Java v9, 10, and 11

A Selection of Added Features

Extra, Extra! Read all about it!

- You can always check out the (surprisingly readable!) Java Language Specification for details on new additions to Java.

More Resources

- How to Do in Java- Java 9
- DZone- Java 10
- Codete Java 9-11
- Oracle Brief Language Updates 9-11
- Oracle What's New in Java 9
- Programming Notes- Java 9 and Java 10

var: Local Variable Type Inference

Previous Type Inference

- Java types are static.
- In general, Java types must always be explicitly declared.
- But Java does have some type inference:

```
List<String> wordList = new ArrayList<String>();
List<String> wordList = new ArrayList<>(); // as of Java 7

Predicate<String> valid = (String x) -> x.length() > 0; // as of Java 8
Predicate<String> valid = x -> x.length() > 0;
```

Local Variable Type Inference

- As of Java 10, you can use `var` in the declaration of local variables.
- The type of the variable is then inferred from the right-hand side instantiation.
- Examples:

```
var number = 1;    // inferred type of number is int
var name = "Jessica";    // inferred type of name is String
var result = true; // inferred type of result is boolean
```

Local Variable Type Inference

- Although the examples on the previous slide are now allowed, they aren't what var was meant for.
- var is meant to decrease boilerplate code and increase readability.
- Example:

```
HashMap<String, List<Customer>> customerByIDMap = new
                    HashMap<String, List<Customer>>();
```

```
HashMap<String, List<Customer>> customerByIDMap = new HashMap<>();
```

can now be written

```
var customerByIDMap = new HashMap<String, List<Customer>>();
```

Local Variable Type Inference

- var can be used for the following types of variables:
  - Local variable declarations with an initialization
  - Enhanced for-loop (for-each loop) indexes
  - Index variables declared in traditional for loops
  - Try-with-resources variable

Local Variable Type Inference

- When should you use var?
- It's subjective!
- Main rule of thumb: use it to improve readability

Local Variable Type Inference

- Example:

```
ArrayList<String> list = new ArrayList<String>();
Stream<String> stream = list.stream();
```

can now be written

```
var list = new ArrayList<String>();
var stream = list.stream();
```

Local Variable Type Inference

- Example: take a collection of Strings and find the String that occurs most often

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

Local Variable Type Inference

- Example: take a collection of Strings and find the String that occurs most often

```
Map<String, Long> mapCountByString = strings.stream()
        .collect(groupingBy(s -> s, counting()));
Optional<Map.Entry<String, Long>> maxCountStringOpt =
         frequencyMap.entrySet()
        .stream()
        .max(Map.Entry.comparingByValue());
String frequentWord = maxCountStringOpt.map(Map.Entry::getKey);
```

Local Variable Type Inference

- Example: take a collection of Strings and find the String that occurs most often

```
var mapCountByString = strings.stream()
      .collect(groupingBy(s -> s, counting()));
var maxCountStringOpt = mapCountByString.entrySet()
      .stream()
      .max(Map.Entry.comparingByValue());
String frequentWord = maxCountStringOpt.map(Map.Entry::getKey);
```

Restrictions on Using var

- `var` is only for local variables
  - It cannot be used for instance data variables.
  - It cannot be used in formal parameter lists.
- `var` can only be used when you declare and initialize in the same statement.
  - You cannot declare in one place and initialize later.
  - Example: `var num;` is not allowed.
- var cannot be initialized to null.

Restrictions on Using var

- You cannot use for compound declarations
  - var a=1, c=2; // not allowed
- You cannot use with initializer lists
  - var numbers = {1, 2, 3}; // not allowed
  - var numbers = new int[]{1, 2, 3}; // allowed
    - consider: is that better than int[] numbers = {1, 2, 3}?

Caution with Using var

- var list = new ArrayList<>();
  - This is allowed, but creates an ArrayList<Object>
  - This is probably not what you want!
  - When using var with a collection object, make sure to include the collection type on the right hand side.

Caution with Using var

- Inherited types
  - var only allows the declared type to be the child/implementing class, not a parent/interface

```
public class Vehicle
public class Car extends Vehicle
public class Bike extends Vehicle

Vehicle v = new Car();
v = new Bike(); // allowed

var vehicle = new Car();
var bike = new Bike();
vehicle = new Bike(); // not allowed
vehicle = bike; // not allowed
```

var is a Reserved Type Name

- var is not a keyword
- var is a reserved type name
- Existing code that uses var as a variable, method, or package name is not affected.
- Existing code that uses var as a class or interface name is affected.

var in Lambdas

- As of Java 11, you can use var in lambdas.
- Example:

```
IntFunction<Integer> doubleIt1 = (int x) -> x * 2;

IntFunction<Integer> doubleIt2 = (var x) -> x * 2;
```

var in Lambdas

- Example:

```
Comparator<Customer> comparator = …

     (Customer c1, Customer c2) ->
          c1.getName().compareTo(c2.getName());

     (c1, c2) -> c1.getName().compareTo(c2.getName());

     (var c1, var c2) ->
          c1.getName().compareTo(c2.getName());
```

var in Lambdas

- Example:

```
Comparator<Customer> comparator = …

      (@NonNull Customer c1, @NonNull Customer c2) ->

            c1.getName().compareTo(c2.getName());

      (c1, c2) -> c1.getName().compareTo(c2.getName());

      (@NonNull var c1, @NonNull var c2) ->
            c1.getName().compareTo(c2.getName());
```

More Resources

- Excellent cheat sheet full of good var-related programming practices (including a discussion of programming to the interface)
- Oracle overview
- var in Lambdas

Other Goodies

Immutable Collection Factory

- Java 9 introduced static "of" factory methods to create unmodifiable instances of a List, Set, and Map.
- Java 10 introduced static "copyOf" methods to create unmodifiable *copies* of these objects.

Immutable Collection Factory

- Prior to Java 9 and 10:

```
List<String> wordList = new ArrayList<String>();
wordList.add("a");
wordList.add("b");
wordList.add("c");
List<String> unmodifiableViewOfWordList =
      Collections.unmodifiableList(wordList);
wordList.set(2, "z"); // allowed
// both lists objects are changed because it's just a view!

unmodifiableViewOfWordList.set(2, "z");
// not allowed- runtime exception
```

Immutable Collection Factory

- List.copyOf(…) creates an unmodifiable *copy* of an existing collection

```
List<String> wordList = new ArrayList<String>();
wordList.add("a");
wordList.add("b");
wordList.add("c");

List<String> unmodifiableWordList = List.copyOf(wordList);
wordList.set(2, "z"); // allowed
// only wordList is affected!

unmodifiableWordList.set(2, "z");
// not allowed- runtime exception
```

Immutable Collection Factory

- Set.copyOf(…) and Map.copyOf(…) do the same

Immutable Collection Factory

- List.of(…), Set.of(…), and Map.of(…) are quick ways to create an unmodifiable collection

```
List<String> unmodifiableWordList =
      List.of("a", "b", "c");
Set<Integer> unmodifiableNumberSet =
      Set.of(1, 2, 3);
Map<String, Customer> unmodifiableCompanyMap =
      Map.of("Jane", janeCustomer, "Bob", bobCustomer);
```

Private Helper Methods in Interfaces

- As of Java 9, interfaces can contain private (static or instance) methods. (How to Do In Java Overview)
- These methods are meant only as helpers.
  - They allow common code to be captured.
- private interface methods can only be used **inside** the interface.
  - They cannot be abstract.
- private, static methods:
  - can be used inside any other interface methods (static or non-static)
- private, non-static methods:
  - cannot be used inside static methods
  - can only be used inside other non-static methods

Private Helper Methods in Interfaces

- Example:

```
public interface HRProcessor {
	void pay();
	void benefits();

	default void review() {
		System.out.println("Processing review");
	}
}
```

Private Helper Methods in Interfaces

- Example:

```
public interface HRProcessor {
      default void pay() {
            System.out.println("Processing pay");
      }
      default void benefits() {
            System.out.println("Processing benefits");
      }
      default void review() {
            System.out.println("Processing review");
      }
}
```

Private Helper Methods in Interfaces

- Example:

```
public interface HRProcessor {
        default void pay() {
                System.out.println(getActionWord() + " pay");
        }
        default void benefits() {
                System.out.println(getActionWord() + " benefits");
        }
        default void review() {
               System.out.println(getActionWord() + "review");
        }
        private static String getActionWord() {
               return "Processing ";
        }
}
```

JShell REPL

- Java 9 introduced JShell, an interactive REPL (Read-Eval-Print Loop) tool.
- Allows you to quickly try out code, statements, methods, or other program elements.
- Launch with "jshell" command
  - Leave with /exit
- Oracle User Guide

Modules

- Modules were introduced in Java 9.
- A module is a group of closely related packages (a "package of packages")
- A module can also contain resource files, such as xml files, data files, images, etc.
- A module has a descriptor file that describes:
  - dependencies- other modules it depends on
  - public packages (by default, packages are module-private)
  - services offered and consumed
  - reflection permissions

Modules

- As Java has grown larger, modules allow for scaling down applications for smaller devices
- Strong encapsulation
- Detecting missing classes at startup, not runtime

More about Modules

- These are all great introductory resources with walkthroughs.
- Baeldung Guide to Java 9 Modularity
- Jenkov Modules Tutorial
- DZone Java 9 Modules

And the list goes on…

- Java 11: Launch single-file source code (e.g., java HelloWorld.java) (DZone Tutorial)
- Java 11: New String methods: repeat, isBlank, strip, lines (DZone Tutorial)
- Java 9: More concise try-with-resources syntax (JournalDev and TutorialsPoint)
- Java 9: underscore no longer allowed as a valid variable name
  - `int _ = 4; // no longer allowed`