

From Java 8 to Java 13

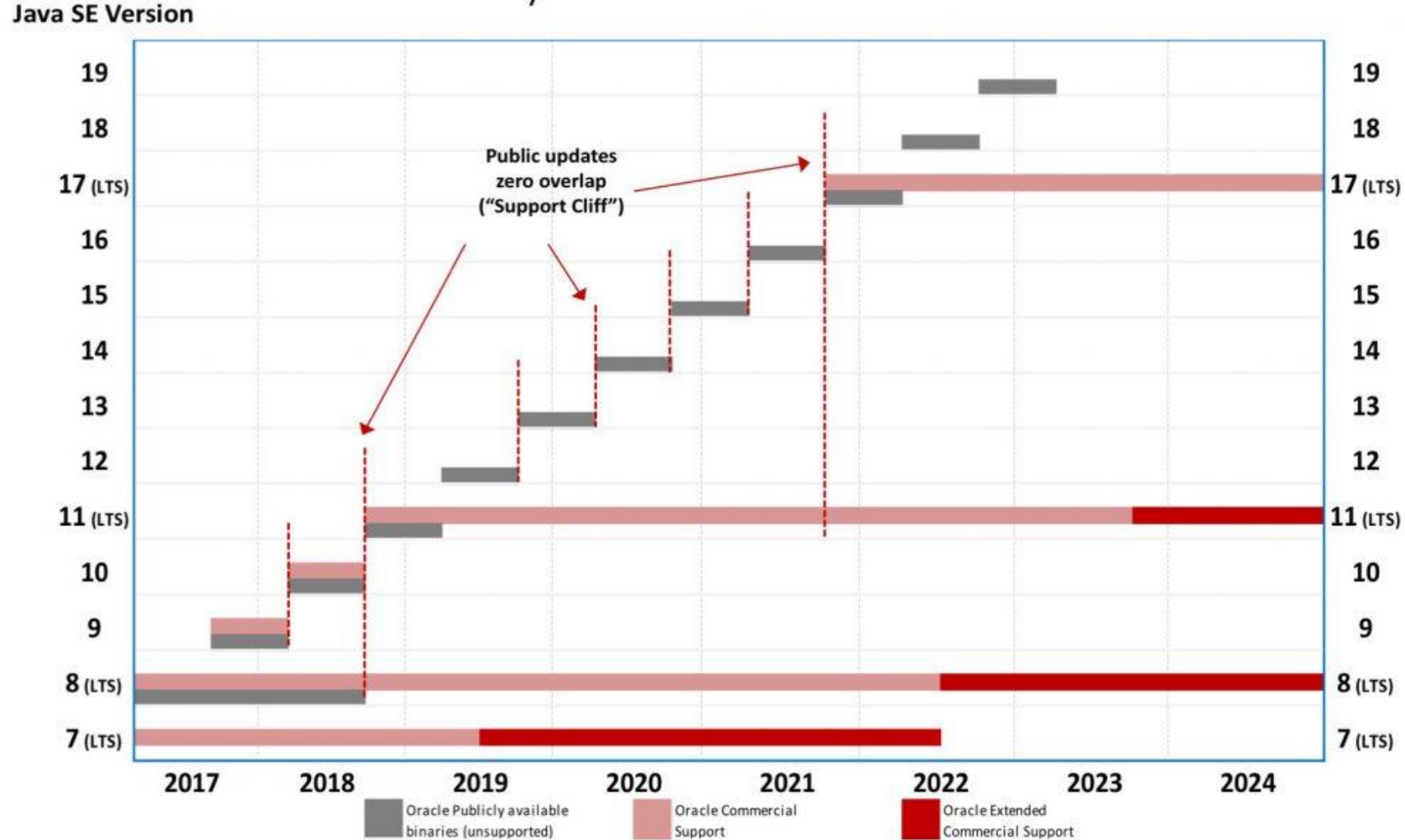
Alexandru Topala @ Telecom Academy

Summary

- ▶ Java Versions
- ▶ Private methods in interfaces (*Java 9*)
- ▶ Resources declared outside of the try-with-resources (*Java 9*)
- ▶ *var* type-reserved word (*Java 10*)
- ▶ Factory methods in collections (*Java 9*)
- ▶ JShell (*Java 9*)
- ▶ Jigsaw Project: Modules (*Java 9*)

Java versions

Java SE Lifecycle – 5+ Year Timeline



Private methods in interfaces

```
public interface A {
```

```
    void m1();
```

} Java 7 and before

```
    default void m2() {
```

```
        System.out.println("Hello");
```

```
    }
```

} Java 8

```
    static void m3() {
```

```
        System.out.println(":)");
```

```
    }
```

```
    private void m4() {
```

```
        System.out.println(":o");
```

```
    }
```

} Java 9

```
}
```

Resources declared outside of the try-with-resources

```
try (FileInputStream fis = new FileInputStream( name: "./file.txt");  
    BufferedInputStream bis = new BufferedInputStream(fis)) {  
  
    //...  
}
```

Starting from Java 7

```
FileInputStream fis =  
    new FileInputStream( name: "./file.txt");
```

```
BufferedInputStream bis =  
    new BufferedInputStream(fis);
```

Starting from Java 9

```
try (fis; bis) {  
    // ...  
}
```

var type-reserved word



```
Map<String, Map.Entry<Character, Integer>> map = new HashMap<>();
```

```
var map = new HashMap<>();
```

var naming rules

```
public class A {  
    public static void main(String[] args) {  
        |    var var = 3;    <  
    }  
  
    private void var() {    <  
        //..  
    }  
  
    class var {    ✖  
        //...  
    }  
}
```

var usage rules

```
public class A {  
  
    var x;      ✖  
  
    public static void main(var[] args) { ✖  
        var y = 3;  ◀  
  
        var z;      ✖  
  
        var a = 0, b = 0; ✖  
  
        var c = "Hello";  
        c = 3.14;    } ✖  
  
        Predicate<Integer> pred = (var x) -> x % 2 == 0;  ◀  
    }  
}
```


Factory methods in collections

```
Set<Integer> set = new HashSet<>();  
set.add(3); set.add(0); set.add(9);
```

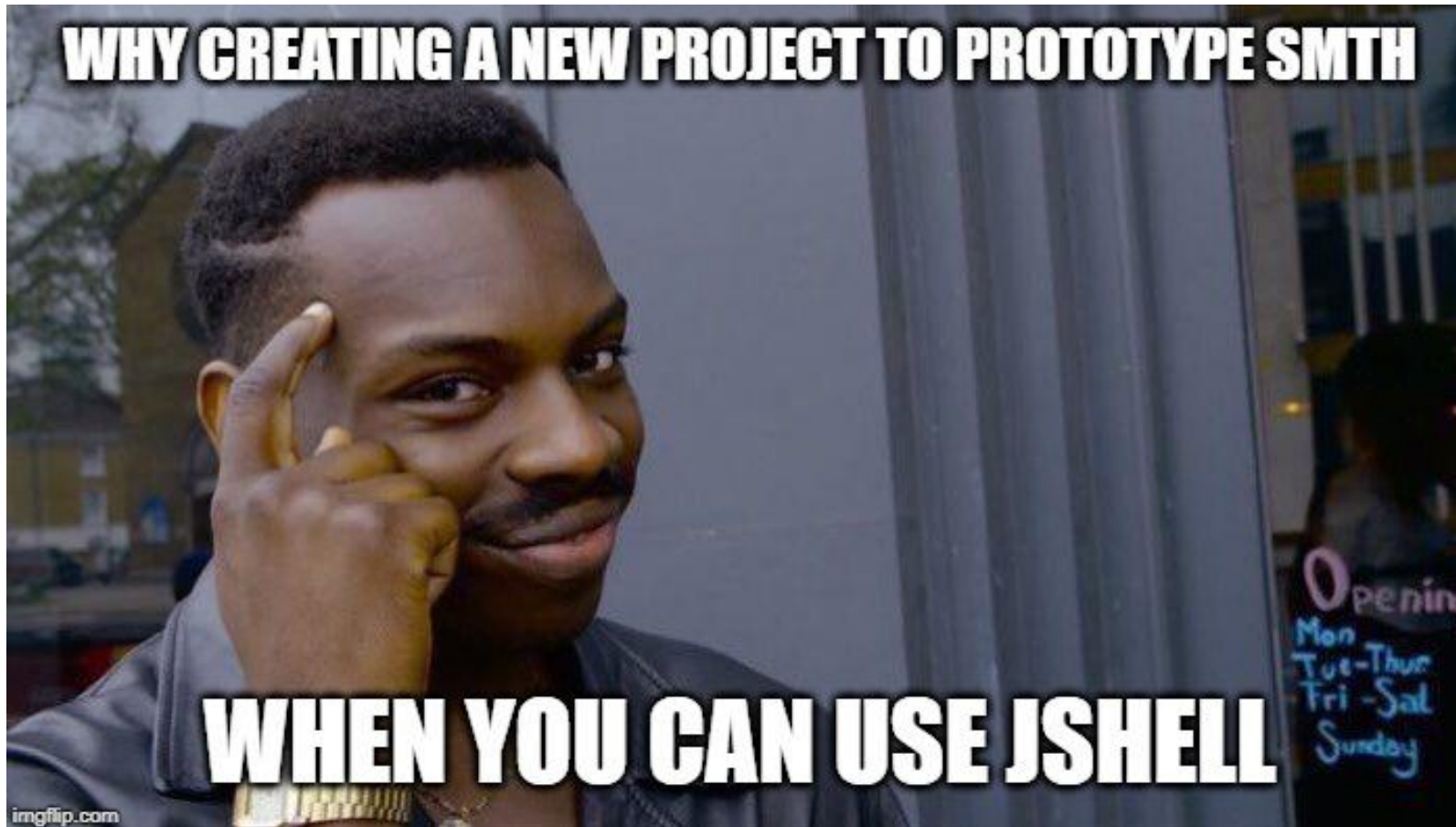
```
Set<Integer> set = new HashSet<>(  
    Arrays.asList(3, 0, 9)  
);
```

```
Set<Integer> set = Set.of(3, 0, 9);
```



JShell

- ▶ a REPL (*Read Evaluate Print Loop*) shell used for testing Java features

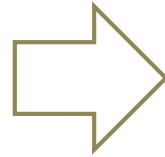


JShell

```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Dell>cd C:\Program Files\Java\jdk-13.0.2\bin

C:\Program Files\Java\jdk-13.0.2\bin>jshell
```



```
C:\Program Files\Java\jdk-13.0.2\bin>jshell
| Welcome to JShell -- Version 13.0.2
| For an introduction type: /help intro

jshell> System.out.println("Hello JShell")
Hello JShell

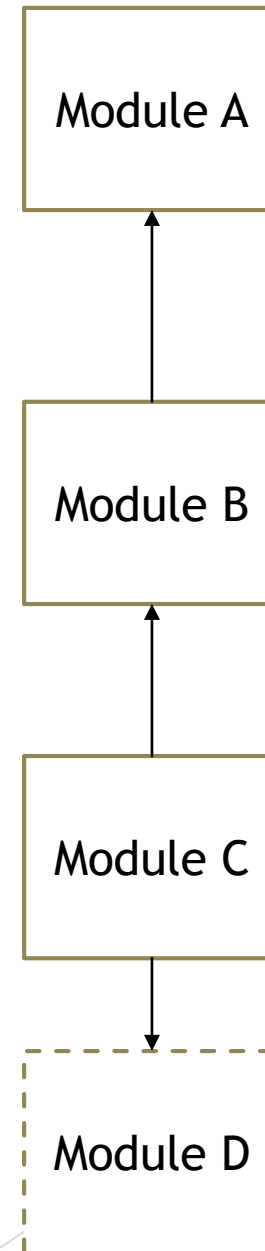
jshell>
```

- */vars* : see variables declared in the current session
- */list* : get the history of executed statements
- */<index>* : execute again the statement indexed by the given *<index>*
- */methods* : see methods defined in current session
- */reset* : erase the data from the current session
- */exit*

Jigsaw Project: Modules

- ▶ Module: set of encapsulated java packages and resources, configured by the *module-info.java* file
- ▶ Advantages
 - ▶ project modularization
 - ▶ strong encapsulation: using modules, we can expose only the functionalities that we want to be exposed. Any other unexposed functionality will not be accessible outside the module, not even with *reflection*.
 - ▶ JDK fragmentation: we can include in our project build only the modules that we need from JDK. This functionality is achieved using JLink.
 - ▶ Dependencies checking at runtime

```
]module A {  
    exports service; // exposing all the public structures  
                      // (classes, interfaces, etc.) from the  
                      // "service" package from module "A"  
  
    requires java.base; // base java module, required by default  
                        // (like java.lang package)  
  
}  
  
]module B {  
    requires transitive A; // declares a dependency on module "A".  
                          // "transitive" means that all the modules  
                          // that will depend on "B", will also depend  
                          // on "A" from now on  
  
}  
  
]module C {  
    requires B; // implicitly, "C" is requiring "A"  
               // because "B" requires "A" in a  
               // transitive manner  
  
    requires static D; // "static" means that the dependency  
                      // on "D" is optional, so the application  
                      // will start even if "D" doesn't exist  
  
}
```



Service Loader with modules

- Used to get all the implementations available for a given interface, without knowing the implementing class and without requiring the module that exposes the implementation

Suppose we have the following:

A service interface inside "service" package, in module "A"

The "EnglishHelloService" inside module "B"

The "RomanianHelloService" inside module "C"

```
public interface HelloService {  
    String sayHello(String name);  
}  
  
public class EnglishHelloService implements HelloService {  
    @Override  
    public String sayHello(String name) {  
        return "Hello, " + name + "!";  
    }  
}  
  
public class RomaniaHelloService implements HelloService {  
    @Override  
    public String sayHello(String name) {  
        return "Buna, " + name + "!";  
    }  
}
```



```
}module A {  
    exports service; // exposing the "service" package, where  
                    // the "HelloService" interface resides  
  
    uses HelloService; // this statement is telling to the compiler  
                      // that module "A" is looking for implementations  
                      // of "HelloService" interface  
}
```

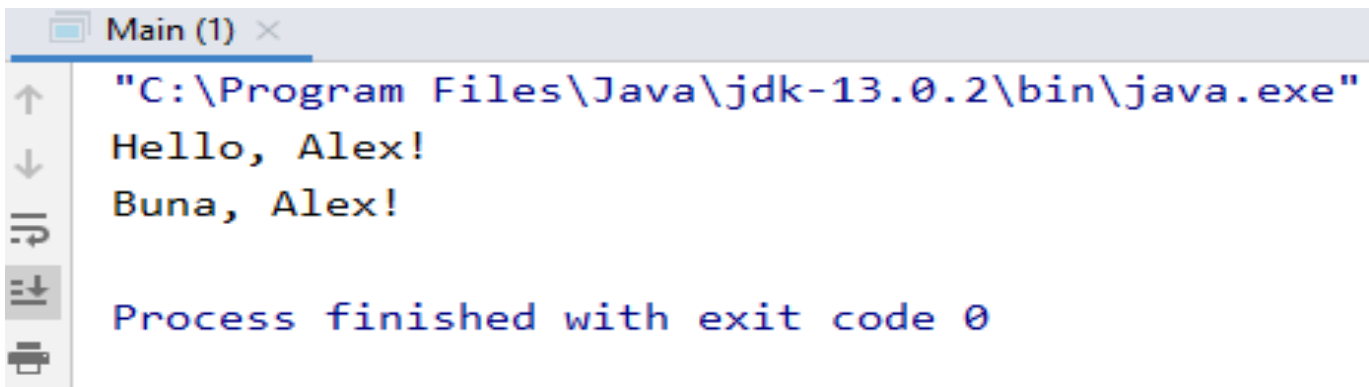
```
}module B {  
    requires A; // used to have access to the "HelloService"  
              // interface, which is inside module "A"  
  
    // exposing an implementation of "HelloService" interface  
    provides HelloService with b.service.impl.EnglishHelloService;  
  
}
```

```
}module C {  
    requires A; // used to have access to the "HelloService"  
              // interface, which is inside module "A"  
  
    // exposing an implementation of "HelloService" interface  
    provides HelloService with c.service.impl.RomaniaHelloService;  
  
}
```

- Now we can obtain the required implementations of the *"HelloService"* interface using the *"ServiceLoader"* class

```
public static void main(String[] args) {  
    ServiceLoader<HelloService> services =  
        ServiceLoader.load(HelloService.class);  
  
    services.stream().map(ServiceLoader.Provider::get).map(helloService -> helloService.sayHello(name: "Alex")).forEach(System.out::println);  
}
```

- After executing the main method above, the output will consist in the corresponding *"hello"* messages from the existing implementations of *"HelloService"*: *"EnglishHelloService"* and *"RomanianHelloService"*



```
Main (1) x  
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe"  
Hello, Alex!  
Buna, Alex!  
  
Process finished with exit code 0
```