**Java 9**

**Modules**

**Jshell**

**Immutable List, Set and Map (List.of, Set.of, Map.of, Map.** **ofEntries)**

**Stream API Improvements:**

* takeWhile
* dropWhile
* ofNullable
* iterate
* filtering
* flatMapping

**Enhancements to Optional**

* ifPresentOrElse
* of
* stream

**Underscore as identifier illegal**

**Improved try-with-resources**

**Private interface methods**

**Reactive Programming with JDK 9**

Deprecated: Concurrent Mark Sweep (CMS) collector. Default G1 (Garbage first) in JDK9 onwards.

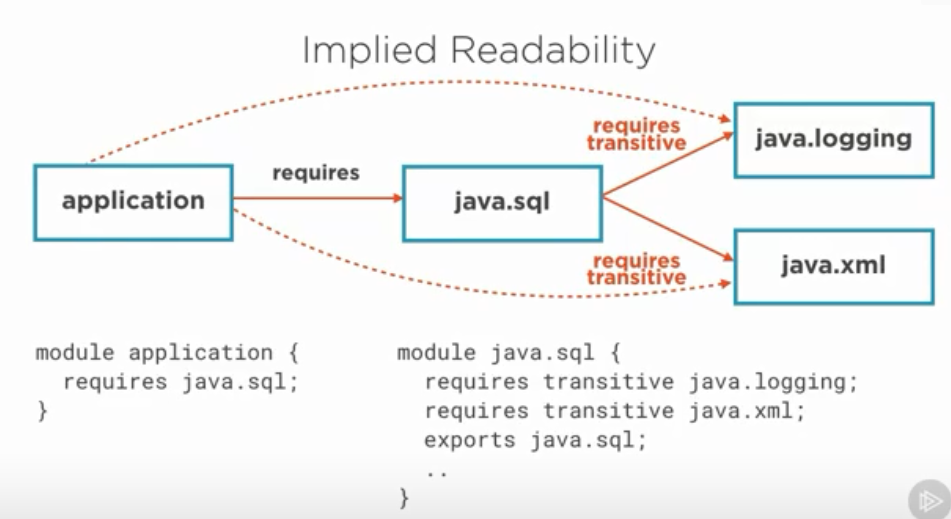
**Module:**

1. Define required statement client module.
2. Export statement server module.
3. Is exported module is public?

Requires transitive

Exposing and consuming services

Linking modules



java --list-modules (displays list of java modules)

java --describe-module java.sql

java.sql@13.0.1

exports java.sql

exports javax.sql

requires java.transaction.xa transitive

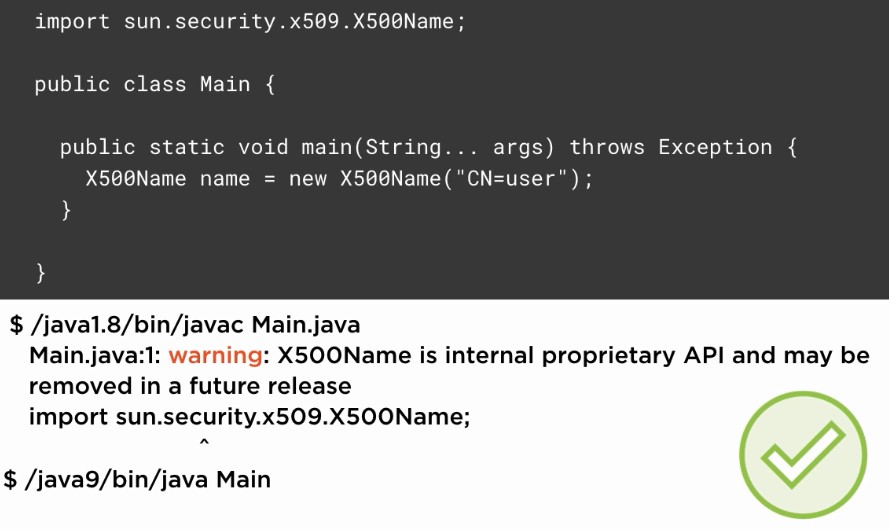
requires java.base mandated

requires java.xml transitive

requires java.logging transitive

uses java.sql.Driver

1. **You can use JDK types that have been encapsulated.**



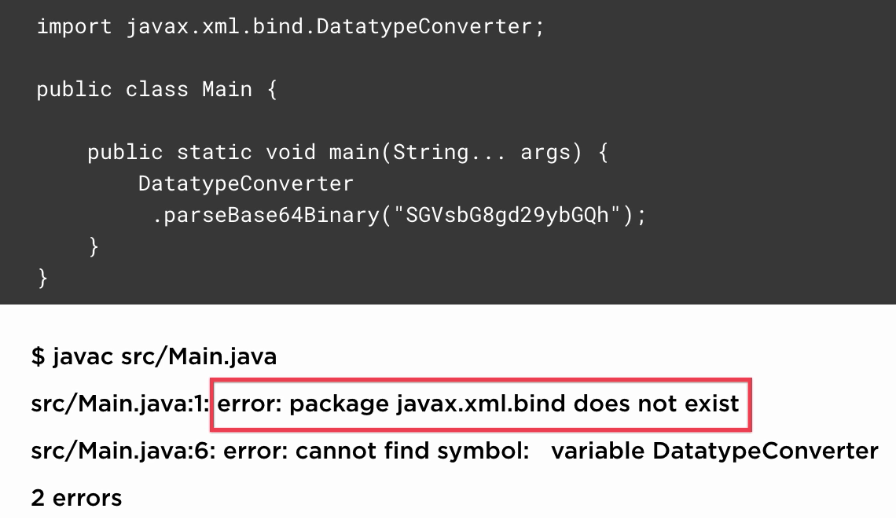


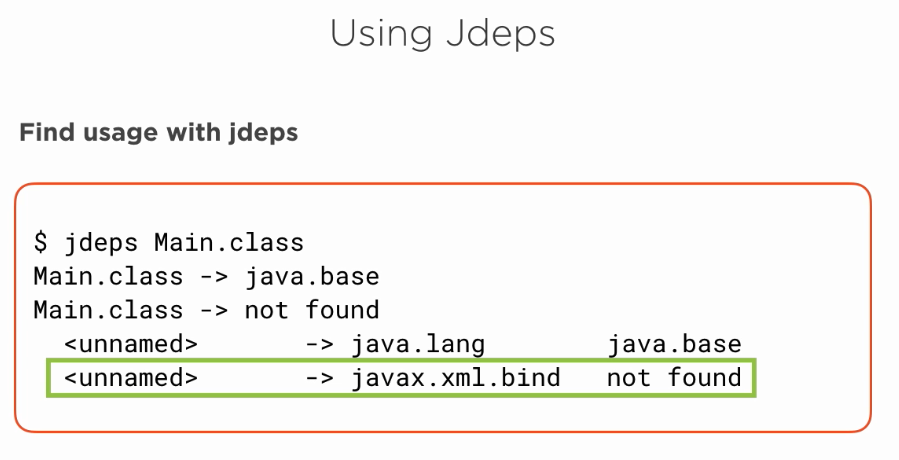
By defult -illegal-access=permit

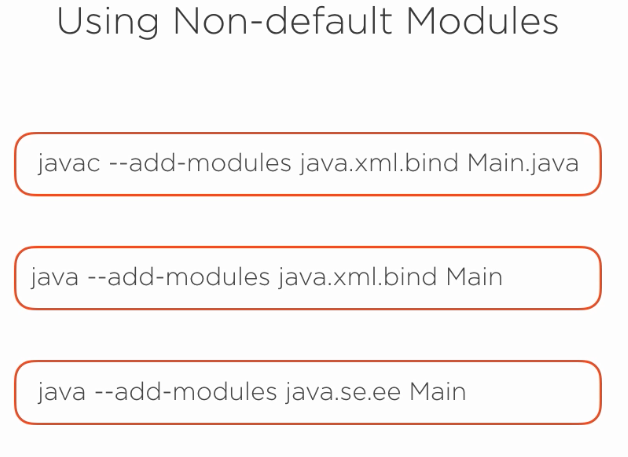


jdeps -jdkinternals single jar name/multiple (this will show whether you are using encapsulated type’s from modules or not, it’s also show suggested public to use)

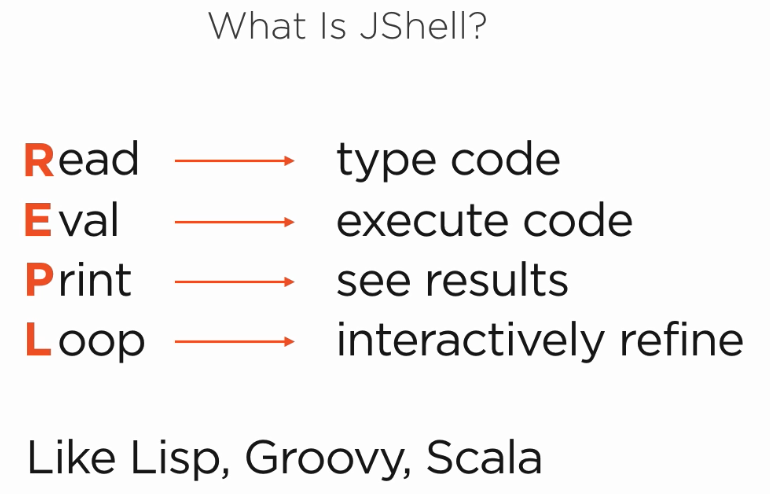
1. **You can use types from non-default Java SE modules**

****

****

****

**JShell:**

****

**Create Immutable List**

Use List.of() static factory methods to create immutable lists. It has following different overloaded versions

static <E> List<E> *of*()  
static <E> List<E> *of*(E e1)  
static <E> List<E> *of*(E e1, E e2)  
static <E> List<E> *of*(E e1, E e2, E e3)

The List instances created by these methods have the following characteristics:

1. These lists are immutable. Elements cannot be added, removed, or replaced in these lists. Calling any mutator method (i.e. add, addAll, clear, remove, removeAll, replaceAll) will always cause UnsupportedOperationException to be thrown.
2. They do not allow null elements. Attempts to add null elements result in NullPointerException.
3. They are [serializable](https://howtodoinjava.com/java/serialization/a-mini-guide-for-implementing-serializable-interface-in-java/) if all elements are serializable.
4. The order of elements in the list is the same as the order of the provided arguments, or of the elements in the provided array.

Lets see few examples of usage of immutable list.

List<String> names = List.*of*("Lokesh", "Amit", "John");

**Create Immutable Set**

Set behave very similar to List with only few differences. e.g.

1. Set do not allow [duplicate elements](https://howtodoinjava.com/puzzles/find-duplicate-elements-in-an-array/) as well. Any duplicate element passed will result in IllegalArgumentException.
2. The iteration order of set elements is unspecified and is subject to change.

All Set factory methods have the same signature as List.

Let’s see few examples of immutable sets.

static <E> Set<E> *of*()  
static <E> Set<E> *of*(E e1)  
static <E> Set<E> *of*(E e1, E e2)  
static <E> Set<E> *of*(E e1, E e2, E e3)

**Create Immutable Map**

Map factory methods are same as List or Set overloaded factory methods. Only difference is that the signatures of the of methods take alternating keys and values as arguments. e.g.

static <K,V> Map<K,V> *of*()  
static <K,V> Map<K,V> *of*(K k1, V v1)  
static <K,V> Map<K,V> *of*(K k1, V v1, K k2, V v2)  
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)  
static <K,V> Map.Entry<K,V> entry(K k, V v)

Let’s take an example of creating immutable Map in java 9.

Map<String, String> names = Map.*ofEntries*(  
 Map.*entry*("1", "Lokesh"),  
 Map.*entry*("2", "Amit"),  
 Map.*entry*("3", "Brian"));

Clearly, new **factory methods to create immutable collections in java 9** are very much readable and easy to use.

**Stream API Improvements:**

In Java SE 9, Oracle Corp has added the followng four useful new methods to java.util.Stream interface.

* takeWhile
* dropWhile
* ofNullable
* iterate

Java 9 added 2 new collectors, filtering and flatMapping, which play nice with the groupingBy collector.

takeWhile:

default Stream<T> takeWhile(Predicate<? super T> predicate);

takeWhile is similar to filter in the sense that it expects a predicate and returns a new stream consisting only of the elements that match the given predicate. But there’s a catch. In an ordered stream, takeWhile takes elements from the initial stream while the predicate holds true. Meaning that when an element is encountered that does not match the predicate, the rest of the stream is discarded.

Stream.of(2, 4, 6, 8, 9, 10, 12).takeWhile(n -> n % 2 == 0)

.forEach(System.out::println);

// prints out: 2 4 6 8

dropWhile:

default Stream<T> dropWhile(Predicate<? super T> predicate);

dropWhile is essentially the opposite of takeWhile. Instead of taking elements from the stream until the first element which does not match the predicate, dropWhile drops these elements and includes the remaining elements in the returned stream.

Stream.of(2, 4, 6, 8, 9, 10, 12).dropWhile(n -> n % 2 == 0)

.forEach(System.out::println);

// prints out:9 10 12

**Unordered streams:**

So far we’ve looked at how takeWhile and dropWhile behave with ordered streams. But what happens if the stream is unordered? According to the [docs](http://download.java.net/java/jdk9/docs/api/java/util/stream/Stream.html), if some of the elements in the stream match the predicate (but not all) then the operation is nondeterministic and an arbitrary subset of matching elements is returned or removed. Meaning that you’ll get different results for each execution.

Set<Integer> numbers = Set.of(2, 4, 6, 3, 8);

numbers.stream().takeWhile(n -> n % 2 == 0)

.forEach(System.out::println);

// prints out a different subset of matching elements every time

// an empty set is also a subset

You can expect a similar behaviour if you replace takeWhile with dropWhile. It’s free to drop any subset of matching elements. That includes the empty set.

**All elements match:**

Regardless of whether the stream is ordered or unordered, if all elements match the given predicate then takeWhile takes and dropWhile drops all elements. The result of takeWhile is the same as the input stream. On the other hand, when all elements match, the result of dropWhile will be an empty stream.

The following is an example of takeWhile applied on an unordered stream where all the elements match the predicate.

Set<Integer> numbers = Set.of(2, 4, 6, 8);

numbers.stream().takeWhile(n -> n % 2 == 0)

.forEach(System.out::println);

// always prints out 2, 4, 6, 8

// the order of course is nondeterministic because the stream is unordered

**No elements match:**

I bet you can already guess what happens if no elements match the given predicate. You’re right if you guessed that the result of takeWhile will be an empty stream. Since no elements matched, there’s nothing to take. dropWhile, on the other hand, returns the input stream if there’s nothing to drop.

Stream.of(2, 4, 6, 8).dropWhile(n -> n % 2 != 0)

.forEach(System.out::println);

// prints out: 2 4 6 8

**ofNullable():**

Until Java 8, you cannot have null value in a stream. It would have caused

NullPointerException.

Stream.ofNullable() takes an element and produces a Stream of single element if the specified element is non-null, otherwise an empty Stream. The main intention is to avoid NullPointerExceptions and to avoid having null checks everywhere.

// Produces a Stream of Single element

Stream.ofNullable("Hello").forEach(System.out::println)

Hello

// Produces an empty Stream

Stream.ofNullable(null).forEach(System.out::println)

*getEmployeeName*(null).forEach(System.*out*::println); // Empty  
*getEmployeeName*(new Book(1,List.*of*("muni","swamy"))).forEach(System.*out*::println);

private static Stream<String> getEmployeeName(Book book){  
 return Stream.*ofNullable*(book).flatMap(e -> e.getAuthors().stream());  
}

**Stream.iterate():**

iterate() methods used for creating a stream which starts with a single element (the seed), and subsequent elements are produced by successively applying the unary operator. The result is an infinite stream. To terminate the stream, a limit or some other short-circuiting function, like findFirst or findAny is used.

The iterate method in Java 8 has the signature:

static Stream iterate(final T seed, final UnaryOperator f)

In Java 9, new overloaded version of iterate takes a Predicate as the second argument:

static Stream iterate(T seed, Predicate hasNext, UnaryOperator next)

Let’s see the difference is use of iterate method from java 8 to java 9.

**iterate method in Java 8**

List<Integer> numbers = Stream.*iterate*(1, i -> i+1)  
 .limit(10)  
 .collect(Collectors.*toList*());  
System.*out*.println(numbers);  
Output:  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

**iterate method in Java 9**

List<Integer> numbers = Stream.*iterate*(1, i -> i <= 10 ,i -> i+1)  
 .collect(Collectors.*toList*());  
System.*out*.println(numbers);  
Output:  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

**Collectors.filtering():**

Similarly to the filter method on streams, the filtering collector is used for filtering elements in a stream. The filter method, however, processes the values before they’re grouped, whereas Collectors.filtering can be used nicely with Collectors.groupingBy to group the values before the filtering step takes place.

List<Integer> numbers = List.*of*(2, 3, 4, 7, 9, 11);  
// Java 8  
numbers.stream().filter(j -> j > 5)

.collect(Collectors.*groupingBy*(i -> i % 2, Collectors.*counting*()));  
// {1=3}

The result correctly says that we have 3 odd numbers greater than 5 in our list. However, it doesn’t tell us we don’t have any even numbers there. The even numbers were filtered out before aggregation, and any trace of them is lost. That’s exactly what the filtering collector addresses. In Java 9, we can count the numbers in both bags as follows:

// Java 9 Way  
Map<Integer, Long> case5 = numbers.stream().collect(Collectors.*groupingBy*(i -> i % 2, Collectors.*filtering*(j -> j > 5, Collectors.*counting*())));  
System.*out*.println("Case 5: "+case5);  
// {0=0, 1=3}

**Collectors.flatMapping():**

Unlike Collectors.mapping, however, Collectors.flatMapping deals with a stream of elements, which allows us to get rid of often unnecessary intermediary collections.

Now suppose we want to aggregate this data, e.g. group the strings by the integers they’re associated with. Using Java 8 constructs, we would accomplish this as follows:

Stream<Map.Entry<Integer,Set<String>>> entries = Stream.*of*(Map.*entry*(1, Set.*of*("a", "b")), Map.*entry*(1, Set.*of*("a", "c")), Map.*entry*(2, Set.*of*("d")));  
entries.collect(Collectors.*groupingBy*(e -> e.getKey(), Collectors.*mapping*(e -> e.getValue(), Collectors.*toSet*())));

// {1=[[b, a], [c, a]], 2=[[d]]}

Although this is technically correct, we end up with nested collections, which we would need to further unwrap to, for example, deal with duplicates. Fortunately, that’s exactly what the new flatMapping collector is for:

entries.collect(Collectors.*groupingBy*(e -> e.getKey(), Collectors.*flatMapping*(e -> e.getValue().stream(), Collectors.*toSet*())));  
// {1=[a, b, c], 2=[d]}

**ifPresentOrElse(*Consumer*, *Runnable*)**

The new [ifPresentOrElse](https://docs.oracle.com/javase/9/docs/api/java/util/Optional.html#ifPresentOrElse-java.util.function.Consumer-java.lang.Runnable-) method allows you to perform one action if the Optional is present and a different action if the Optional is not present. This new method performs the given Consumer action if a value is present, otherwise runs the given Runnable action. It’s replacement for ifPresent and else.

IntStream.*of*(1, 2, 4)  
 .filter(i -> i % 3 == 0)  
 .findFirst()  
 .ifPresentOrElse(System.*out*::println, () -> {  
 System.*out*.println("No multiple of 3 found");  
 });

The new [stream](https://docs.oracle.com/javase/9/docs/api/java/util/Optional.html#stream--) method makes it easier to convert a stream of Optional objects into a stream of values that are present in them.

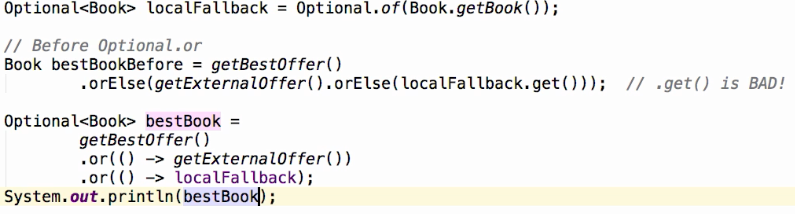
**or(Supplier)**

The [or](https://docs.oracle.com/javase/9/docs/api/java/util/Optional.html#or-java.util.function.Supplier-) method is somewhat similar to the [orElseGet](https://docs.oracle.com/javase/9/docs/api/java/util/Optional.html#orElseGet-java.util.function.Supplier-) method but returns Optional objects instead of values. If a value is present, it returns the existing Optional. If the value is not present, it returns the Optional produced by the supplying function.

In Java SE 9 Optional API, or() method is used to return a value, if Optional contains a value. Otherwise returns a value specified in the Supplier. This or() method takes a Supplier as an argument to specify a default value.

public Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)

Optional<String> opStr = Optional.*empty*();  
Supplier<Optional<String>> supStr = () -> Optional.*of*("No Name");  
opStr.or(supStr);  
// No Name

****

.get() method may throw NPE.

**Stream()**

The new [stream](https://docs.oracle.com/javase/9/docs/api/java/util/Optional.html#stream--) method makes it easier to convert a stream of Optional objects into a stream of values that are present in them. Previously (in Java 8), you needed two steps in order to achive this. First, you would filter out the empty Optionals and then you would unbox the rest in order to get their values.

// In Java 8:  
Stream.*of*("alice", "bob", "charles")  
 .map(UserDirectory::lookup)  
 .filter(Optional::isPresent)  
 .map(Optional::get)  
 .collect(toList());

In Java 9, the code becomes simpler using the [stream](https://docs.oracle.com/javase/9/docs/api/java/util/Optional.html#stream--) method:

// In Java 9:  
Stream.*of*("alice", "bob", "charles")  
 .map(UserDirectory::lookup)  
 .flatMap(Optional::stream)  
 .collect(toList());

**Underscore as identifier illegal**

String \_ = "muni";

As of Java 9, '\_' is a keyword, and may not be used as an identifier

**Improved try-with-resources**

even though we have reader1 referring to BufferedReader object, we should create a duplicate one “reader2” to use it in Try-With-Resources. It is one small bug or issue in Java SE 7 or 8 versions.

We cannot use any Resource (which is declared outside the Try-With-Resources) within try() block of Try-With-Resources statement.

// Java 8

void testJava9(BufferedReader reader1) throws IOException {  
 try (BufferedReader reader2 = reader1) {  
 System.*out*.println(reader2.readLine());  
 }  
}

// Java 9

void testJava9(BufferedReader reader) throws IOException {  
 try (reader) {  
 System.*out*.println(reader.readLine());  
 }  
}

**Private interface methods**

public interface Java8Methods {  
 double getPrice();  
 default double getPriceWithTax(){  
 return getPrice() \* 1.21;  
 }  
 default double getPriceWithTax(double discount){  
 return getPrice() \* 1.21 \* discount;  
 }  
}

public interface Java9PrivateMethods {  
 double getPrice();  
  
 default double getPriceWithTax(){  
 return getTaxPriceInternal();  
 }  
 default double getPriceWithTax(double discount){  
 return getTaxPriceInternal() \* discount;  
 }  
 // Java 9 private method

// and abstract the private helper methods from child class  
 private double getTaxPriceInternal(){  
 return getPrice() \* 1.21;  
 }  
}

**Java 10**

**Local Variable Type Inference:**

var str = "muni"; // type will get from right side of expression  
var var = "var"; // compiles  
// var is not a keyword

Type inference can’t applicable for method parameters, return types and catch blocks.

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

// type will get from left side.  
Predicate<String> string = str1 -> str1.length() > 3;

// type will get from left side.

List.copyOf();