = new NumberWrapper<>();
```

In this case, the inferred type is the upper bound of the type parameter `E extends Number`, which is `Number`. The compiler will replace the previous variable declaration with the following

```
NumberWrapper<Number> wrapper = new NumberWrapper<>();
```

Inferring Non-Denotable Types

Using `var` to declare variables may lead the compiler to infer non-denotable types such as capture variable types, intersection types, and anonymous class types. Consider the following snippet of code, which uses an anonymous class. Note that an anonymous class does not have a name, so it is non-denotable.

```
class Template implements Serializable, Comparable {
    @Override
    public int compareTo(Object o) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    // More code goes here
}
```

```
var myTemplate = new Template() {
    // More code goes here
};
```

What would be the inferred type of the variable `myTemplate`? If you specify an explicit type for the `myTemplate` variable, you have the following four choices:

- `Template myTemplate = …`
- `Serializable myTemplate = …`
- `Comparable myTemplate = …`
- `Object myTemplate = …`

The compiler has a fifth choice, which is the non-denotable type of the anonymous class. In this case, the compiler infers the non-denotable type of the anonymous class. When you add a new method to the anonymous class, which does not override any method in its superclass or superinterface, you cannot call that method statically. However, using `var` in anonymous class declaration allows you to do so, because your inferred variable type is a compiler-generated non-denotable type of the anonymous class. The following JShell session demonstrates this:

```
jshell> var runnable = new Runnable() {
   ...>   public void run() {
   ...>     System.out.println("Inside run()...");
   ...>   }
   ...>
   ...>   public void test() {
   ...>     System.out.println("Calling this method is possible because of var...");
   ...>   }
   ...> };
runnable ==> $1@2a2d45ba
|  created variable runnable : <anonymous class implementing Runnable>

jshell> runnable.run();
Inside run()...

jshell> runnable.test();
Calling this method is possible because of var...
```

Consider the following statement, where the return type of the `getClass()` method in the `Object` class is `Class<?>`:

```
// The inferred type of name is String
var name = "John";

// The inferred type of cls is Class<? extends String>
var cls = name.getClass();
```

In this case, the capture variable type in `Class<?>` has been projected upward to the supertype of the class of the `name` variable. The inferred type of `cls` is `Class<? extends String>` rather than `Class<?>`.

Here is the last example of the inferred non-denotable types:

```
var list = List.of(10, 20, 45.89);
```

What would be the inferred type of the `list` variable? This is a harder case to understand. The arguments to the `List.of()` method are of `Integer` and `Double` types. The `Integer` and `Double` classes are declared as follows:

- `final class Integer extends Number implements Comparable<Integer>`
- `final class Double extends Number implements Comparable<Double>`

Both `Integer` and `Double` types implement the `Comparable` interface. In this case, the compiler projects the `Integer` and `Double` to their common supertype `Number` and uses an intersection of the `Number` and `Comparable` as the inferred type.

The previous variable declaration is equivalent to the following. Note that an intersection type is non-denotable, so you cannot use the following statement in your source code. I am showing it here just to show you the actual type being inferred. You can also use JShell to see this.

```
List<Number&Comparable<? extends Number&Comparable<?>>> list = List.of(10, 20, 45.89);
```

Using `var` to declare variables may become complex at the compiler level. Use explicit types in such situations to make your intention clear. Remember: clarity is more important than brevity. You could rewrite the previous statement as follows:

```
List<Number> list = List.of(10, 20, 45.89);
```

Not Allowed in Fields and Methods Declarations

Using `var` to declare fields (instance and static), method parameters, and method return type is not allowed. Public fields and methods define the interface for a class. They are referenced in other classes. Making a subtle change in their declaration may require that other classes be recompiled. Using `var` in local variables has local effects. This is the reason that fields and methods are not supported by `var`. Private fields and methods were considered, but not supported, to keep the implementation of `var` simple.

Using final with var

You can declare a variable declared using `var` as `final`. The `final` modifier behaves the same as it did before the introduction of `var`. The following local variable declaration is valid:

```
// The inferred type of personId is int.
// personId is final. You cannot assign another value to it.
final var personId = 1001;
```

Backward Compatibility

If your existing code uses `var` as a class or interface name, your code will break in Java 10. Using it as method, package, and variable names are fine. It is very unlikely that you have named a class or interface as `var` because that would have been against the Java naming convention.

Future Enhancements

In JDK10, `var` is not supported to declare formal parameters of implicitly typed lambda expressions. Consider the following statement, which uses an implicitly typed lambda expression:

```
BiFunction<Integer,Integer,Long> adder = (x, y) ->  (long) (x + y);
```

In this case, the types of the `x` and `y` parameters are inferred as `Integer`. In JDK10, you cannot write:

```
// A compile-time error in JDK 10
BiFunction<Integer,Integer,Long> adder = (var x, var y) ->  (long) (x + y);
```

It will be possible to write the previous statement in JDK11, which will be released in September 2018. This enhancement is tracked by a JDK Enhancement Proposal (JEP 323) at http://openjdk.java.net/jeps/323.

IDE Support

At the time of this writing, local variable inference has limited support in leading Java Integrated Development Environments (IDEs) such as IntelliJ and Eclipse. NetBeans does not support it yet. Visit the websites for these IDEs to see the support for the local variable inference. It is expected that IDEs will show the inferred type of variables declared using `var`, maybe as a tooltip when you hover over the variables.

Summary

Deriving type of a variable or an expression from the context of its use is called type inference. Java 10 introduced a restricted type name called `var` that can be used to declare local variables with initializes, and indexes for `for-each` and `for` loops. The type of the variable declared using `var` is inferred from the expression in the initializer. `var` is not a keyword; rather, it is a restricted type name, which means you cannot have a type such as a class or interface named as `var`. If you have a type named `var` in your existing code, your code will break when compiled in JDK10.

It is not always possible to infer the type of local variables from initializers. For example, if the initializer is `null`, it is not possible to inter the type. There are few limitations for using `var` to declare variables:

- Variable declarations without initializers are not supported.
- The initializer cannot be of the `null` type.
- Multiple variables in a single declaration are not supported.
- The initializer cannot reference the variable being declared.
- Array initializers by themselves are not supported.
- No array dimensions can be specified while declaring arrays.
- No poly expressions such as lambda expressions and method references are allowed in initializer.
- Instance and static fields are not supported.
- `var` is not supported as the return type of a method. You cannot use it to declare method's arguments.