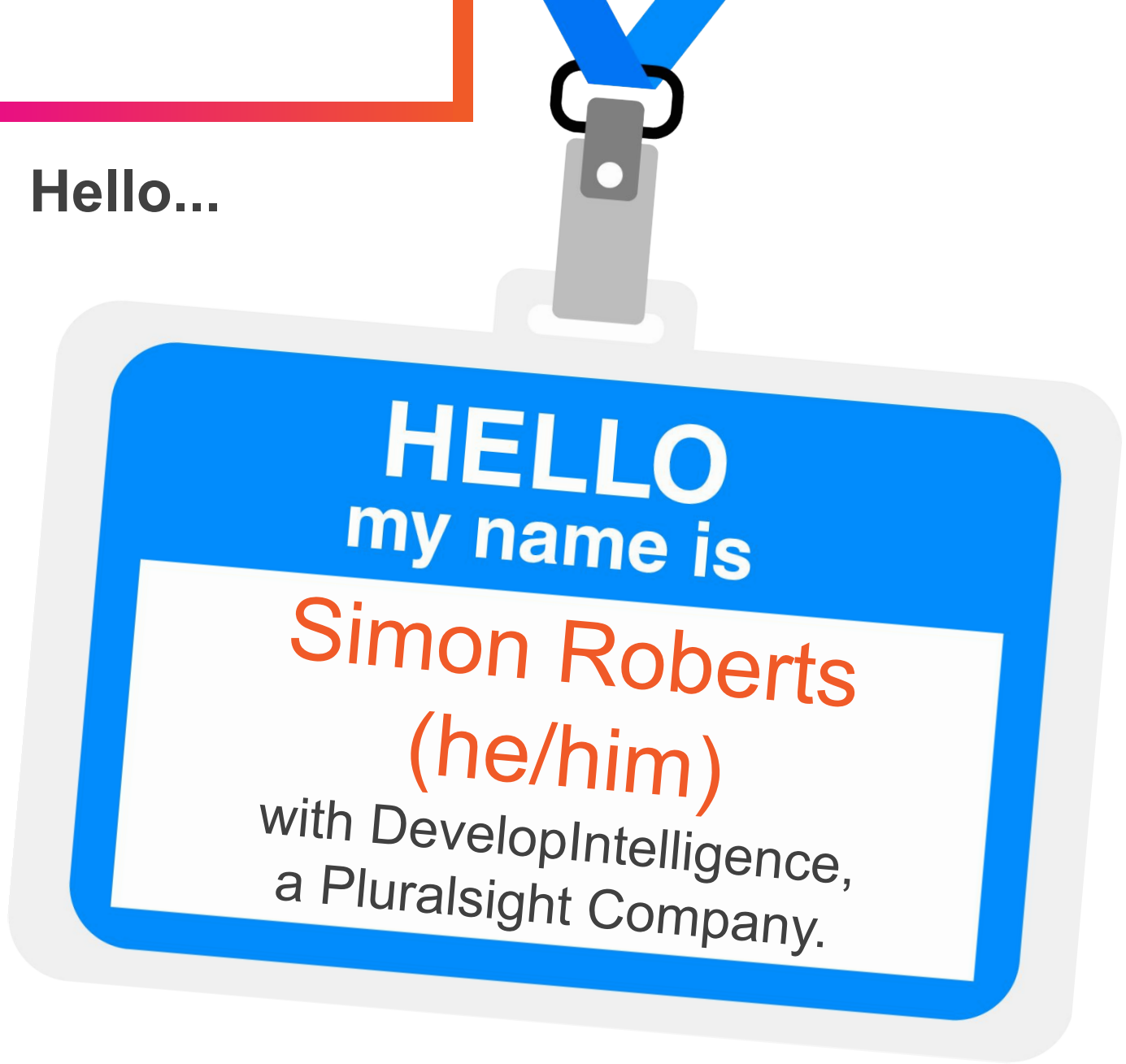
Welcome

Advanced Java TT
JPMS & var

DevelopIntelligence
A PLURALSIGHT COMPANY

Hello...



About me...

We teach over 400 technology topics

You experience our impact on a daily basis!

Prerequisites

This course assumes you

- Good understanding of the Java programming language to Java 8
- Understanding of issues surrounding project dependencies and their management

Why study this subject?

- The module system can improve security, startup speed, distribution footprint
- The var pseudotype can reduce verbosity in your code

My pledge to you

I will...

- Make this interactive
- Ask you questions
- Ensure everyone can speak
- Create an inclusive learning environment
- Use an on-screen timer for breaks

…also, if you have an accessibility need, please let me know

Objectives

At the end of this course you will be able to:

- Create modular software using the Java Platform Module System
- Use the var pseudo-type

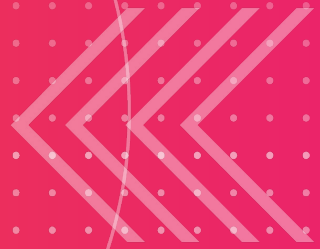
How we’re going to work together

- Discussions, whiteboard diagrams
- Code examples
- You'll have a copy of all the course materials in github
    - Please note, the git repository will be deleted–clone it if you want it!

Student Introductions

- Job title?
- Where are you based?
- Experience with Java
- Fun fact?

Thank you!



DevelopIntelligence
A PLURALSIGHT COMPANY

Problems JPMS Addresses

- Project organization / dependencies between components
- Access control for components larger than classes
- Control of reflection
- Provision of, and access to, services
- Migration of non-modular projects to the module system

Declaring a module

Develop Intelligence

A module is created by the presence of module-info.java or module-info.class in the root directory of the package tree.

- This file indicates the name of the module
- along with access control directives

```
module <name> {
   // directives
}
```

Module names are generally dotted lower-case names, in a manner similar to packages, e.g. **`mycompany.accounting`**

Key directives: `exports`

Develop Intelligence

A module that permits access to some features uses the exports directive to give access to a ***package***:

```
module my.mod {
   exports <package-name> [to mod1, mod2…]
}
```

The `exports` targets (if present) may include unavailable / non-existent modules

To export multiple packages, use multiple exports directives

Key directives: `requires`

Develop Intelligence

A module that wishes to use features from another module issues a requires directive naming the target module:

```
module my.mod {
   requires <module-name>
}
```

The `requires` target must exist and be available or compilation / execution will fail

All modules implicitly declare `requires java.base;`

To declare dependency on multiple modules, use multiple `requires` directives

Finding modules and classes

Develop Intelligence

The JVM searches for modules on the **module-path**

- specify as a separated list using the command line parameter -p or --module-path
- separator is colon or semi-colon depending on the host platform OS

Classes are not simply loaded from any/all available modules, instead a module graph is built transitively from the `requires` directives found, starting at the root module(s)

- a key root module will be specified on the command line with the program entry point
- classes are only loaded from modules that are determined to be necessary
- this speeds up loading and can reduce distribution size of software

Reading vs requiring a module

If a module has the right to access exported features of a module, it is said to "read" the module

- The `requires` directive implies reading the specified module
- The `requires` directive additionally puts that module into the graph of modules to load classes from

Opening a package or module to reflection

Develop Intelligence

In the module system a module is protected against reflection from outside that module by default

- enabling reflection is referred to as "opening"
- an entire module can be declared as open
- or individual packages can be declared as open

```
open module reflectable.mod {}


module partly.reflectable {
   opens some.package;
}
```

Declaring and implementing a service

Develop Intelligence

Services are defined by types that describe their features (e.g. methods)

- this is nothing more than providing a type (class, abstract class, interface) in a public package and exporting that package

Service can have multiple implementations, perhaps in the declaring module, perhaps elsewhere.

- These are announced using the **`provides`** directive

```
module offer.a.service {
    provides serv.if.ServiceIF with my.serv.ServImpl
}
```

- Implementations need not be directly visible to the service clients

Using a service

Develop Intelligence

A module wishing to use a service declares its intention:

```
module uses.a.service {
   requires module.declaring.service.type;
   uses serv.if.ServiceIF;
}
```

The available service implementations are loaded by the `ServiceLoader`:

```
ServiceLoader<ServiceIF> loader =
            ServiceLoader.load(ServiceIF.class);
for (ServiceIF srv : loader) {
   // investigate / use the various implementations
}
```

Module system command line parameters

Several command line parameters support compiling & running with JPMS

- **`--module-path`** or **`-p`** -- where to find binary modules
- **`--module`** or **`-m`**
  - which module/class to launch a program from
    `java [...] -m my.module/your.pak.MainClass`
  - which modules to compile in a multi-module compilation
    `javac [...] -m one.module,other.module`
  - note: NO SPACES around the comma!

Module system command line parameters

- **`--module-source-path`** -- describes how to find the "roots" of source directory trees for compilation of modular projects
  - an asterisk in this specification will be substituted with the name of the module to be compiled
  - usually needs to be surrounded by quotes to avoid OS shell handling of *
  - can be a list of paths with OS specific separator (colon/semicolon)

E.g.

```
javac \
  -d modules \
  --module-path modules \
  --module-source-path "mod-srcs/*/src/main/java" \
  --module module.one,module.two
```

Using the pseudo type **`var`**

Develop Intelligence

Java 10 and 11 added the pseudo-type **`var`** to the language

- This is not a keyword, but has special meaning only in the places a typename is expected

The var pseudo type can be used in place of an explicit type specification for:

- local variables that are declared and initialized with an unambiguous value
  - but cannot be used for field declarations
- unambiguously initialized formal-parameter-like variables
  - e.g. in for loops, try with resources
  - but not in regular method arguments nor catch parameters because these are not initialized
- formal parameters for lambdas, provided the type is inferrable
  - this serves as a placeholder for annotation

Using the pseudo type **`var`**

Develop Intelligence

`var` can express types that are "not denotable" in the regular scheme of the language

- For example:

```
var x = true ? "" : 0;
```

- declares x to be the intersection of all the interfaces of `String` and `Integer`
- This allows `x` to be treated as (e.g.) `Comparable`, which would not be possible if `x` were simply declared as `Object`