Java language evolution (JDK 7 – 10)

Produced by:

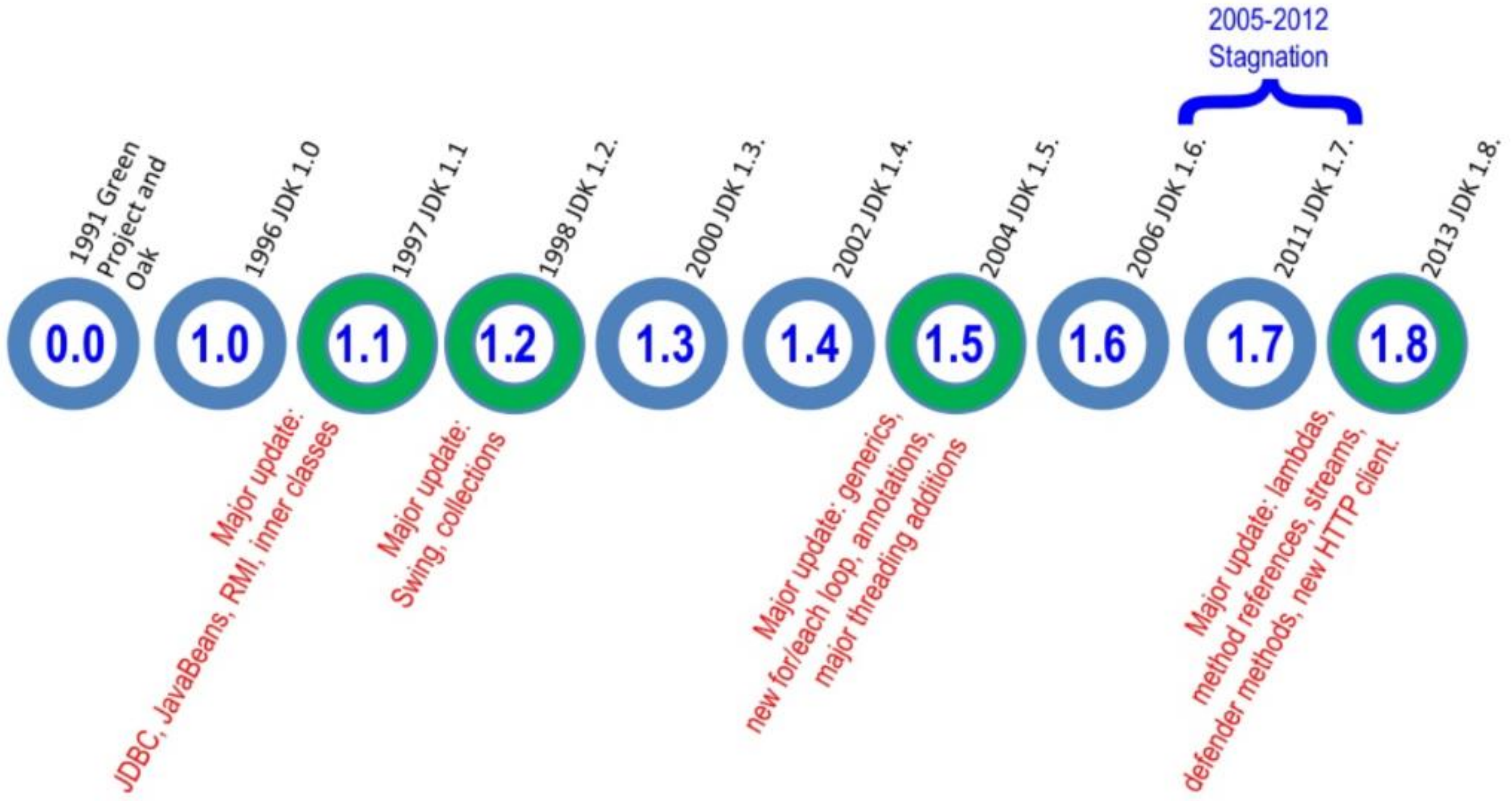
Dr. Siobhán Drohan (sdrohan@wit.ie)

Eamonn de Leastar (edeleastar@wit.ie)

Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Java SE Version History (Green: Major; Blue: Minor)



2005-2012
Stagnation

1991 Green Project and Oak

0.0

1996 JDK 1.0

1.0

Major update:
JDBC, JavaBeans, RMI, inner classes

1997 JDK 1.1

1.1

Major update:
Swing, collections

1998 JDK 1.2.

1.2

2000 JDK 1.3.

1.3

2002 JDK 1.4.

1.4

Major update: generics,
new for/each loop, annotations,
major threading additions

2004 JDK 1.5.

1.5

2006 JDK 1.6.

1.6

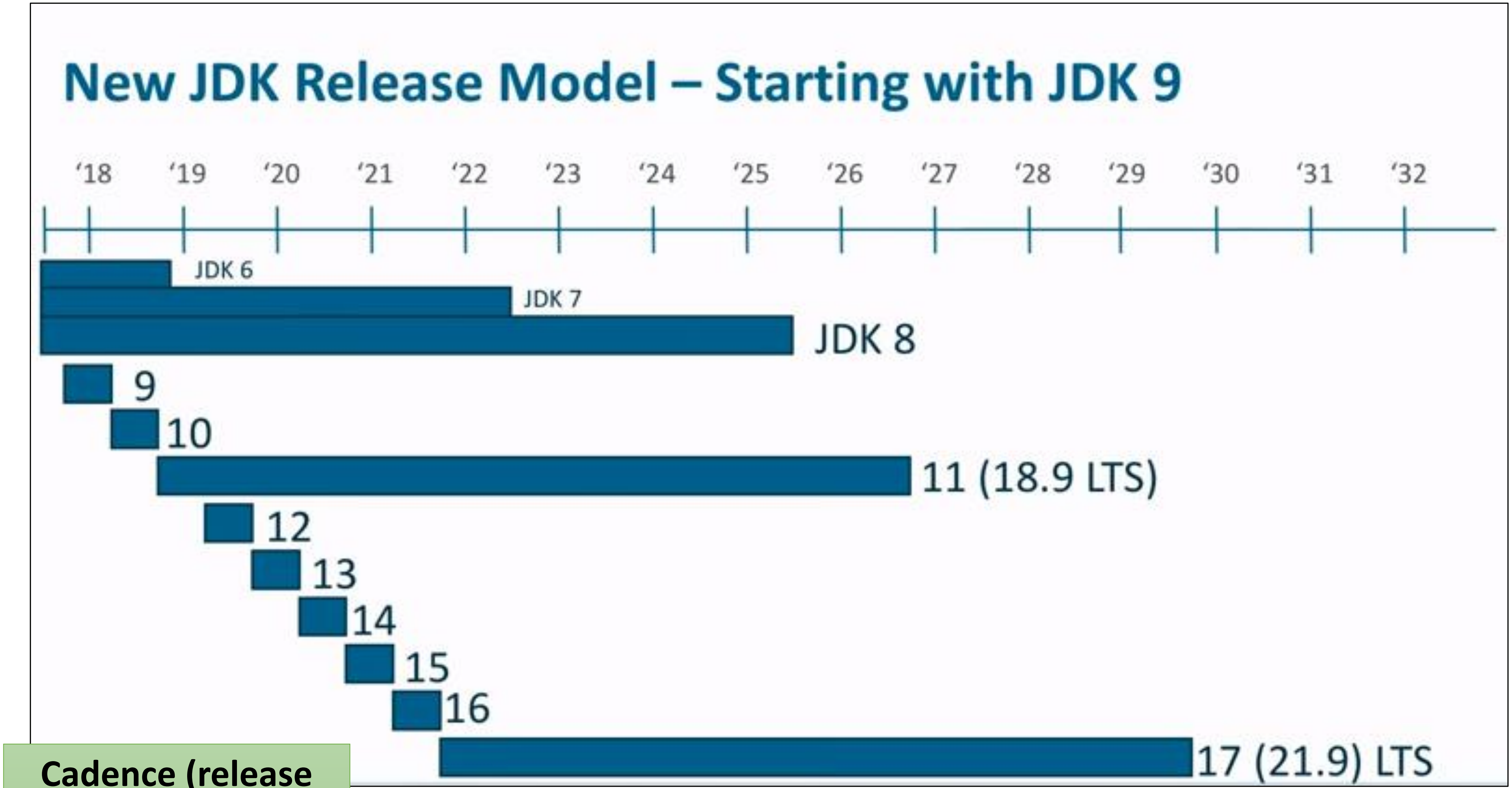
Major update: lambdas,
method references, streams,
defender methods, new HTTP client.

2011 JDK 1.7.

1.7

2013 JDK 1.8.

1.8

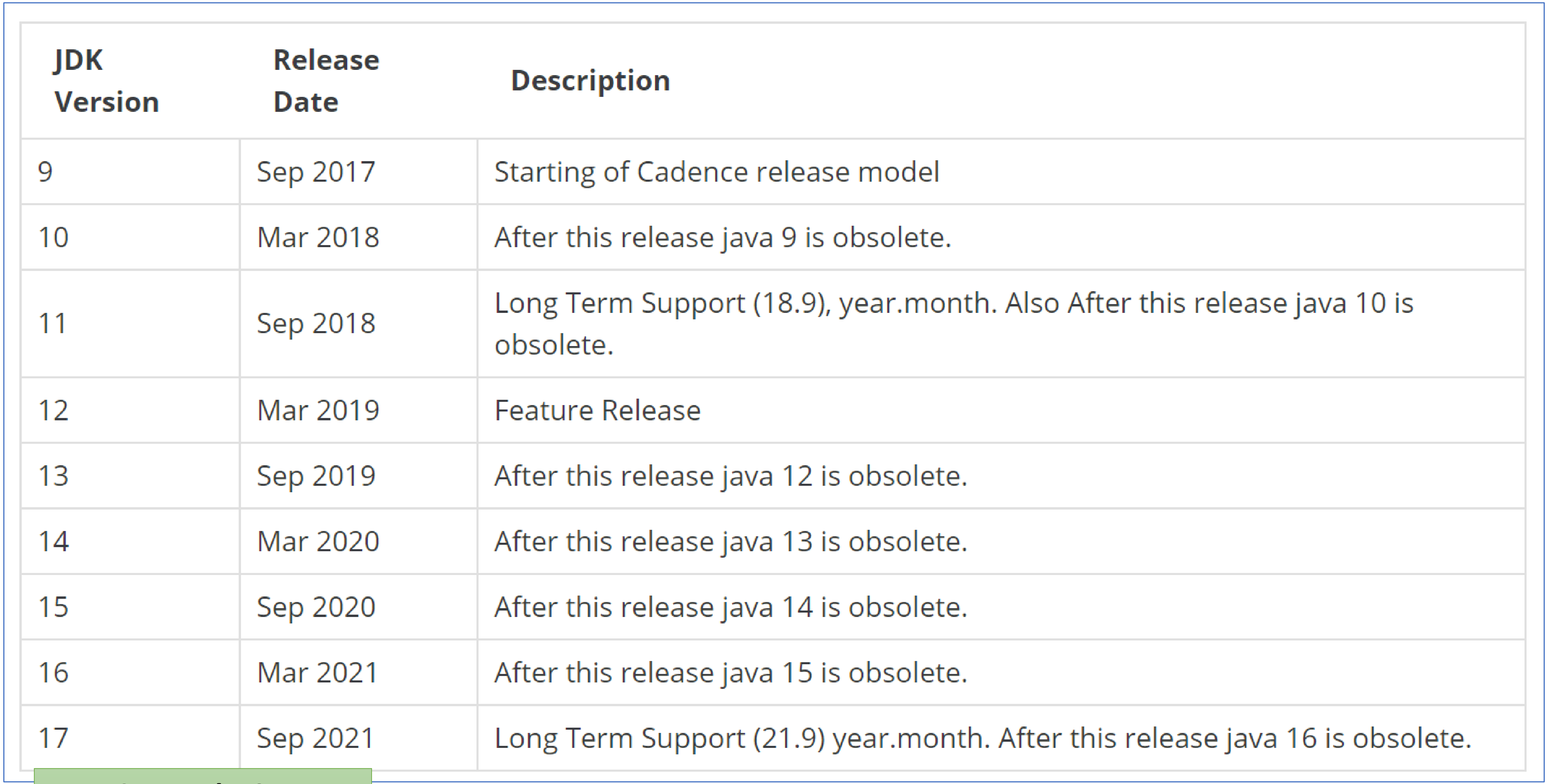


New JDK Release Model – Starting with JDK 9

Cadence (release frequency) is accelerating → two releases per year.

Longterm support for Java 8, 11 and 17.
Oracle initially were aiming for a model of longterm support for one-in-three versions.

https://www.rushis.com/java-release-model/



JDK Version	Release Date	Description
9	Sep 2017	Starting of Cadence release model
10	Mar 2018	After this release java 9 is obsolete.
11	Sep 2018	Long Term Support (18.9), year.month. Also After this release java 10 is obsolete.
12	Mar 2019	Feature Release
13	Sep 2019	After this release java 12 is obsolete.
14	Mar 2020	After this release java 13 is obsolete.
15	Sep 2020	After this release java 14 is obsolete.
16	Mar 2021	After this release java 15 is obsolete.
17	Sep 2021	Long Term Support (21.9) year.month. After this release java 16 is obsolete.

Cadence (release frequency) is accelerating → two releases per year.

Longterm support for Java 8, 11 and 17.
Oracle initially were aiming for a model of longterm support for one-in-three versions.

https://www.rushis.com/java-release-model/

Release Date:
2011

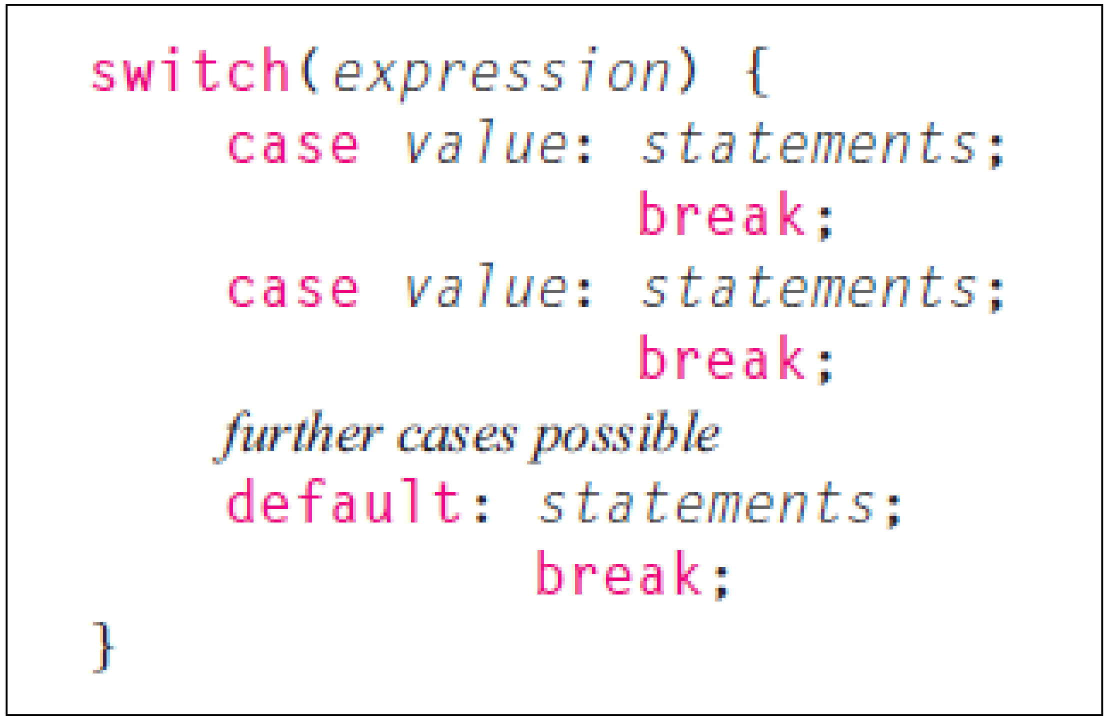
https://docs.oracle.com/javase/7/docs/index.html

An outline of some changes

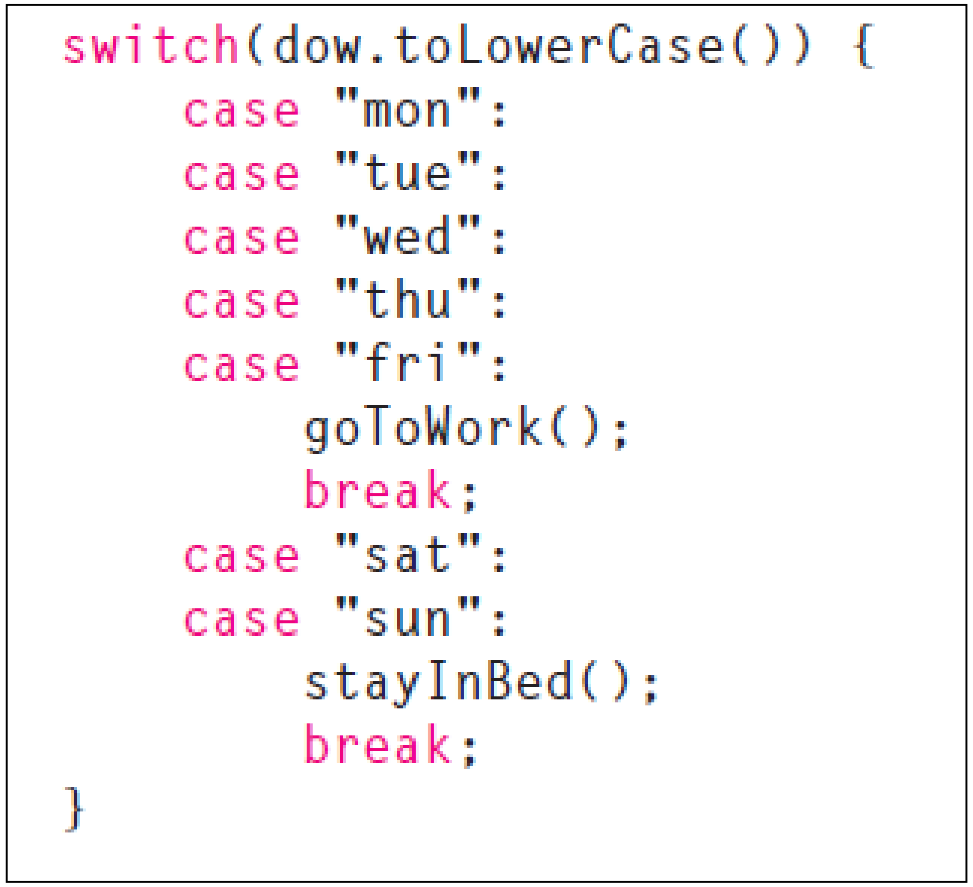
1. Can now switch on Strings
2. Inclusion of try-with-resources
3. Multi-catch
4. Improved type inference
5. More new I/O APIs for the Java platform

RECAP: switch control statement

Pre Java 7: can switch on int and char.
Post Java 7: can also switch on String

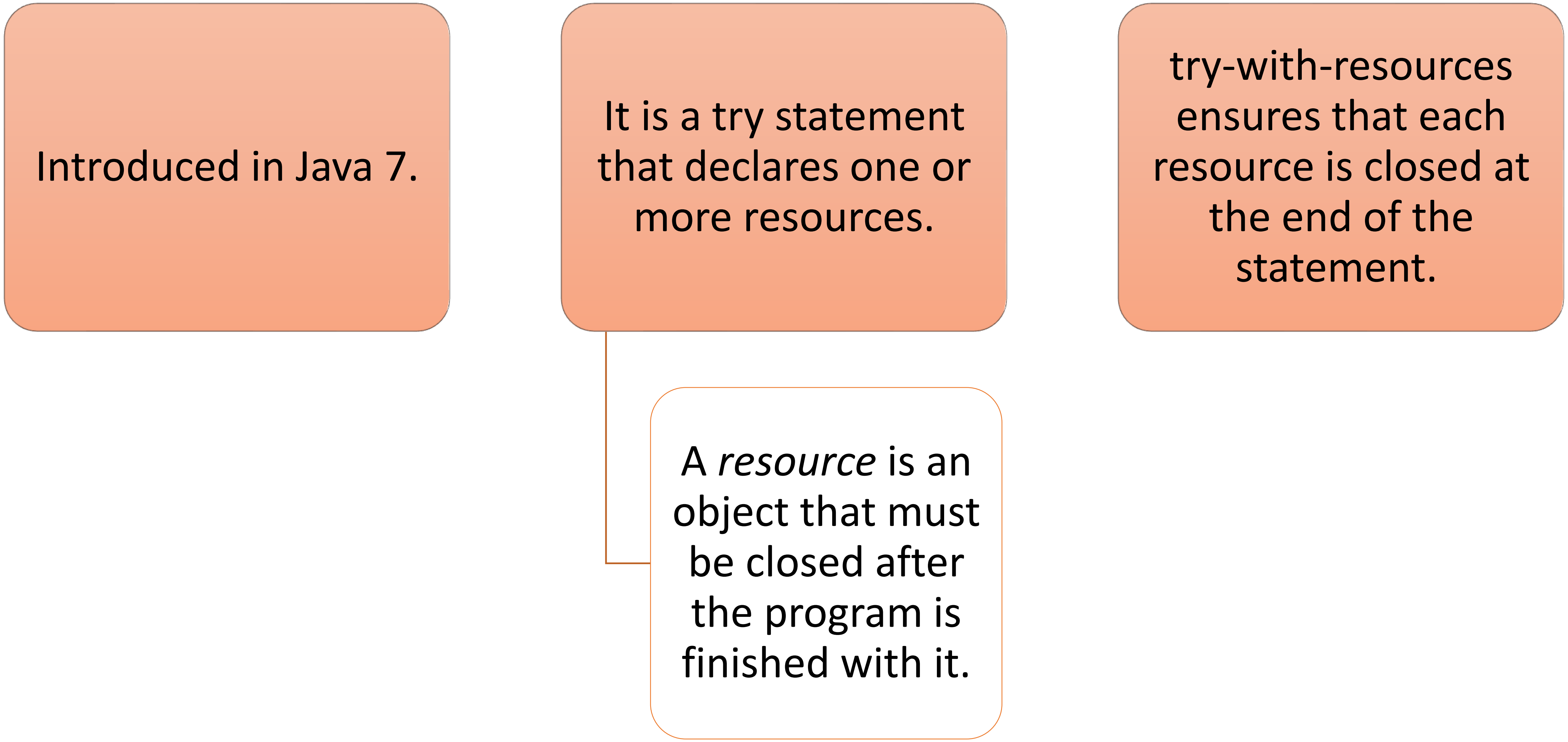


```
switch(expression) {
    case value: statements;
                break;
    case value: statements;
                break;
    further cases possible
    default: statements;
             break;
}
```



```
switch(dow.toLowerCase()) {
    case "mon":
    case "tue":
    case "wed":
    case "thu":
    case "fri":
        goToWork();
        break;
    case "sat":
    case "sun":
        stayInBed();
        break;
}
```

RECAP: try-with-resources



Introduced in Java 7.

It is a try statement that declares one or more resources.

A *resource* is an object that must be closed after the program is finished with it.

try-with-resources ensures that each resource is closed at the end of the statement.

RECAP: try-with-resources

```
static String readFirstLineFromFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    //try with a finally block, pre Java 7.
    try {
         return br.readLine();
    }
    finally {
        if (br != null)
            br.close();
   }
}
```

```
static String readFirstLineFromFile(String path) throws IOException {
    //try-with-resources, Java 7.  br will be closed regardless of
   //whether the try statement completes normally or abruptly
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
         return br.readLine();
     }
}
```

A try-with-resources statement can have catch and finally blocks just like an ordinary try statement.

In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

All classes implementing the java.lang.AutoCloseable interface can be used inside the try-with-resources construct.

RECAP: try-with-resources

RECAP: multiple exception handling

Since Java 7, a single catch block can handle more than one type of exception, separated by a vertical bar (|).

This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

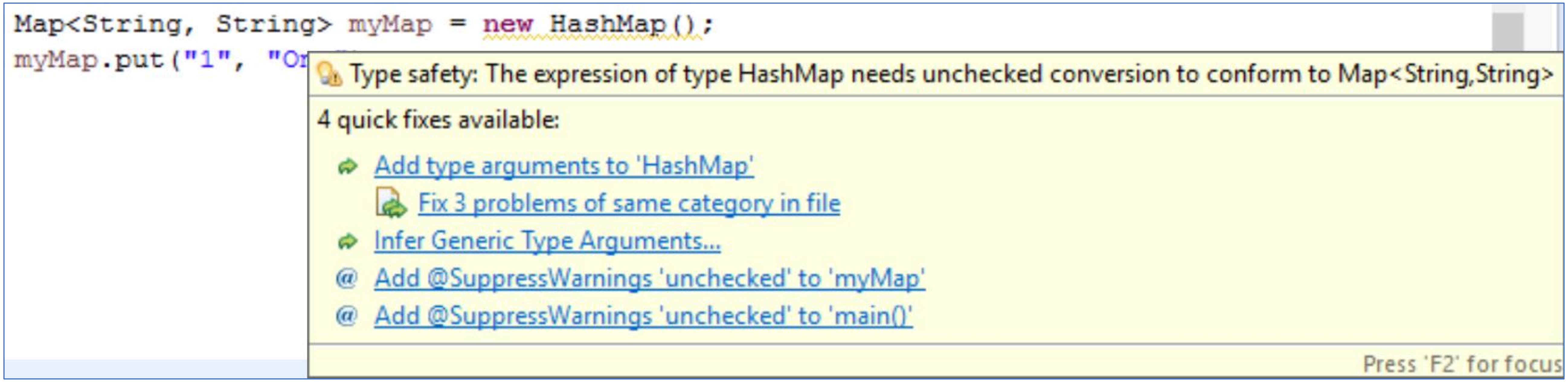
```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

RECAP: type inference

Since Java 7, type inference applies to collections (<>) i.e.:

```
Map<String, String> myMap = new HashMap<>();
```

<> is required.



```
Map<String, String> myMap = new HashMap();
myMap.put("1", "Or
```

Type safety: The expression of type HashMap needs unchecked conversion to conform to Map<String,String>

4 quick fixes available:

- Add type arguments to 'HashMap'
  - Fix 3 problems of same category in file
- Infer Generic Type Arguments...
- Add @SuppressWarnings 'unchecked' to 'myMap'
- Add @SuppressWarnings 'unchecked' to 'main()'

Press 'F2' for focus

Most important package:

java.nio.file which contains many practical file utilities, new file I/O related classes and interfaces.

We will briefly look at:

java.nio.file.Path (interface)
java.nio.file.Files (class)

More new I/O APIs for Java

java.nio.file.Path (interface)

- A Java Path instance represents a *path* in the file system e.g. an absolute or relative file and/or directory.
- This interface can be used in place of the java.io.File class.

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathExample {
        public static void main(String[] args) {

                Path path = Paths.get("c:\\data\\myfile.txt");
        }
}
```

http://tutorials.jenkov.com/java-nio/path.html

java.nio.file.Files (class)

- This class consists exclusively of static methods that operate on files, directories, or other types of files.

- > 50 utility methods for File related operations which many developers would have wanted to be a part of earlier Java releases e.g.:

  copy() – copy a file, with options e.g. REPLACE_EXISTING.
  move() – move or rename a file to a target file.
  newInputStream() – Opens a file, returning an input stream to read from the file.
  readAllBytes() – Reads all the bytes from a file.

https://jaxenter.com/java-7-the-top-8-features-103625.html

Release Date:
2013

https://docs.oracle.com/javase/8/

An outline of some changes

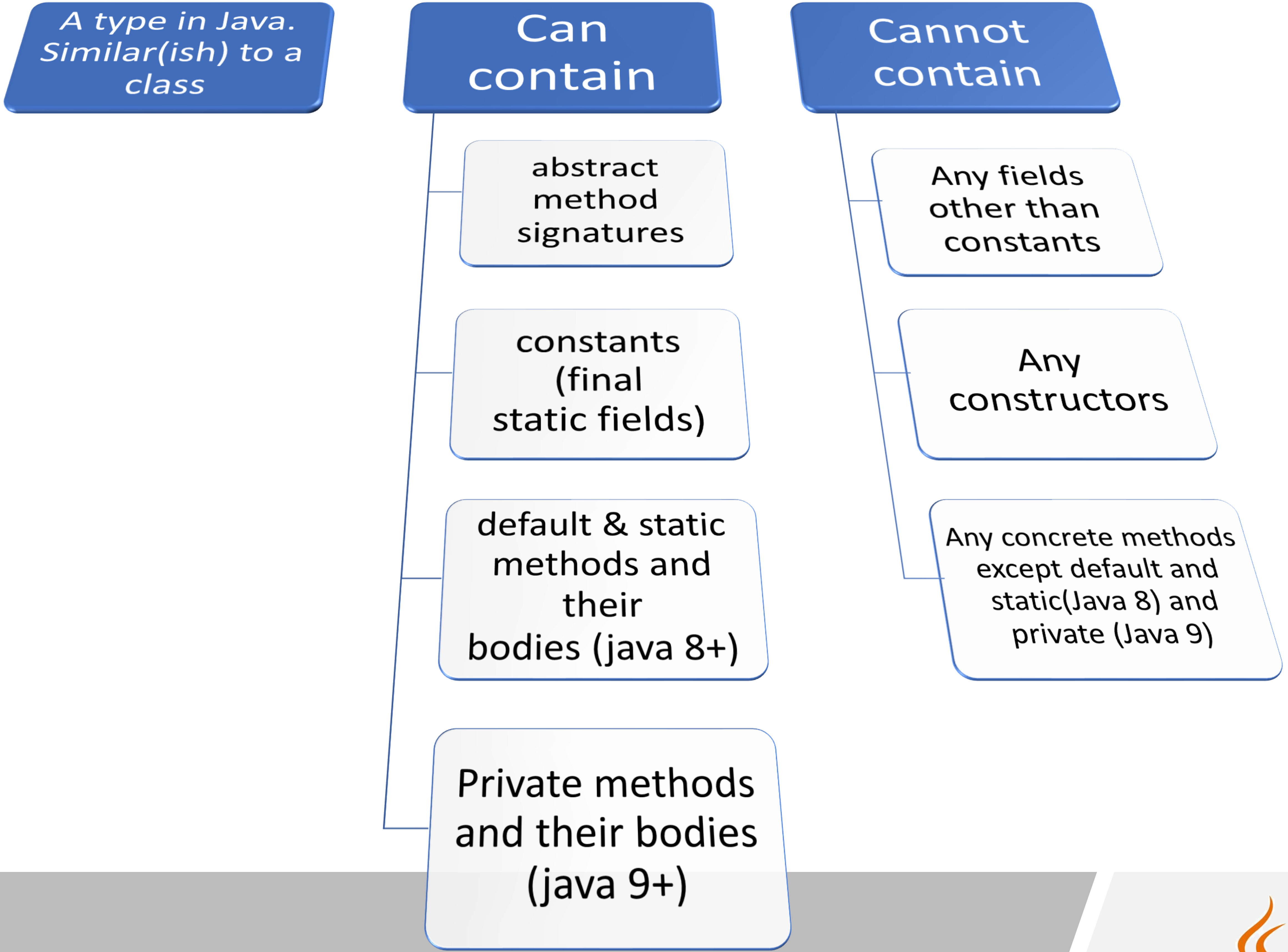
1. Interfaces – default and static methods

2. Lambdas

3. Stream collection types (and new method reference, ::)

4. Date/time improvements

5. Optionals



A type in Java. Similar(ish) to a class

Can contain

- abstract method signatures
- constants (final static fields)
- default & static methods and their bodies (java 8+)
- Private methods and their bodies (java 9+)

Cannot contain

- Any fields other than constants
- Any constructors
- Any concrete methods except default and static(Java 8) and private (Java 9)

RECAP: what is an interface?

1. Defining Interfaces (JDK 7)

Only abstract methods

IAddressBook.java

```
public interface IAddressBook
{
  void clear();

  IContact getContact(String lastName);

  void addContact(IContact contact);

  int numberOfContacts();

  void removeContact(String lastName);

  String listContacts();
}
```

Methods are implicitly public and abstract

1. Defining Interfaces (JDK 8)

Can include default methods

Java 8 introduced **default methods** as a way to extend Interfaces in a backward compatible way.

They can be overridden in implementation classes.

```
public interface IAddressBook
{
  void clear();

  IContact getContact(String lastName);

  void addContact(IContact contact);

  int numberOfContacts();

  void removeContact(String lastName);

  String listContacts();

 default String typeOfEntity(){
      return “Address book”;
  }
}
```

IAddressBook.java

1. Defining Interfaces (JDK 8)

Can include static methods

Java 8 allows **static methods** as a way to organise utility methods in a convenient location.

They <u>cannot</u> be overridden in implementation classes.

```
public interface IAddressBook{
  static final int CAPACITY= 1000;

  void clear();
  IContact getContact(String lastName);
  void addContact(IContact contact);
  int numberOfContacts();
  void removeContact(String lastName);
  String listContacts();

  default String typeOfEntity(){
       return “Address book”;
  }
  static int getCapacity(){
       return CAPACITY;
  }
}
```

IAddressBook.java

- Java's first step into **functional** programming.
- A Lambda is a **function** which can be created *without belonging to any class*.
- Can be passed around as if it was an object and executed on demand.

2. Lambdas – new in JDK 8

http://tutorials.jenkov.com/java/lambda-expressions.html

In Java, **anonymous inner classes** provide a way to implement classes that may occur only once in an application.

Rather than writing a separate event-handling class for each event, you can write something like this.

```
JButton testButton = new JButton("Test Button");

testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent e){
        System.out.println("Click Detected by Anon Class");
  }
});
```

2. Lambdas - Anonymous Inner Classes

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

The code that defines the ActionListener is a **functional interface** i.e. one abstract method.

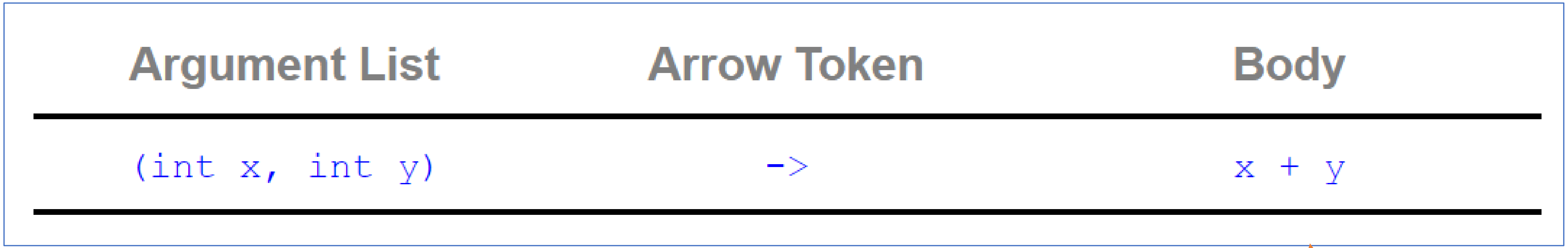
```
package java.awt.event;
import java.util.EventListener;

public interface ActionListener extends EventListener {

    public void actionPerformed(ActionEvent e);

 }
```

```
JButton testButton = new JButton("Test Button");

testButton.addActionListener(new ActionListener(){
     @Override public void actionPerformed(ActionEvent e){
             System.out.println("Click Detected by Anon Class");
  }
});
```

2. Lambdas – Functional Interfaces

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

2. Lambdas – Syntax



Argument List	Arrow Token	Body
`(int x, int y)`	`->`	`x + y`

Expression takes two integer arguments, named x and y, and uses the expression form to return x+y.

Can be *either* a single expression or a statement block.

Body is **evaluated** and **returned**.

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

```
JButton testButton = new JButton("Test Button");

testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent e){
          System.out.println("Click Detected by Anon Class");
  }
});
```

becomes

```
testButton.addActionListener(e ->
System.out.println("Click Detected by Lambda
Listner"));
```

2. Lambdas – Example

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

A stream represents a sequence of objects from an input source, which supports aggregate operations e.g.

- filter
- reduce
- find
- match
- etc..

A stream takes, as input:

- collections
- arrays
- I/O sources

3. Stream

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – for each

- Stream has provided a new method ‘forEach’ to iterate each element of the stream.
- The following code segment shows how to print 10 random numbers using **forEach** over an **IntStream**.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

ints() returns an unlimited IntStream of random int values.

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – for each

- Stream has provided a new method ‘forEach’ to iterate each element of the stream.
- The following code segment shows how to print 10 random numbers using **forEach** over an **IntStream**.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

limit(10) returns an IntStream with 10 entries.

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – for each

- Stream has provided a new method ‘forEach’ to iterate each element of the stream.
- The following code segment shows how to print 10 random numbers using **forEach** over an **IntStream**.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

forEach() performs an action for each element in the IntStream

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – for each

- Stream has provided a new method ‘forEach’ to iterate each element of the stream.
- The following code segment shows how to print 10 random numbers using **forEach** over an **IntStream**.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

Method reference (::) here refers to the static method **println** within the containing class. More information here: https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – map

- The ‘map’ method is used to map each element to its corresponding result.
- This code prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
        numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

stream() returns a sequential Stream of the numbers collection.

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – map

- The ‘map’ method is used to map each element to its corresponding result.
- This code prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
        numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

map() returns a Stream consisting of the results of applying the given function to the elements of the numbers collection.

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – map

- The ‘map’ method is used to map each element to its corresponding result.
- This code prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
        numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

map() returns a Stream consisting of distinct elements in the Stream *(uses Objects.equals(Object)*).

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – map

- The ‘map’ method is used to map each element to its corresponding result.
- This code prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
        numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

collect() returns a **mutable** list of the elements in the Stream.

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – filter

- The ‘filter’ method is used to eliminate elements based on a criteria.
- The following code segment prints a count of empty strings using filter.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```

stream() returns a sequential Stream of the strings collection

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – filter

- The ‘filter’ method is used to eliminate elements based on a criteria.
- The following code segment prints a count of empty strings using filter.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```

filter() returns a Stream consisting of the elements that match the predicate (i.e. are empty).

https://www.tutorialspoint.com/java8/java8_streams.htm

3. Stream – filter

- The ‘filter’ method is used to eliminate elements based on a criteria.
- The following code segment prints a count of empty strings using filter.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```

count() returns an int representing the number of elements in the Stream.

https://www.tutorialspoint.com/java8/java8_streams.htm

Old Date/Time API (java.util.Date):

- **Not thread safe** – not thread safe → developers had to deal with concurrency issues.
- **Poor design** – Default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.
- **Difficult time zone handling** – Developers had to write a lot of code to deal with timezone issues. The new API keeps domain-specific design in mind.

4. Date/time improvements

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

New Date/Time API (java.time):

- **Local** – Simplified date-time API with no complexity of timezone handling.
- **Zoned** – Specialized date-time API to deal with various timezones.
- **Joda** – based on the Joda component’s approach.

4. Date/time improvements

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

```
// Get the current date and time
LocalDateTime currentTime = LocalDateTime.now();    System.out.println("Current
DateTime: " + currentTime);

LocalDate date1 = currentTime.toLocalDate();
System.out.println("date1: " + date1);

Month month = currentTime.getMonth();
int day = currentTime.getDayOfMonth();
int seconds = currentTime.getSecond();
System.out.println("Month: " + month +“, day: " + day +“, seconds: " + seconds);
```

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Month;
```

4. Date/time improvements (local)

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

```
// Get the current date and time
LocalDateTime currentTime = LocalDateTime.now();    System.out.println("Current
DateTime: " + currentTime);

LocalDate date1 = currentTime.toLocalDate();
System.out.println("date1: " + date1);

Month month = currentTime.getMonth();
int day = currentTime.getDayOfMonth();
int seconds = currentTime.getSecond();
System.out.println("Month: " + month +“, day: " + day +“, seconds: " + seconds);
```

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Month;
```

Console Output

```
Current DateTime: 2017-10-16T19:53:55.053
date1: 2017-10-16
Month: OCTOBER, day: 16, seconds: 55
```

4. Date/time improvements (local)

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

```
// Get the current date and time
ZonedDateTime date1 =
    ZonedDateTime.parse("2017-10-03T10:15:30+05:30[Asia/Karachi]");
System.out.println("date1: " + date1);

ZoneId id = ZoneId.of("Europe/Paris");
System.out.println("ZoneId: " + id);

ZoneId currentZone = ZoneId.systemDefault();
System.out.println("CurrentZone: " + currentZone);
```

```
import java.time.ZonedDateTime;
import java.time.ZoneId;
```

4. Date/time improvements (zoned)

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

```
// Get the current date and time
ZonedDateTime date1 =
    ZonedDateTime.parse("2017-10-03T10:15:30+05:30[Asia/Karachi]");
System.out.println("date1: " + date1);

ZoneId id = ZoneId.of("Europe/Paris");
System.out.println("ZoneId: " + id);

ZoneId currentZone = ZoneId.systemDefault();
System.out.println("CurrentZone: " + currentZone);
```

```
import java.time.ZonedDateTime;
import java.time.ZoneId;
```

Console Output

```
date1: 2017-10-03T10:15:30+05:00[Asia/Karachi]
ZoneId: Europe/Paris
CurrentZone: Etc/UTC
```

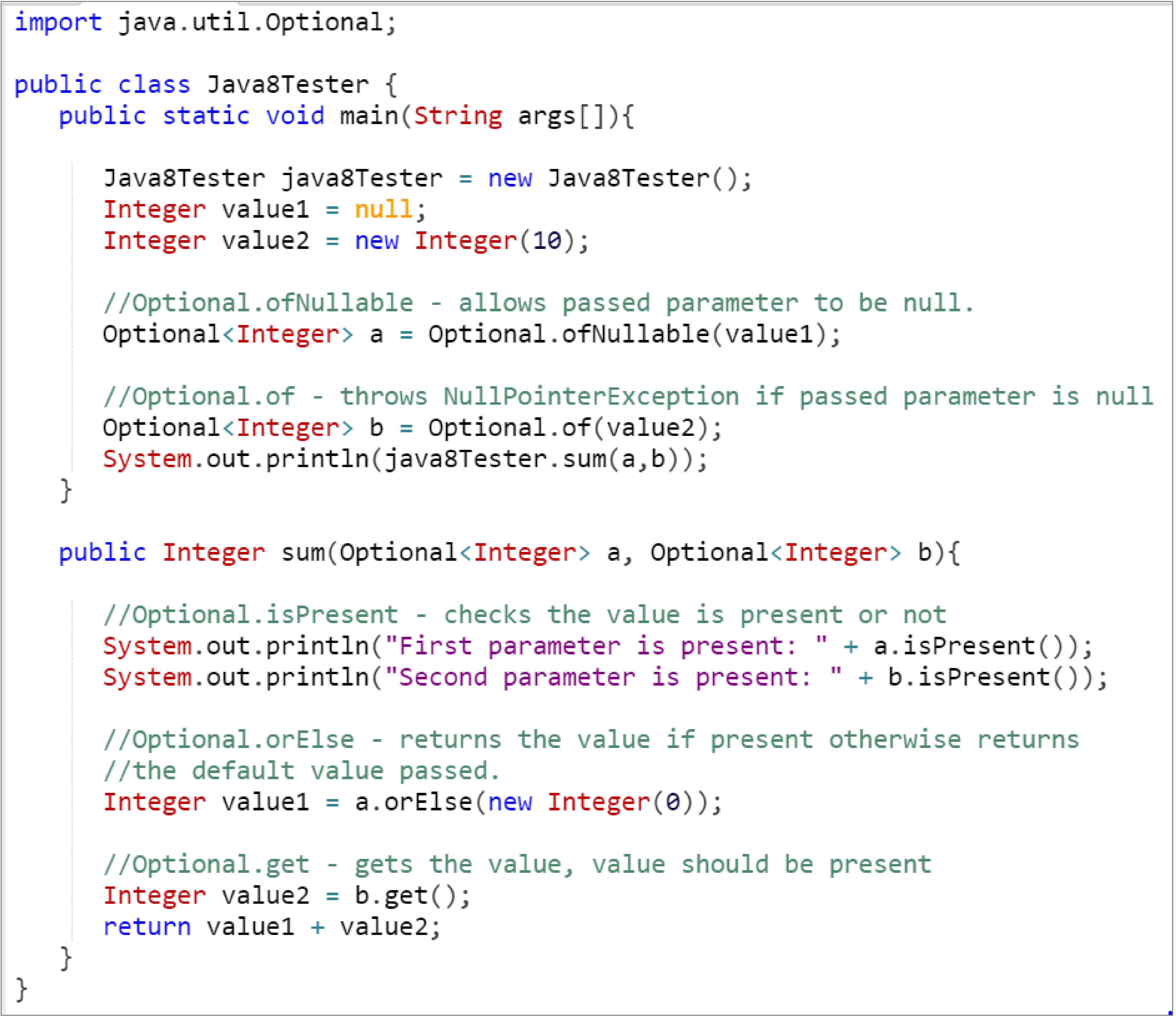
4. Date/time improvements (zoned)

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

- Is similar to **Optional** in Guava.
- is a container object which is used to contain not-null objects:
  - object is used to represent null with absent value.
  - has various utility methods to handle values as ‘available’ or ‘not available’ instead of checking null values.

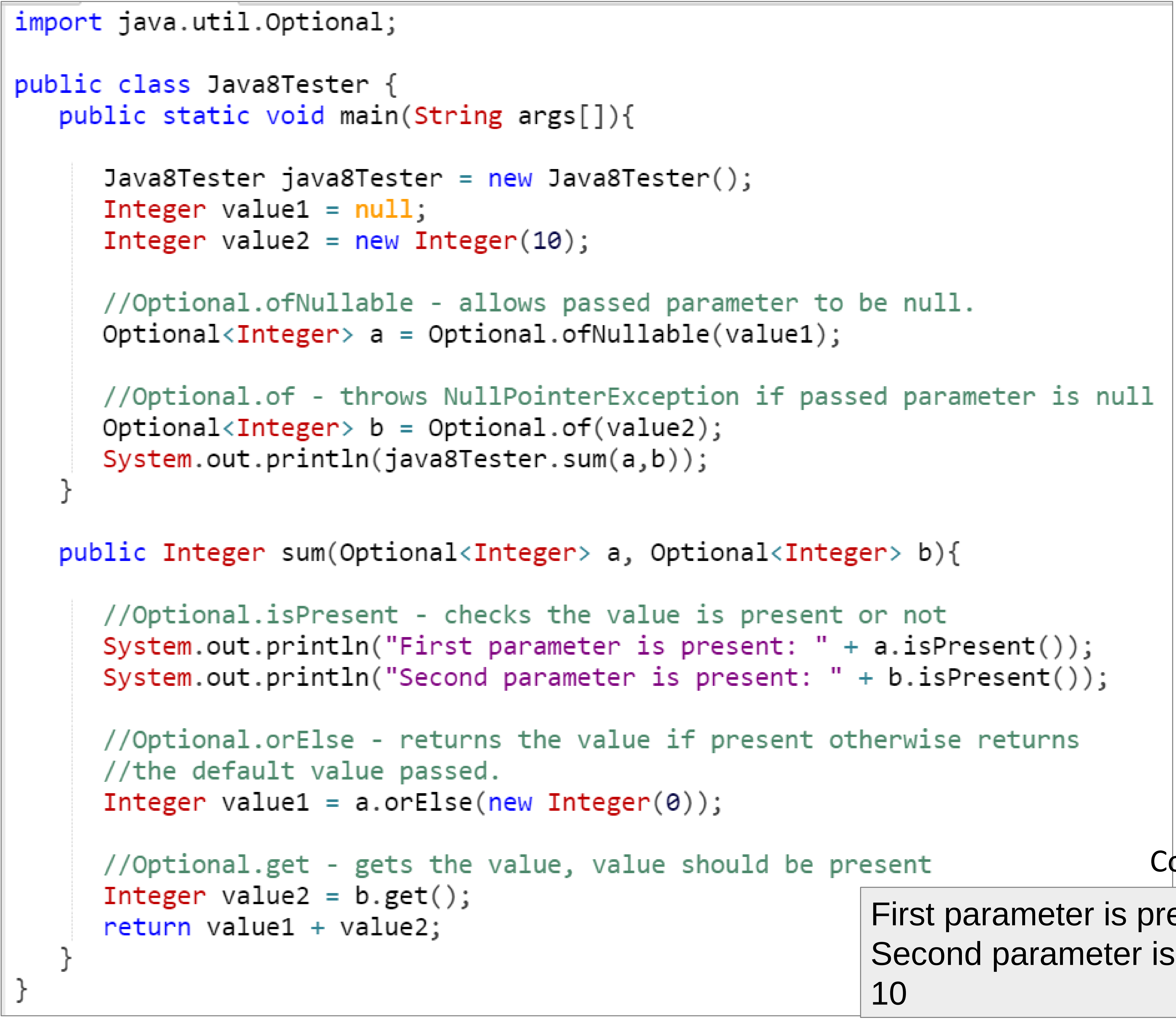
5. Optionals: java.util.Optional<T>

https://www.tutorialspoint.com/java8/java8_optional_class.htm



```java
import java.util.Optional;

public class Java8Tester {
   public static void main(String args[]){

      Java8Tester java8Tester = new Java8Tester();
      Integer value1 = null;
      Integer value2 = new Integer(10);

      //Optional.ofNullable - allows passed parameter to be null.
      Optional<Integer> a = Optional.ofNullable(value1);

      //Optional.of - throws NullPointerException if passed parameter is null
      Optional<Integer> b = Optional.of(value2);
      System.out.println(java8Tester.sum(a,b));
   }

   public Integer sum(Optional<Integer> a, Optional<Integer> b){

      //Optional.isPresent - checks the value is present or not
      System.out.println("First parameter is present: " + a.isPresent());
      System.out.println("Second parameter is present: " + b.isPresent());

      //Optional.orElse - returns the value if present otherwise returns
      //the default value passed.
      Integer value1 = a.orElse(new Integer(0));

      //Optional.get - gets the value, value should be present
      Integer value2 = b.get();
      return value1 + value2;
   }
}
```

https://www.tutorialspoint.com/java8/java8_optional_class.htm



```java
import java.util.Optional;

public class Java8Tester {
   public static void main(String args[]){

      Java8Tester java8Tester = new Java8Tester();
      Integer value1 = null;
      Integer value2 = new Integer(10);

      //Optional.ofNullable - allows passed parameter to be null.
      Optional<Integer> a = Optional.ofNullable(value1);

      //Optional.of - throws NullPointerException if passed parameter is null
      Optional<Integer> b = Optional.of(value2);
      System.out.println(java8Tester.sum(a,b));
   }

   public Integer sum(Optional<Integer> a, Optional<Integer> b){

      //Optional.isPresent - checks the value is present or not
      System.out.println("First parameter is present: " + a.isPresent());
      System.out.println("Second parameter is present: " + b.isPresent());

      //Optional.orElse - returns the value if present otherwise returns
      //the default value passed.
      Integer value1 = a.orElse(new Integer(0));

      //Optional.get - gets the value, value should be present
      Integer value2 = b.get();
      return value1 + value2;
   }
}
```

Console Output

```
First parameter is present: false
Second parameter is present: true
10
```

https://www.tutorialspoint.com/java8/java8_optional_class.htm

Release Date:
Sept 2017

https://docs.oracle.com/javase/9/

https://www.oracle.com/java/java9-screencasts.html

1. Interfaces – private methods
2. Collection factory methods
3. Try with resources improvements
4. Stream API improvements
5. REPL (Shell)
6. Module system

An outline of some changes

1. Defining Interfaces (JDK 9)
With private methods

Java 9 allows **private methods** as a way to avoid writing duplicate code (i.e. promote re-usability) and also to hide interface implementation.

The methods can be private and private static and are written in the same way you would write a private method in a class.

```
public interface IAddressBook{
  static final int CAPACITY= 1000;

  void clear();
  IContact getContact(String lastName);
  void addContact(IContact contact);
  int numberOfContacts();
  void removeContact(String lastName);
  String listContacts();


  default String typeOfEntity(){
       return “Address book”;
  }
  static int getCapacity(){
        return CAPACITY;
  }


  private static void displayDetails(){
     //method implementation in here
  }
}
```

IAddressBook.java

2. Collection Factory Methods

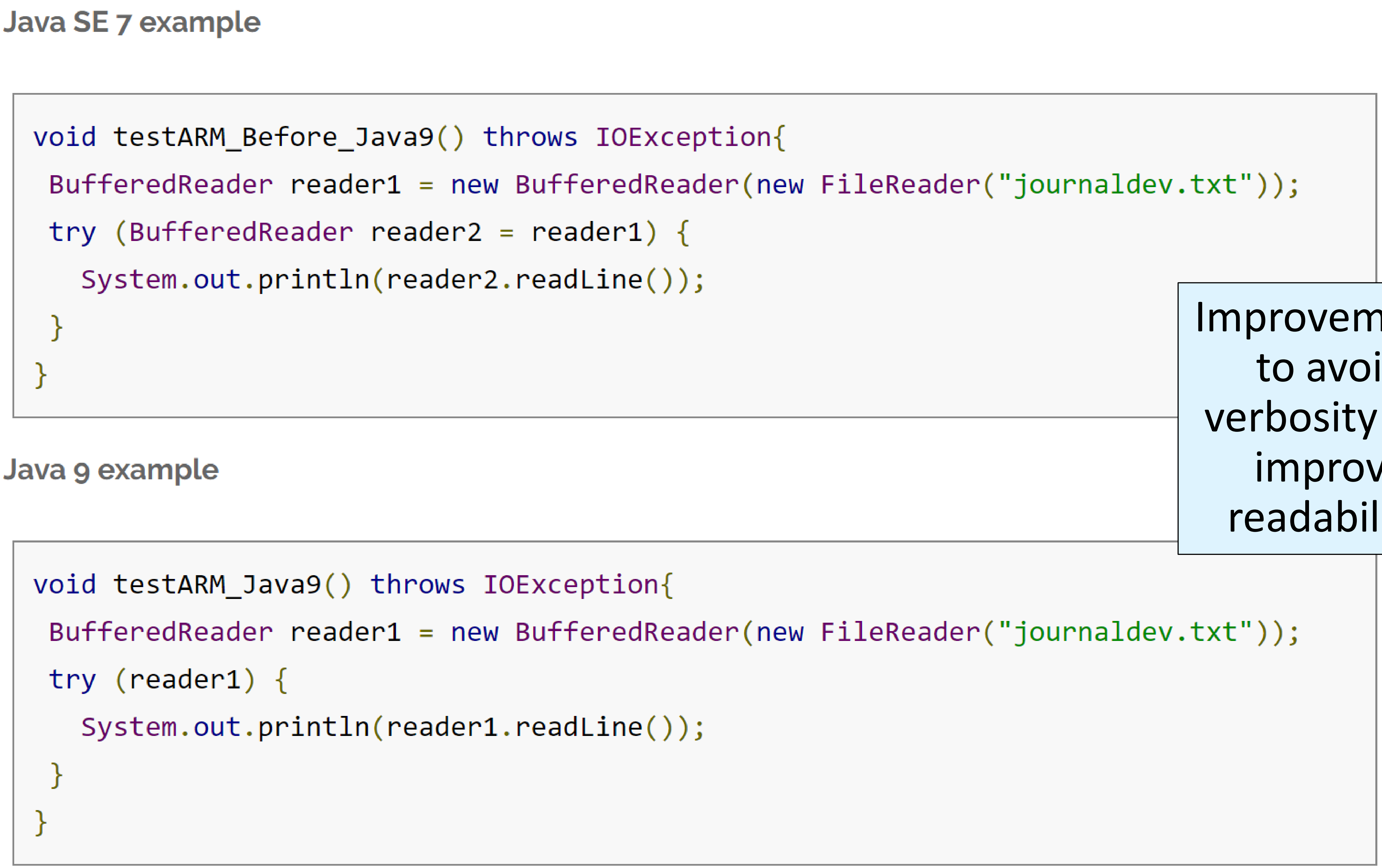
- Often you want to create a collection (e.g., a List or Set) in your code and directly populate it with some elements…
  - That leads to repetitive code where you instantiate the collection, followed by several `*add*` calls.
- With Java 9, several **collection factory methods** have been added:

```
Set<Integer> ints         = Set.of(1, 2, 3);
List<String> strings      = List.of("first", "second");
Map<String, String> map = Map.of("foo", "a", "bar", "b", "c");
```

NOTE: Immutable collections are created (i.e. cannot add to it) and the collection implementation is selected by Java (e.g. ArrayList, LinkedList).

https://www.pluralsight.com/blog/software-development/java-9-new-features

3. try-with-resources improvement



Java SE 7 example

```
void testARM_Before_Java9() throws IOException{
 BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));
 try (BufferedReader reader2 = reader1) {
   System.out.println(reader2.readLine());
 }
}
```

Java 9 example

```
void testARM_Java9() throws IOException{
 BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));
 try (reader1) {
   System.out.println(reader1.readLine());
 }
}
```

Improvements to avoid verbosity and improve readability.

https://www.pluralsight.com/blog/software-development/java-9-new-features

4. Stream API improvement

- Addition of extra methods…as a sample, we will look at the **takeWhile** method:
  - takes a predicate as an argument and returns a Stream of subset of the given Stream values until that Predicate returns false for first time. If first value does NOT satisfy that Predicate, it just returns an empty Stream.

```
Stream.of(1,2,3,4,5,6,7,8,9,10)
        .takeWhile(i -> i < 5 )
        .forEach(System.out::println);
```

Console Output

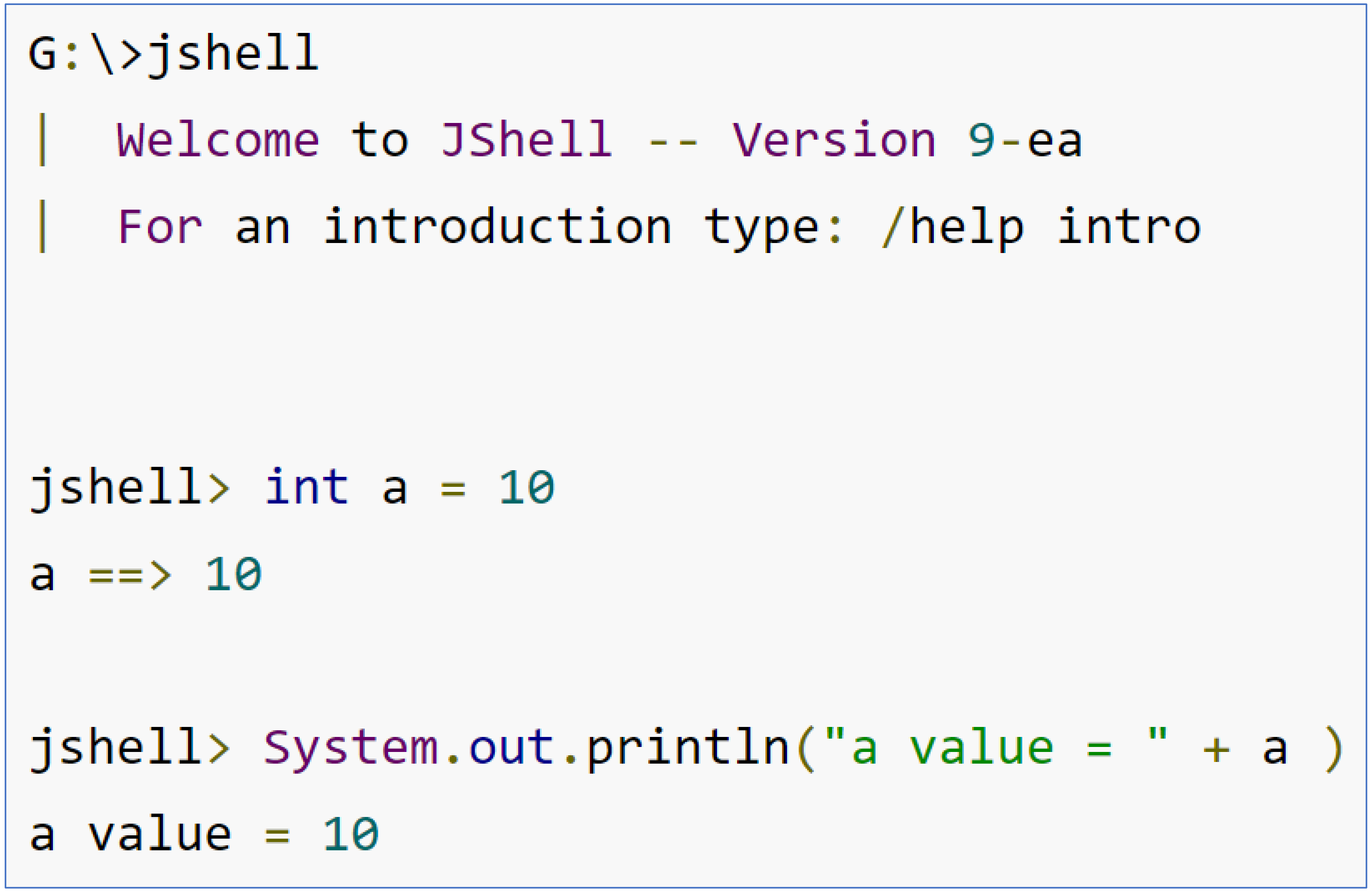
```
1
2
3
4
```

https://www.pluralsight.com/blog/software-development/java-9-new-features

- The jshell is used to easily execute and test Java constructs like class, interface, enum, object, statements etc.

On your command line, start the shell by typing jshell.

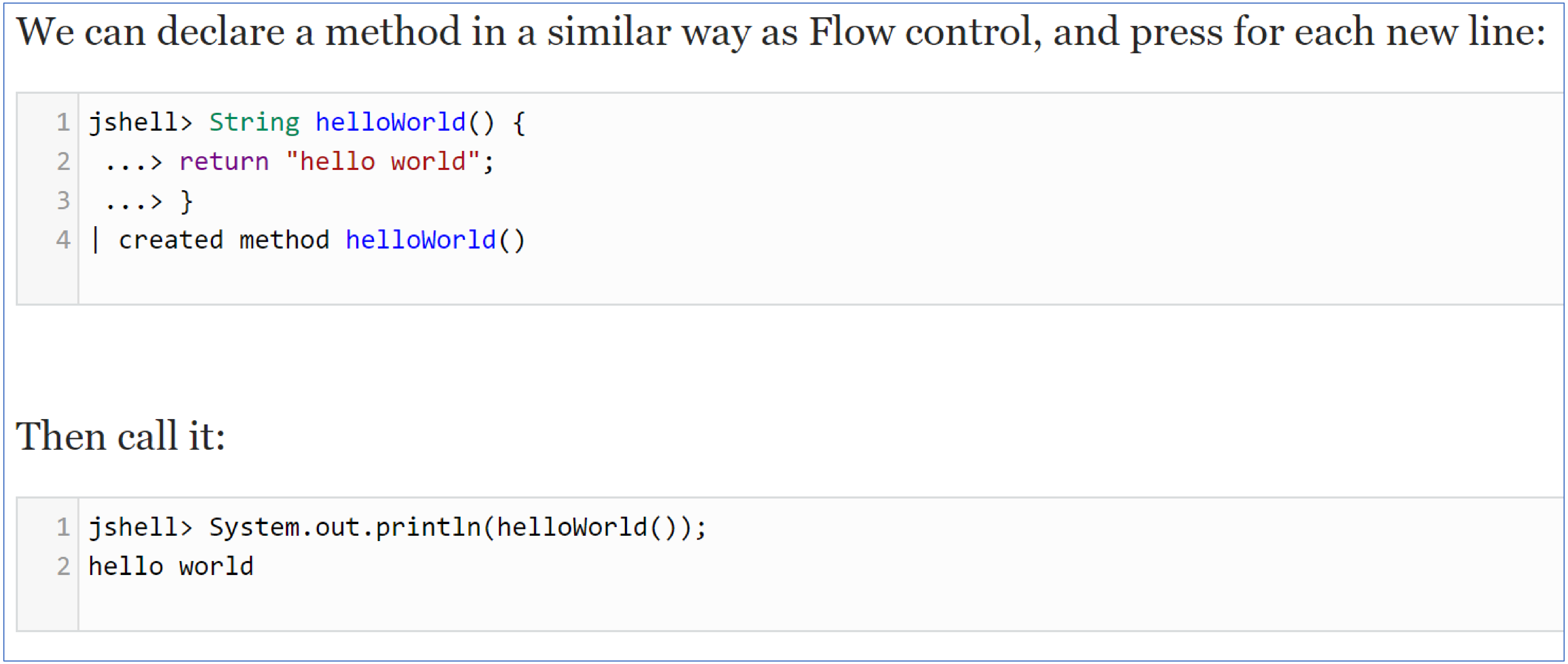
Then enter any Java 9 statements you wish.



```
G:\>jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro


jshell> int a = 10
a ==> 10

jshell> System.out.println("a value = " + a )
a value = 10
```

5. REPL (Read Evaluate Print Loop i.e. Shell)

https://www.pluralsight.com/blog/software-development/java-9-new-features



We can declare a method in a similar way as Flow control, and press for each new line:

```
jshell> String helloWorld() {
 ...> return "hello world";
 ...> }
| created method helloWorld()
```

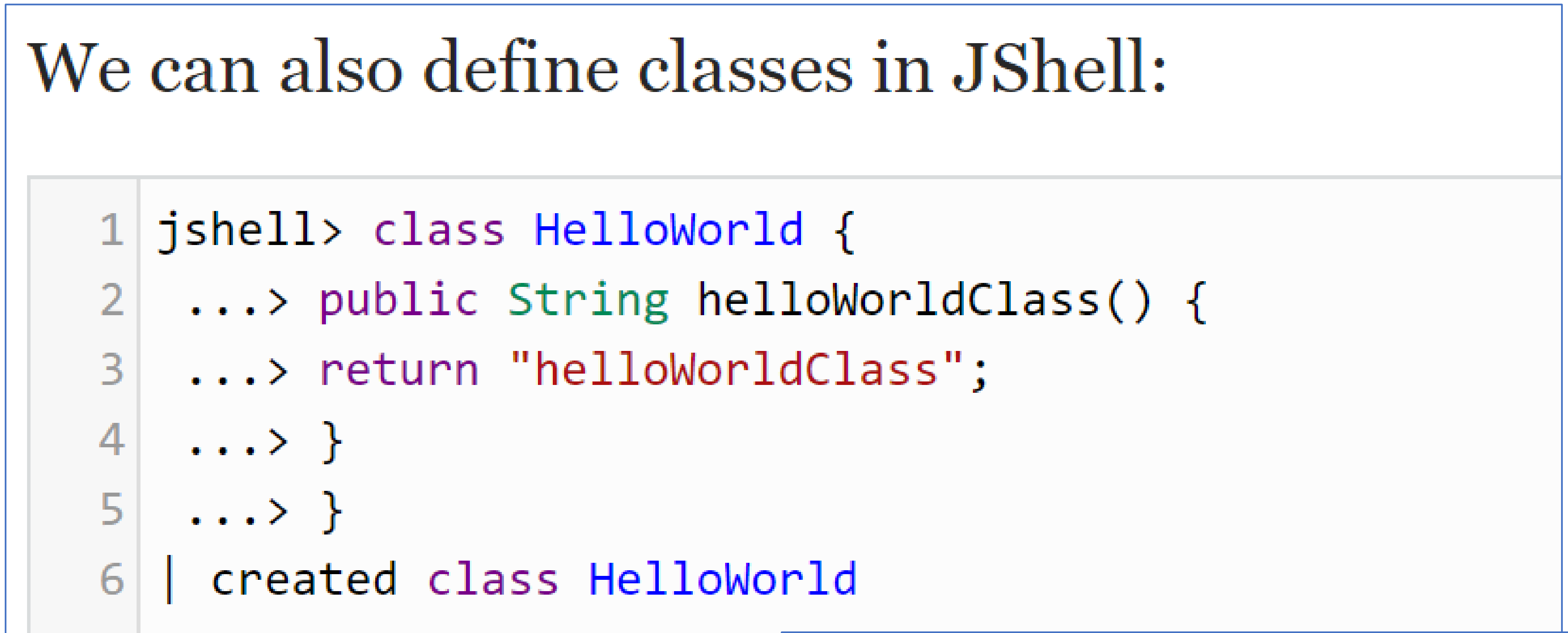
Then call it:

```
jshell> System.out.println(helloWorld());
hello world
```

5. REPL (Read Evaluate Print Loop i.e. Shell)

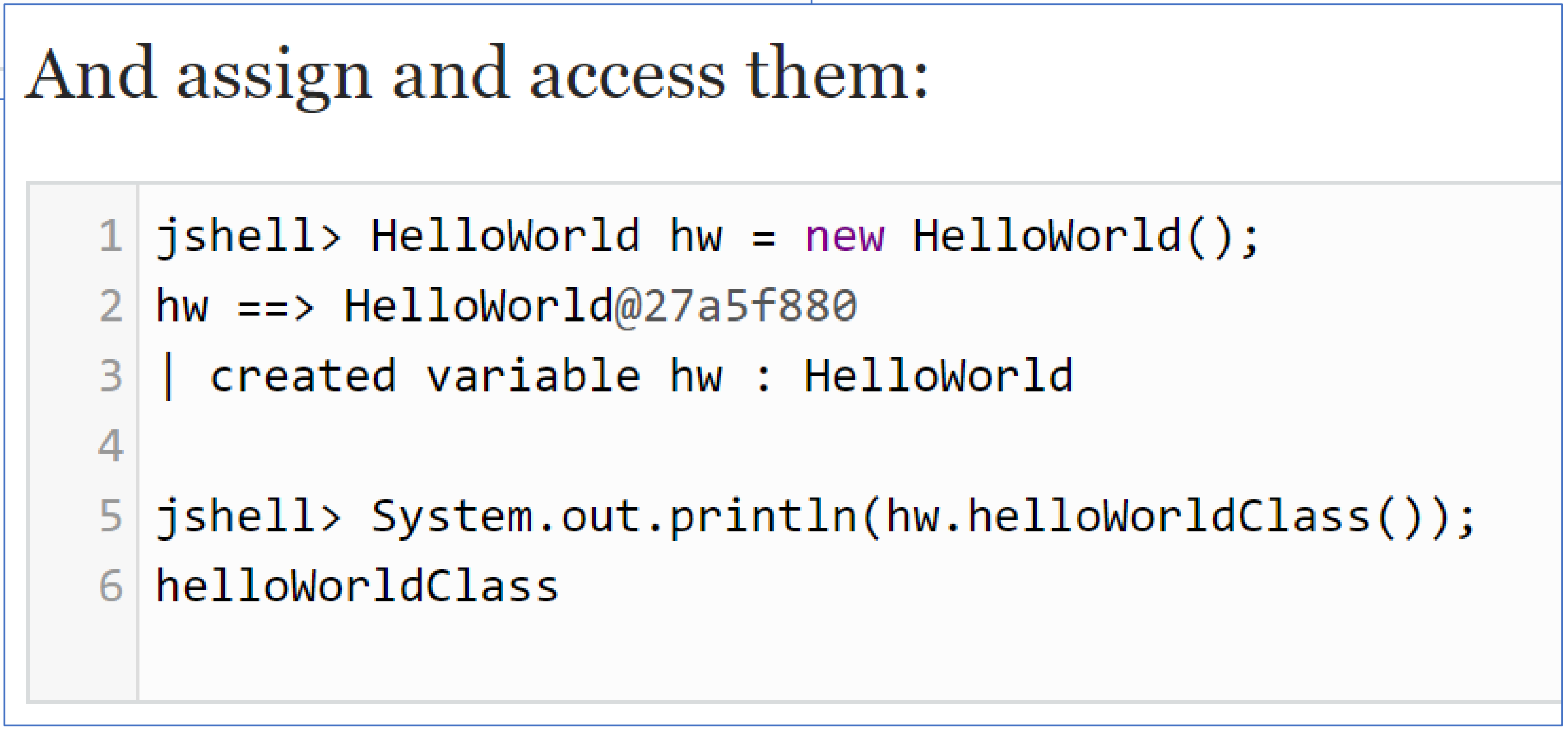
https://www.pluralsight.com/blog/software-development/java-9-new-features

5. REPL (Read Evaluate Print Loop i.e. Shell)



We can also define classes in JShell:

```
jshell> class HelloWorld {
 ...> public String helloWorldClass() {
 ...> return "helloWorldClass";
 ...> }
 ...> }
| created class HelloWorld
```



And assign and access them:

```
jshell> HelloWorld hw = new HelloWorld();
hw ==> HelloWorld@27a5f880
| created variable hw : HelloWorld

jshell> System.out.println(hw.helloWorldClass());
helloWorldClass
```

https://www.pluralsight.com/blog/software-development/java-9-new-features

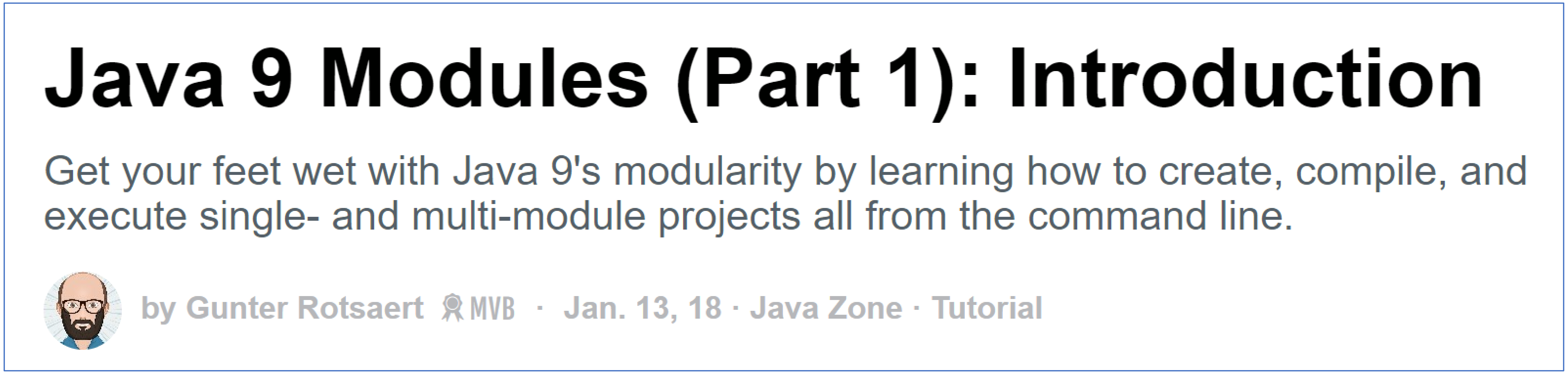
6. Module System

- One of the biggest changes in Java 9 (part of the **Jigsaw** Project).
- With JDK9, you can separate your code into individual modules.
- *“A module is a named, self-describing program component that consists of one or more packages (and data)”* https://jaxenter.com/new-features-in-java-9-137344.html

  - Each module needs a **module-info.java** file. It is placed in the root directory of the module. Within this file, you can declare:
    - which modules your code is dependent upon i.e. jdk modules, external jars, etc.
    - which packages are allowed to see/use your module.

https://www.pluralsight.com/blog/software-development/java-9-new-features

https://dzone.com/articles/java-9-modules-introduction-part-1

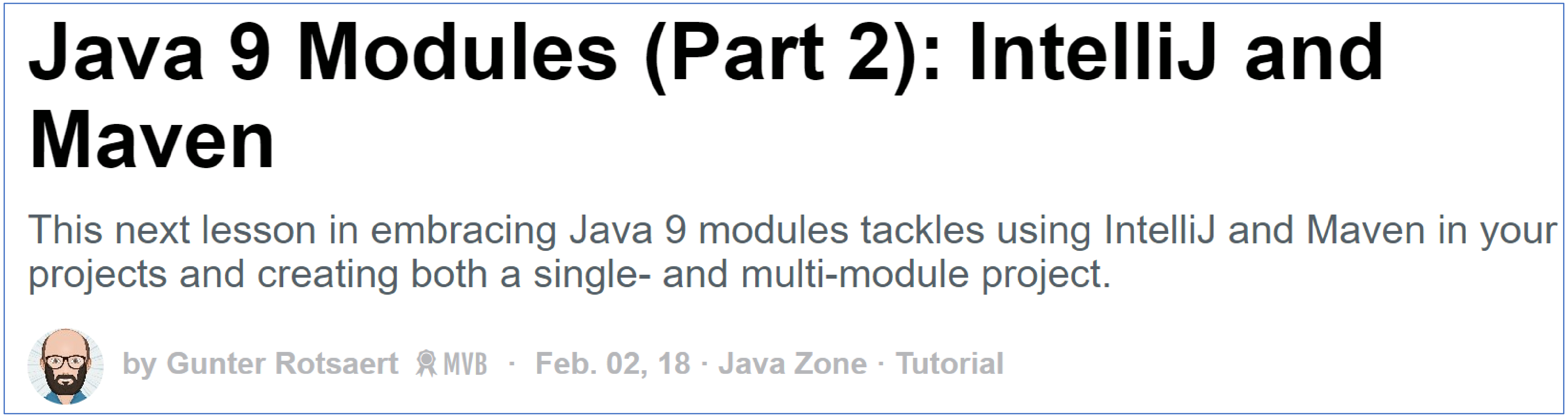


Java 9 Modules (Part 1): Introduction

Get your feet wet with Java 9's modularity by learning how to create, compile, and execute single- and multi-module projects all from the command line.

by Gunter Rotsaert MVB · Jan. 13, 18 · Java Zone · Tutorial

https://dzone.com/articles/java-9-modules-part-2-intellij-and-maven



Java 9 Modules (Part 2): IntelliJ and Maven

This next lesson in embracing Java 9 modules tackles using IntelliJ and Maven in your projects and creating both a single- and multi-module project.

by Gunter Rotsaert MVB · Feb. 02, 18 · Java Zone · Tutorial

6. Module System

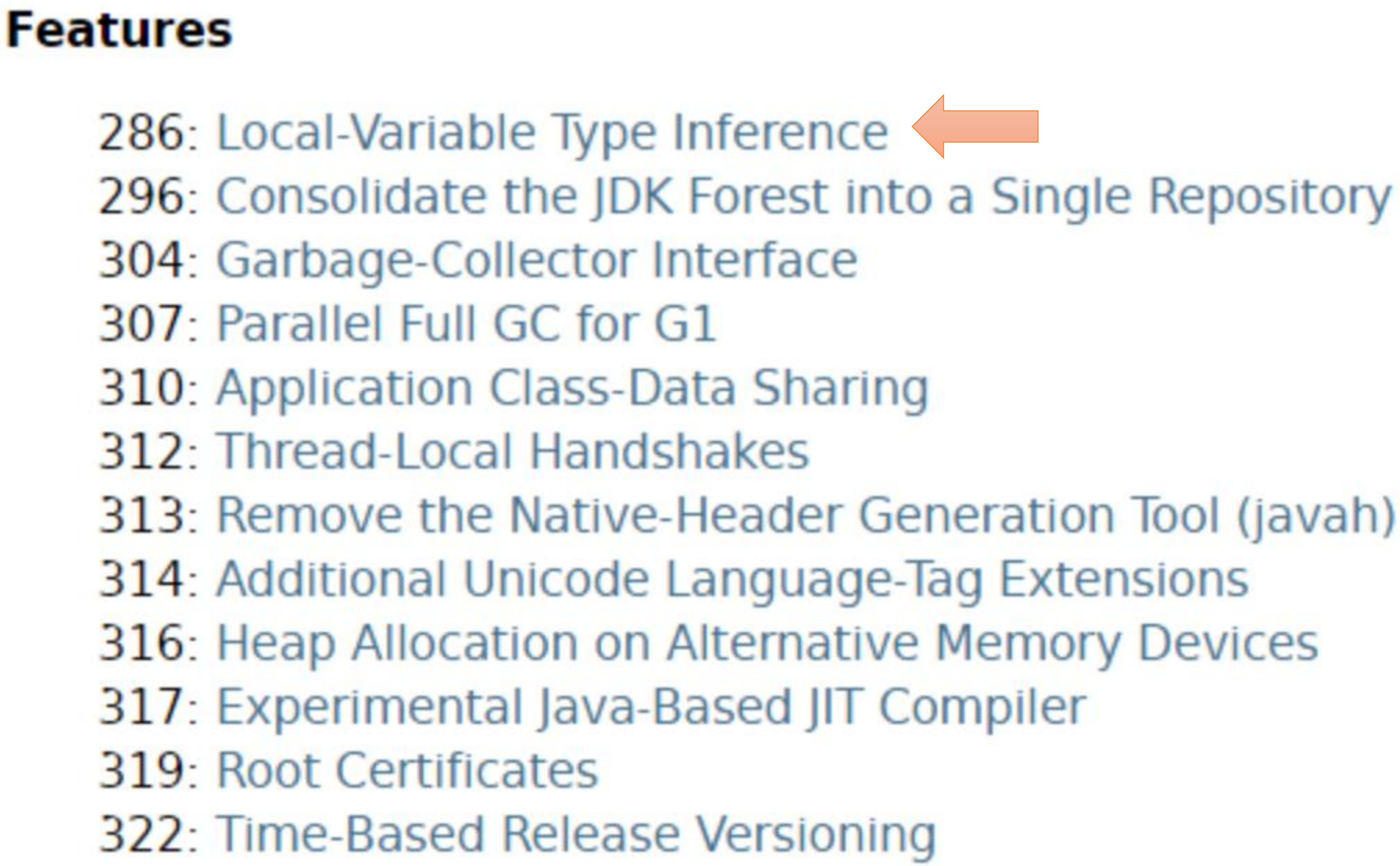
https://www.pluralsight.com/blog/software-development/java-9-new-features

Java10

Release Date:
March 2018

https://docs.oracle.com/javase/10/

An outline of some changes



Features

- 286: Local-Variable Type Inference
- 296: Consolidate the JDK Forest into a Single Repository
- 304: Garbage-Collector Interface
- 307: Parallel Full GC for G1
- 310: Application Class-Data Sharing
- 312: Thread-Local Handshakes
- 313: Remove the Native-Header Generation Tool (javah)
- 314: Additional Unicode Language-Tag Extensions
- 316: Heap Allocation on Alternative Memory Devices
- 317: Experimental Java-Based JIT Compiler
- 319: Root Certificates
- 322: Time-Based Release Versioning

- A new identifier named var is now available for **local** variables with **non-null initializers**.

- Using this identifier, the type of the variable is inferred from the context.

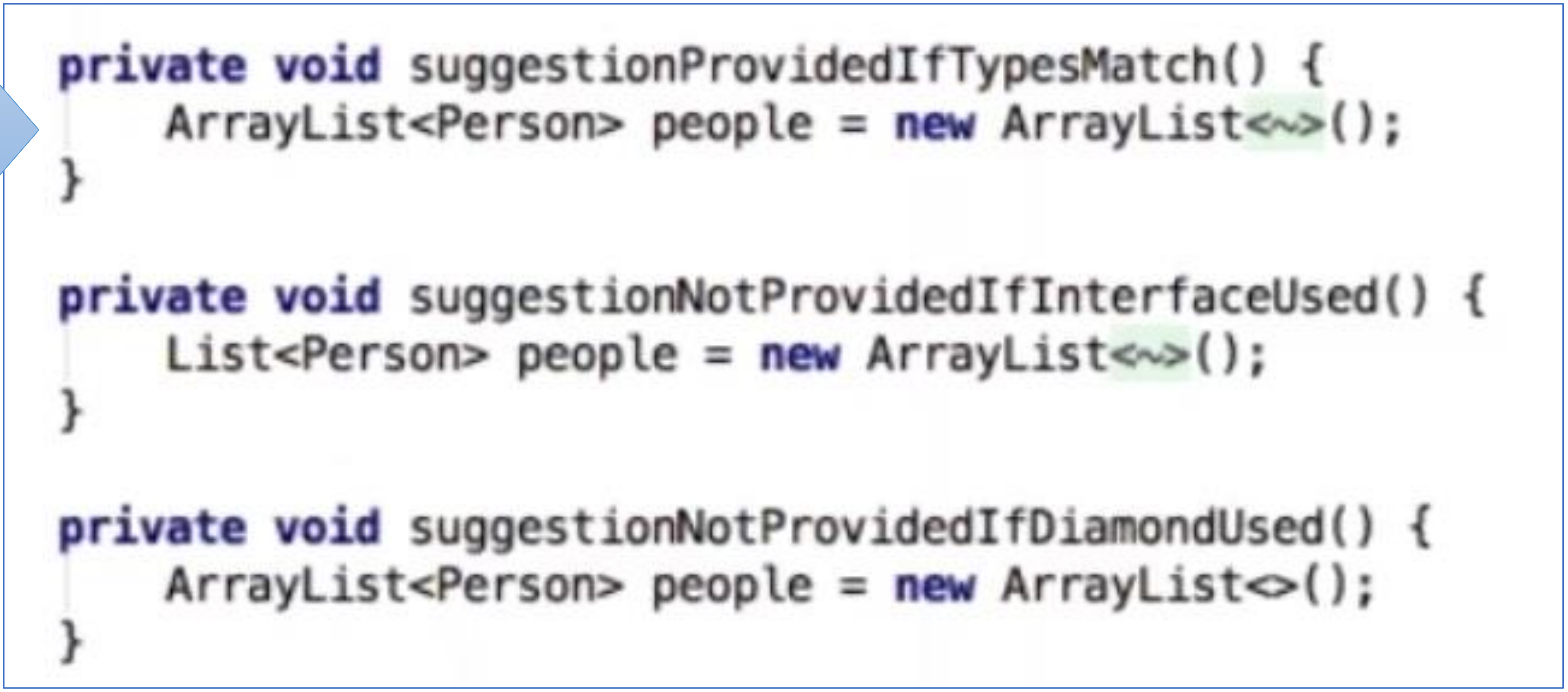
- val was not introduced, instead you use final var (as it is more consistent with the rest of Java).

- Why was this level of type inference introduced?

  → Reduce boilerplate, ease readability.

Local Variable Type Inference

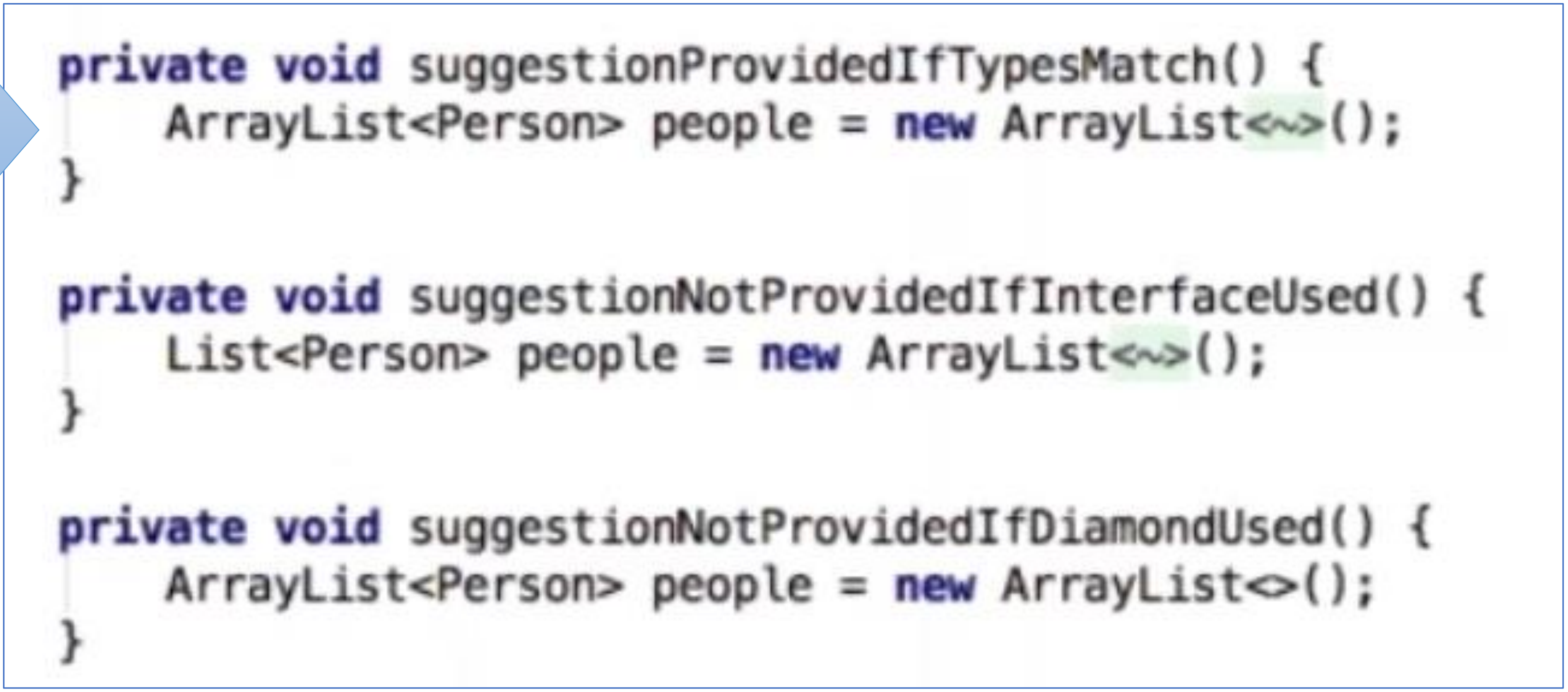
https://www.youtube.com/watch?v=H-LzZofU3zk&feature=youtu.be



```java
private void suggestionProvidedIfTypesMatch() {
    ArrayList<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfInterfaceUsed() {
    List<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfDiamondUsed() {
    ArrayList<Person> people = new ArrayList<>();
}
```

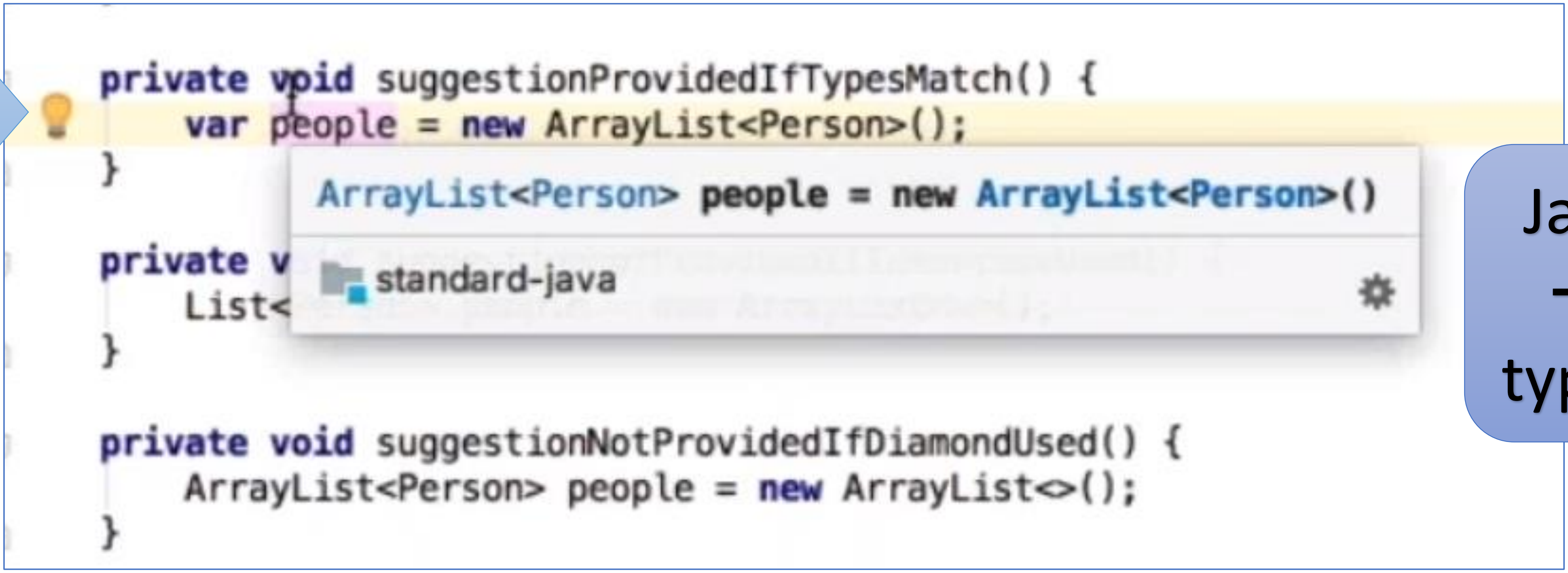
Pre Java 10

https://www.youtube.com/watch?v=H-LzZofU3zk&feature=youtu.be



```
private void suggestionProvidedIfTypesMatch() {
    ArrayList<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfInterfaceUsed() {
    List<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfDiamondUsed() {
    ArrayList<Person> people = new ArrayList<>();
}
```

Pre Java 10

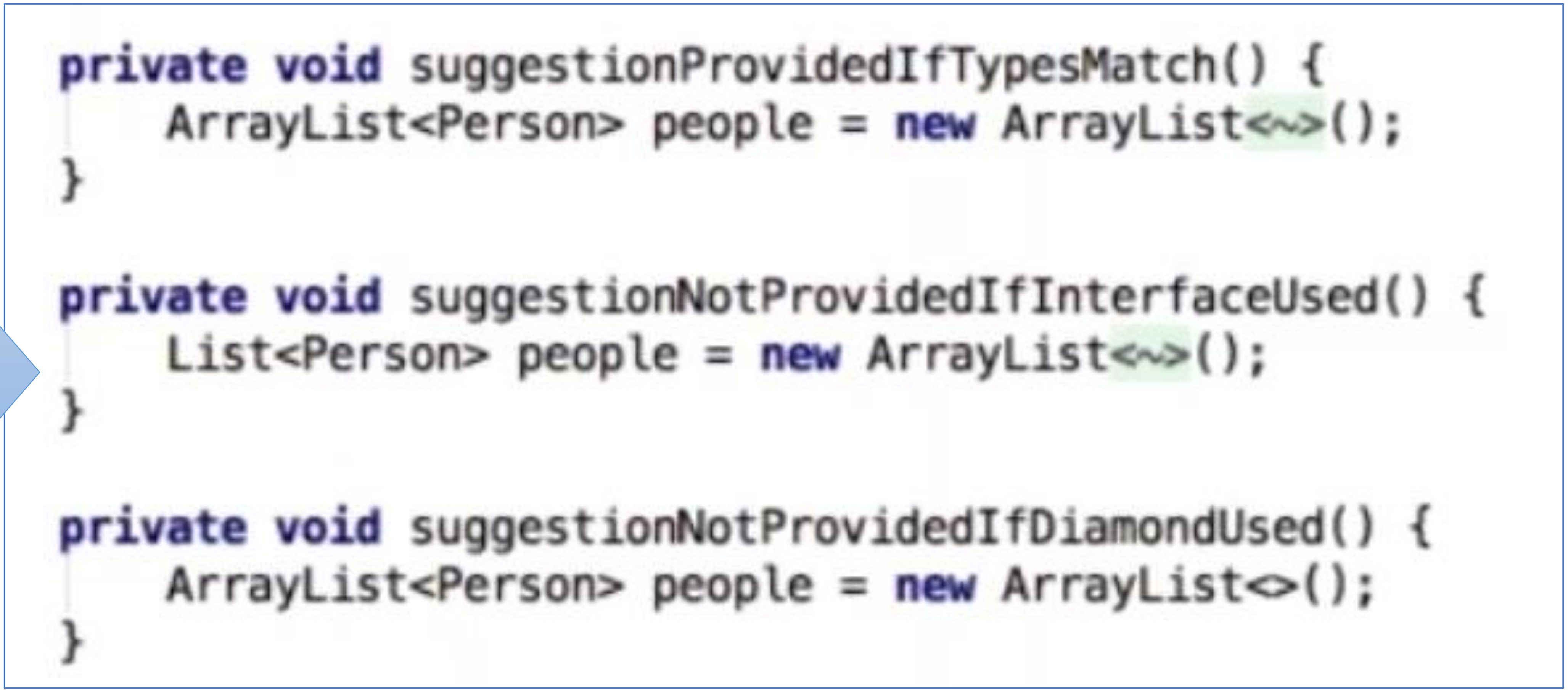


```
private void suggestionProvidedIfTypesMatch() {
    var people = new ArrayList<Person>();
}

private v
    List<
}

private void suggestionNotProvidedIfDiamondUsed() {
    ArrayList<Person> people = new ArrayList<>();
}
```

```
ArrayList<Person> people = new ArrayList<Person>()
standard-java
```

Java 10 → no type loss

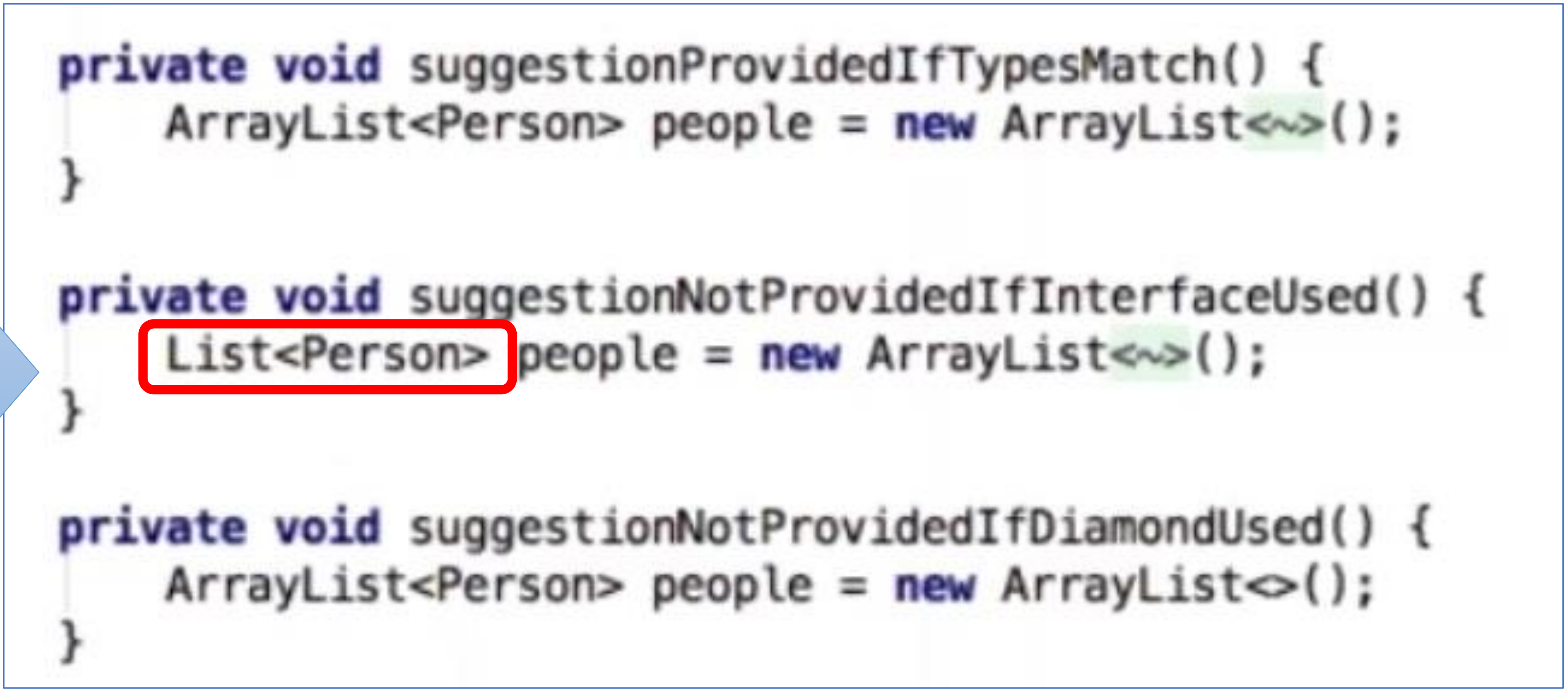
https://www.youtube.com/watch?v=H-LzZofU3zk&feature=youtu.be



```
private void suggestionProvidedIfTypesMatch() {
    ArrayList<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfInterfaceUsed() {
    List<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfDiamondUsed() {
    ArrayList<Person> people = new ArrayList<>();
}
```

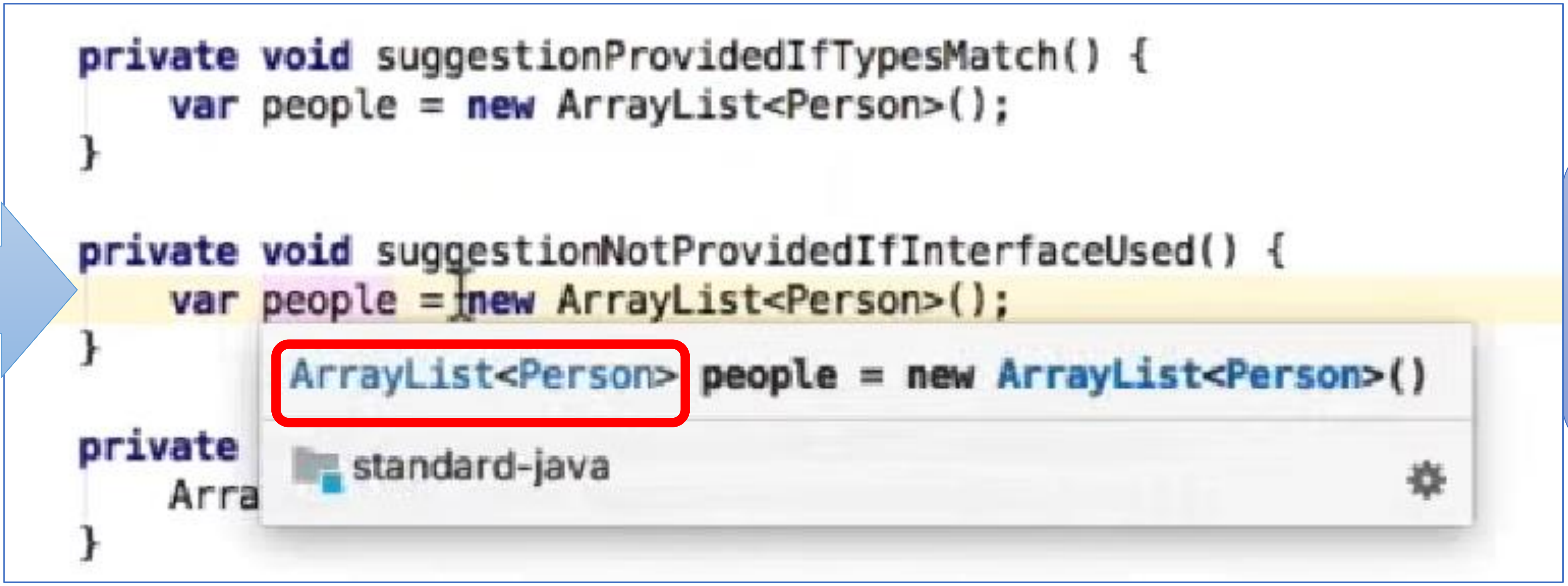
Pre Java 10

https://www.youtube.com/watch?v=H-LzZofU3zk&feature=youtu.be



```java
private void suggestionProvidedIfTypesMatch() {
    ArrayList<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfInterfaceUsed() {
    List<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfDiamondUsed() {
    ArrayList<Person> people = new ArrayList<>();
}
```

Pre Java 10

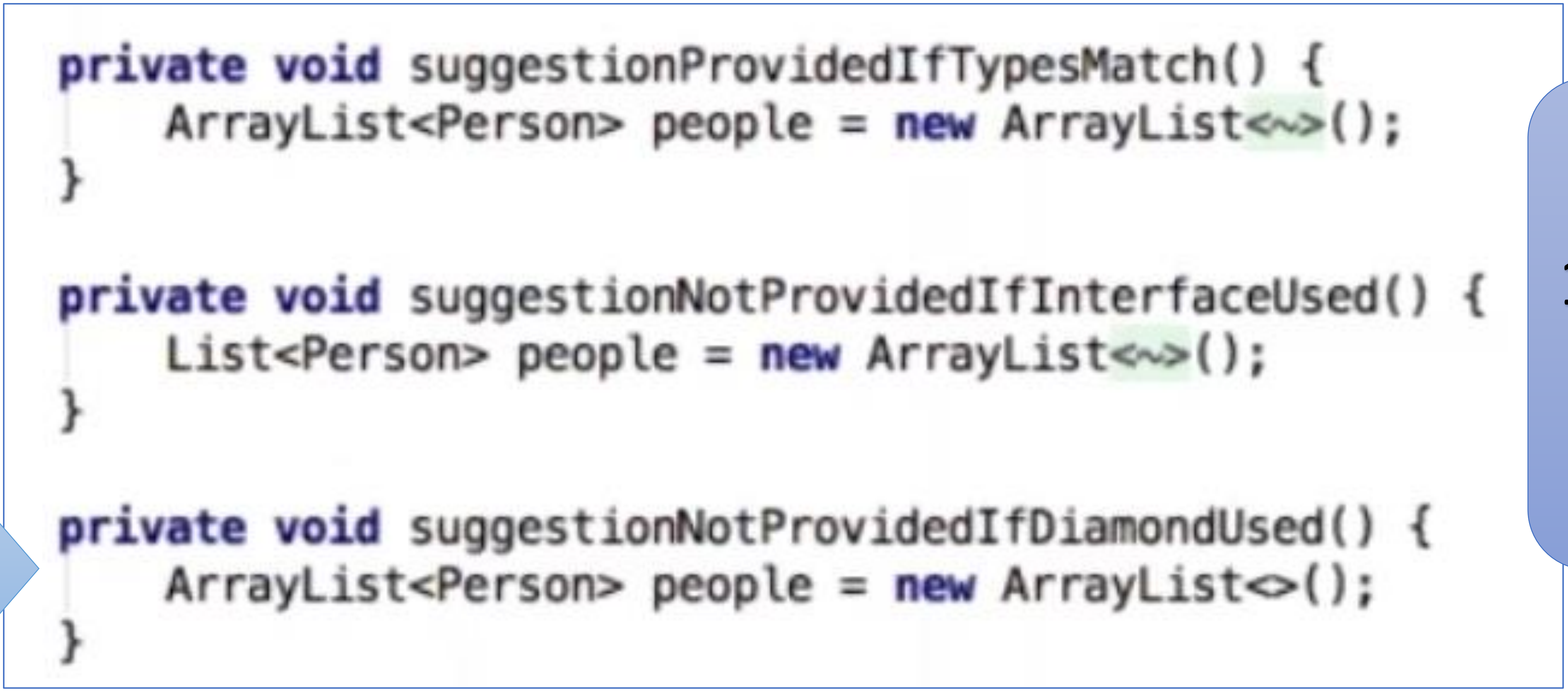


```java
private void suggestionProvidedIfTypesMatch() {
    var people = new ArrayList<Person>();
}

private void suggestionNotProvidedIfInterfaceUsed() {
    var people = new ArrayList<Person>();
}

private
    Arra
}
```

```
ArrayList<Person> people = new ArrayList<Person>()
standard-java
```

Java 10 → **type is changed**

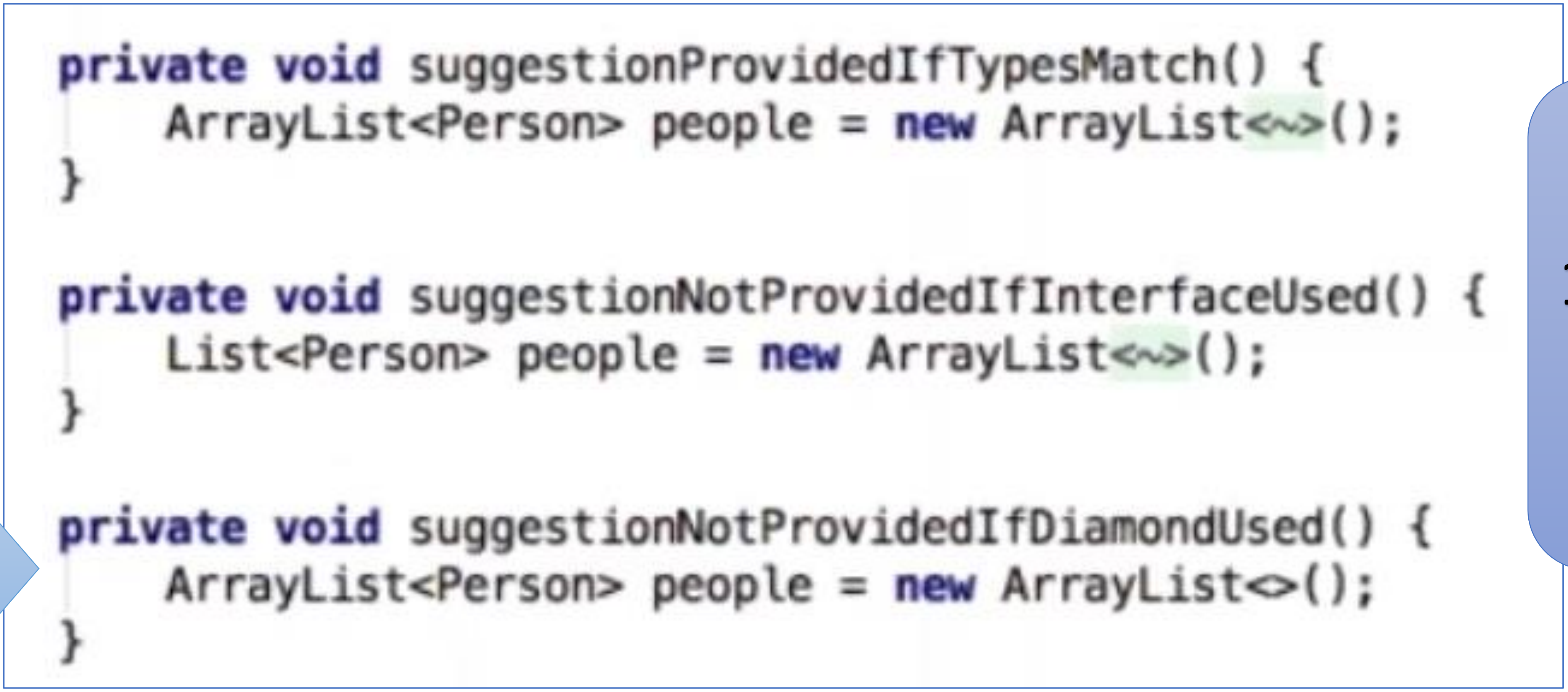
https://www.youtube.com/watch?v=H-LzZofU3zk&feature=youtu.be



```
private void suggestionProvidedIfTypesMatch() {
    ArrayList<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfInterfaceUsed() {
    List<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfDiamondUsed() {
    ArrayList<Person> people = new ArrayList<>();
}
```

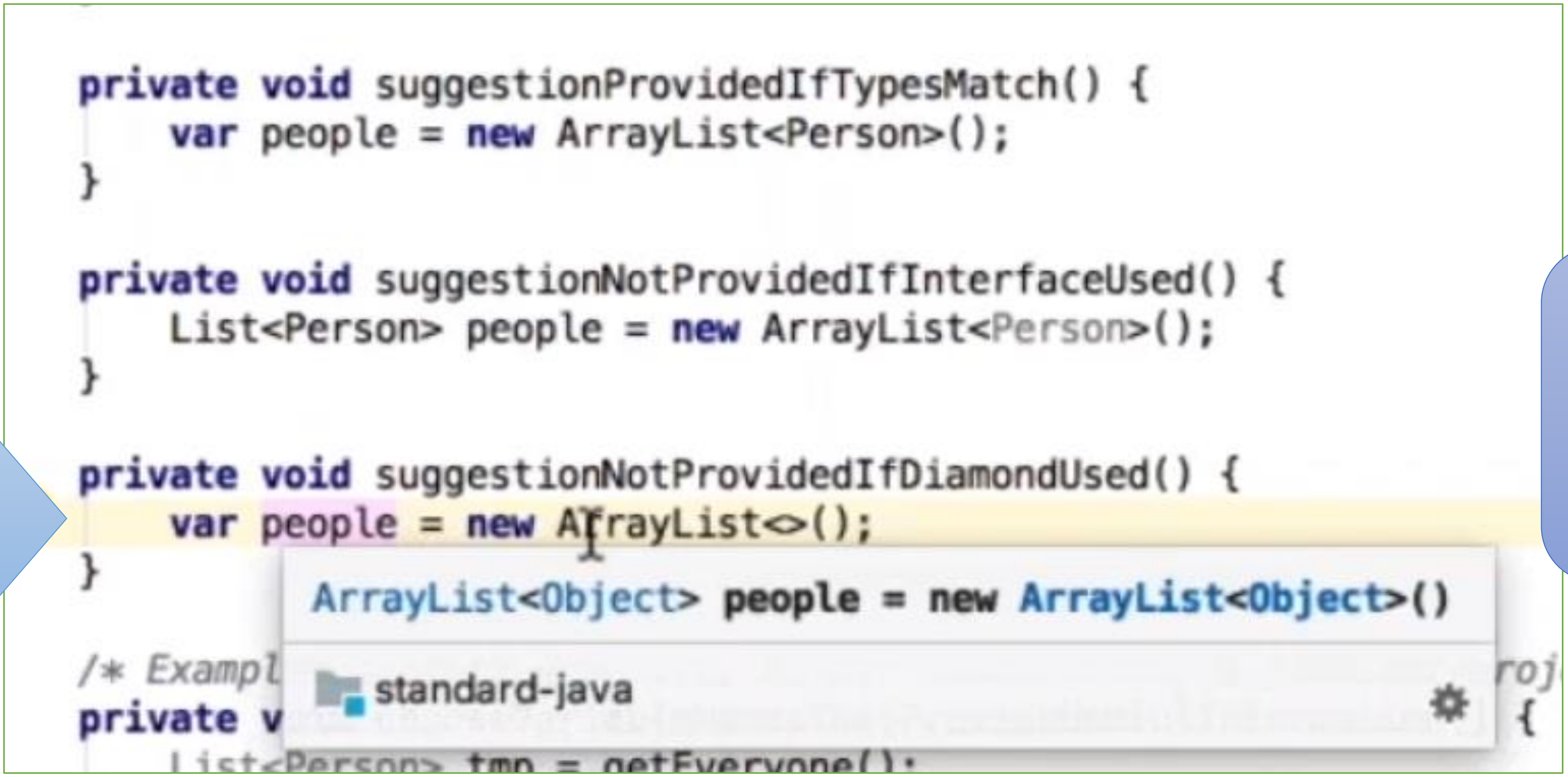
Pre Java 10 – using type inference

https://www.youtube.com/watch?v=H-LzZofU3zk&feature=youtu.be



```
private void suggestionProvidedIfTypesMatch() {
    ArrayList<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfInterfaceUsed() {
    List<Person> people = new ArrayList<~>();
}

private void suggestionNotProvidedIfDiamondUsed() {
    ArrayList<Person> people = new ArrayList<>();
}
```

Pre Java 10 – using type inference

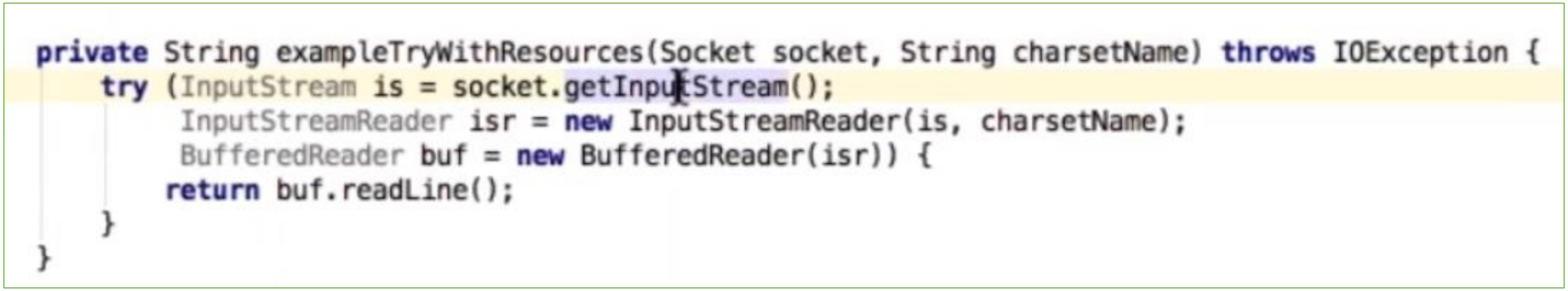


```
private void suggestionProvidedIfTypesMatch() {
    var people = new ArrayList<Person>();
}

private void suggestionNotProvidedIfInterfaceUsed() {
    List<Person> people = new ArrayList<Person>();
}

private void suggestionNotProvidedIfDiamondUsed() {
    var people = new ArrayList<>();
}

/* Exampl                                          roj
private v                                          {
    List<Person> tmp = getEveryone();
```

```
ArrayList<Object> people = new ArrayList<Object>()
standard-java
```

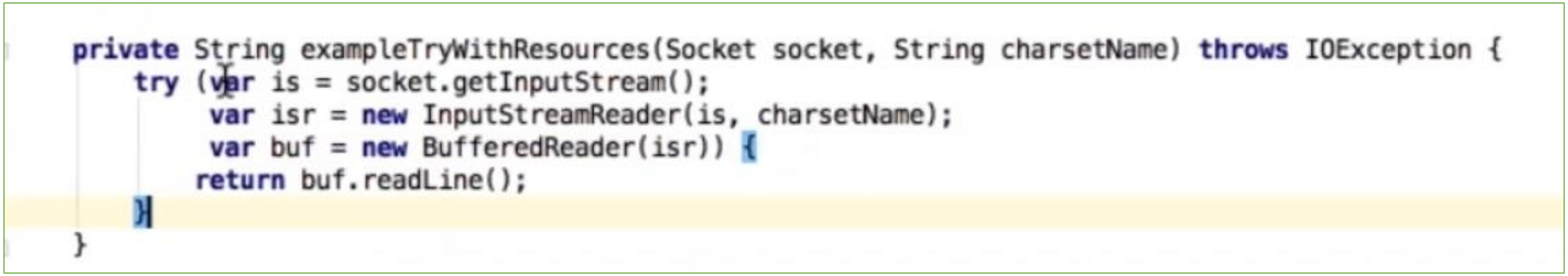
Java 10 → **type is changed**

https://www.youtube.com/watch?v=H-LzZofU3zk&feature=youtu.be



```java
private String exampleTryWithResources(Socket socket, String charsetName) throws IOException {
    try (InputStream is = socket.getInputStream();
         InputStreamReader isr = new InputStreamReader(is, charsetName);
         BufferedReader buf = new BufferedReader(isr)) {
        return buf.readLine();
    }
}
```

becomes



```java
private String exampleTryWithResources(Socket socket, String charsetName) throws IOException {
    try (var is = socket.getInputStream();
         var isr = new InputStreamReader(is, charsetName);
         var buf = new BufferedReader(isr)) {
        return buf.readLine();
    }
}
```

https://www.youtube.com/watch?v=H-LzZofU3zk&feature=youtu.be

Can be used for the following types of variables:

- Local variable declarations with initializers
- Enhanced for-loop indexes
- Index variables declared in traditional for loops
- Try-with-resources variable

Cannot be used for the following types of variables:

- Fields
- Return types
- Parameters

Note: it only works with local variables; in the above, there is (generally) no way to figure out what that type really is.

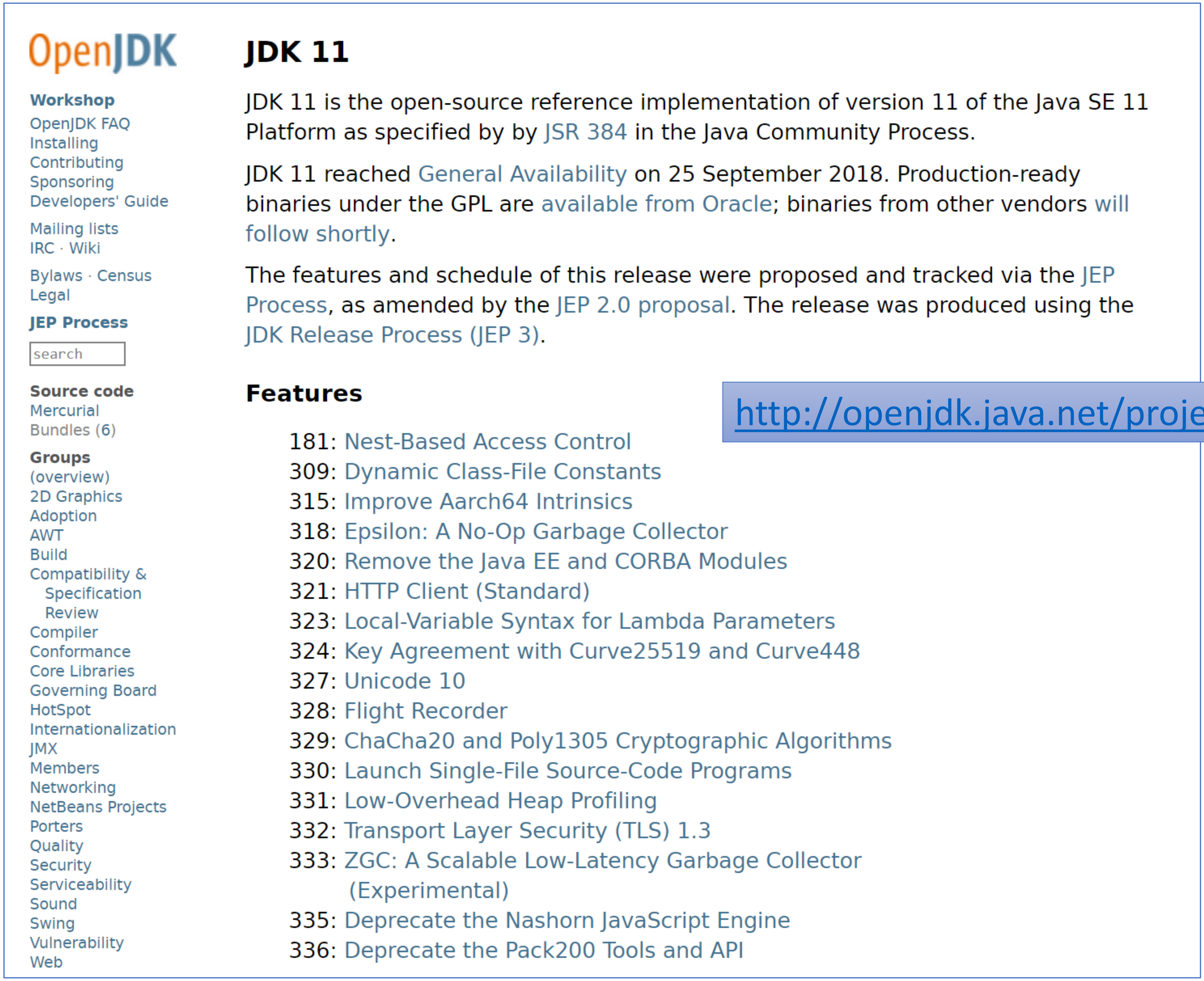
Java is still a statically typed language

- It might “look” like Java 10 is moving to a more dynamic version of the language…
  - ***Not true*** - you still have full type safety, it’s just a removal of some of the syntax.

- IntelliJ (and other IDEs too) will help you figure out where you can make use of this:
  - Use Inspections → set Java 10 local variable type inference to a “weak warning” to highlight the areas in your code you can use this new feature.

Release Date:
Sept 2018

Currently outside the scope of this module, but you can review the changes yourself:



OpenJDK

Workshop
OpenJDK FAQ
Installing
Contributing
Sponsoring
Developers' Guide

Mailing lists
IRC · Wiki

Bylaws · Census
Legal

JEP Process

search

Source code
Mercurial
Bundles (6)

Groups
(overview)
2D Graphics
Adoption
AWT
Build
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
Internationalization
JMX
Members
Networking
NetBeans Projects
Porters
Quality
Security
Serviceability
Sound
Swing
Vulnerability
Web

JDK 11

JDK 11 is the open-source reference implementation of version 11 of the Java SE 11 Platform as specified by by JSR 384 in the Java Community Process.

JDK 11 reached General Availability on 25 September 2018. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal. The release was produced using the JDK Release Process (JEP 3).

Features

181: Nest-Based Access Control
309: Dynamic Class-File Constants
315: Improve Aarch64 Intrinsics
318: Epsilon: A No-Op Garbage Collector
320: Remove the Java EE and CORBA Modules
321: HTTP Client (Standard)
323: Local-Variable Syntax for Lambda Parameters
324: Key Agreement with Curve25519 and Curve448
327: Unicode 10
328: Flight Recorder
329: ChaCha20 and Poly1305 Cryptographic Algorithms
330: Launch Single-File Source-Code Programs
331: Low-Overhead Heap Profiling
332: Transport Layer Security (TLS) 1.3
333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)
335: Deprecate the Nashorn JavaScript Engine
336: Deprecate the Pack200 Tools and API

http://openjdk.java.net/projects/jdk/11/

Any questions?