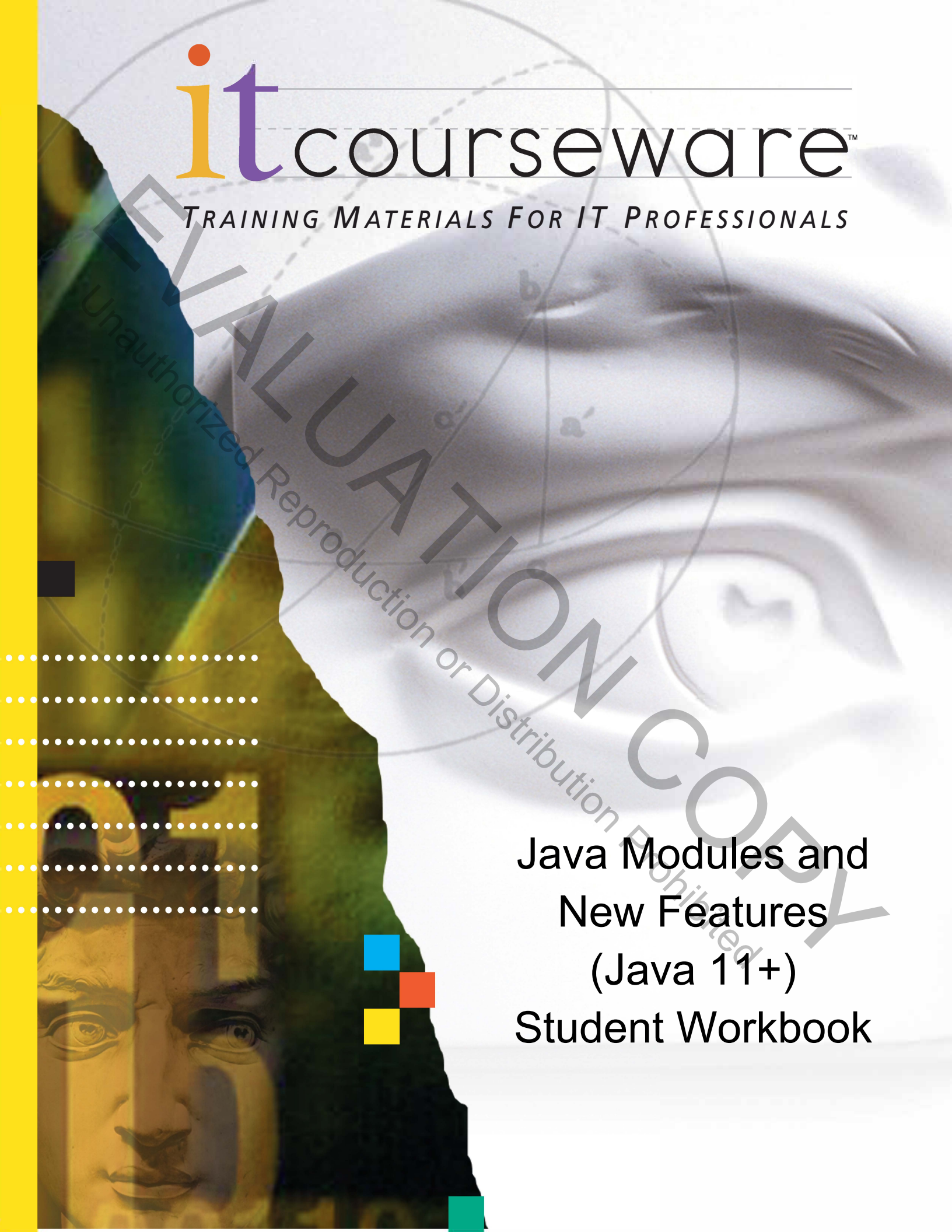


itcourseware™

TRAINING MATERIALS FOR IT PROFESSIONALS

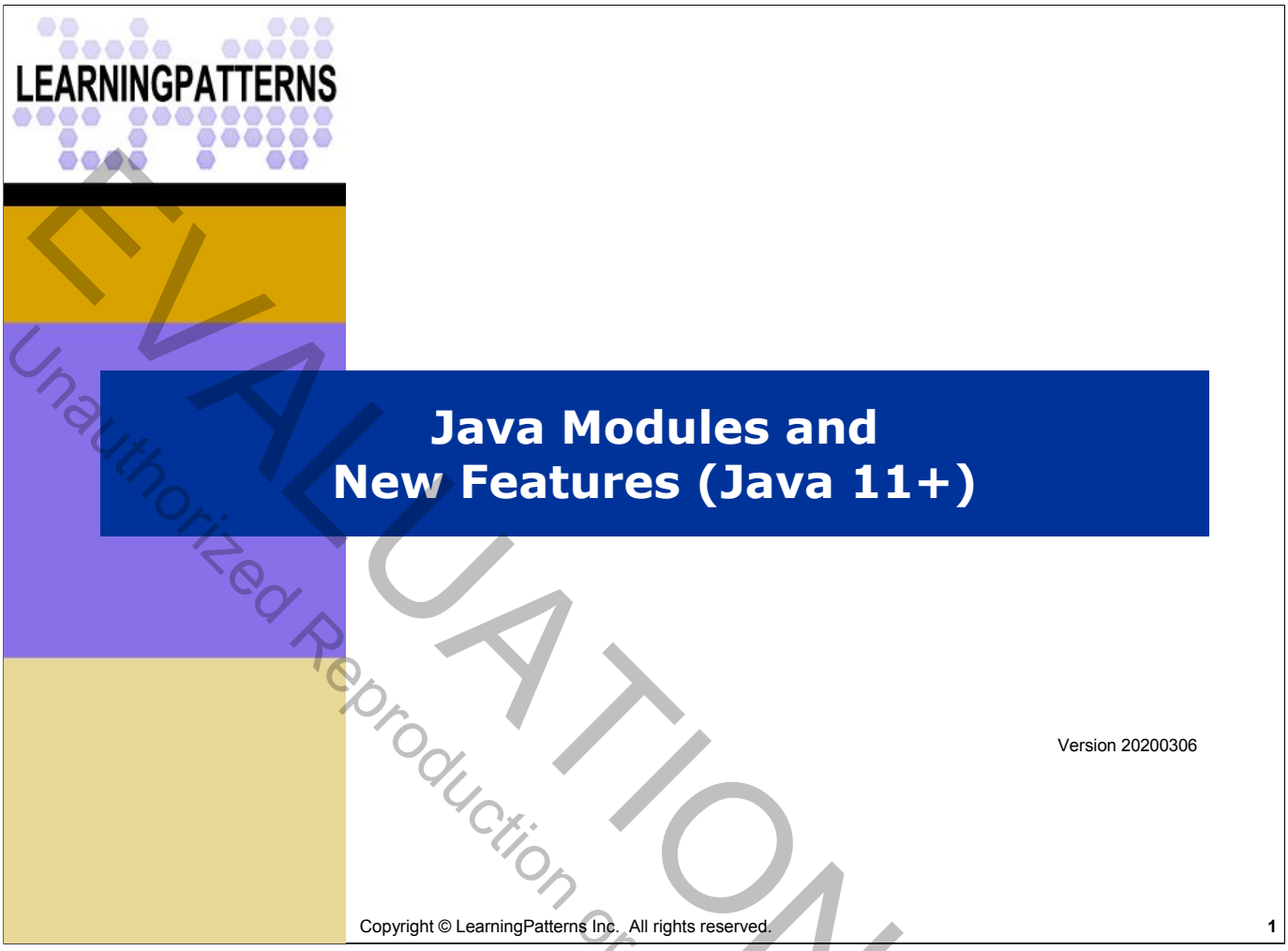
Java Modules and New Features (Java 11+) Student Workbook

For more information about Java Enterprise Java, or related courseware, please contact us. Our courses are available globally for license, customization and/or purchase.

LearningPatterns. Inc. Services@learningpatterns.com | www.learningpatterns.com

Global Courseware Services 982 Main St. Ste. 4-167 | Fishkill NY, 12524 USA
212.487.9064 voice and fax

Table of Contents – Java Modules and New Features (Java 11+)



Notes:

- Version 20200306

Course Agenda

- Preface: **Java State of the Union**
- Session 1: **Introduction to Modules**
- Session 2: **Working with Modules**
- Session 3: **Type Inference**
- Session 4: **JShell**
- Session 5: **HTTP Client**
- Session 6: **Other New Features**
- Session 7: **Custom Runtime Images**

Notes:

Typographic Conventions

- Code in the text uses a fixed-width font

  `Catalog catalog = new CatalogImpl()`
- Code fragments are the same, e.g., `catalog.findById(4)`
- We **bold/color** text for emphasis
- ***New terms*** are often introduced in ***bold italics***
  - *Filenames* and *paths* are also in italics, e.g., *Catalog.java*
- Notes are indicated with a superscript number (1) or a **star ***
- Longer code examples appear in a separate code box (below)

```
package com.javatunes.catalog;

public class CatalogImpl implements Catalog {
   public MusicItem findById(long id) {
      ...
   }
}
```

Introduction 3

Notes:

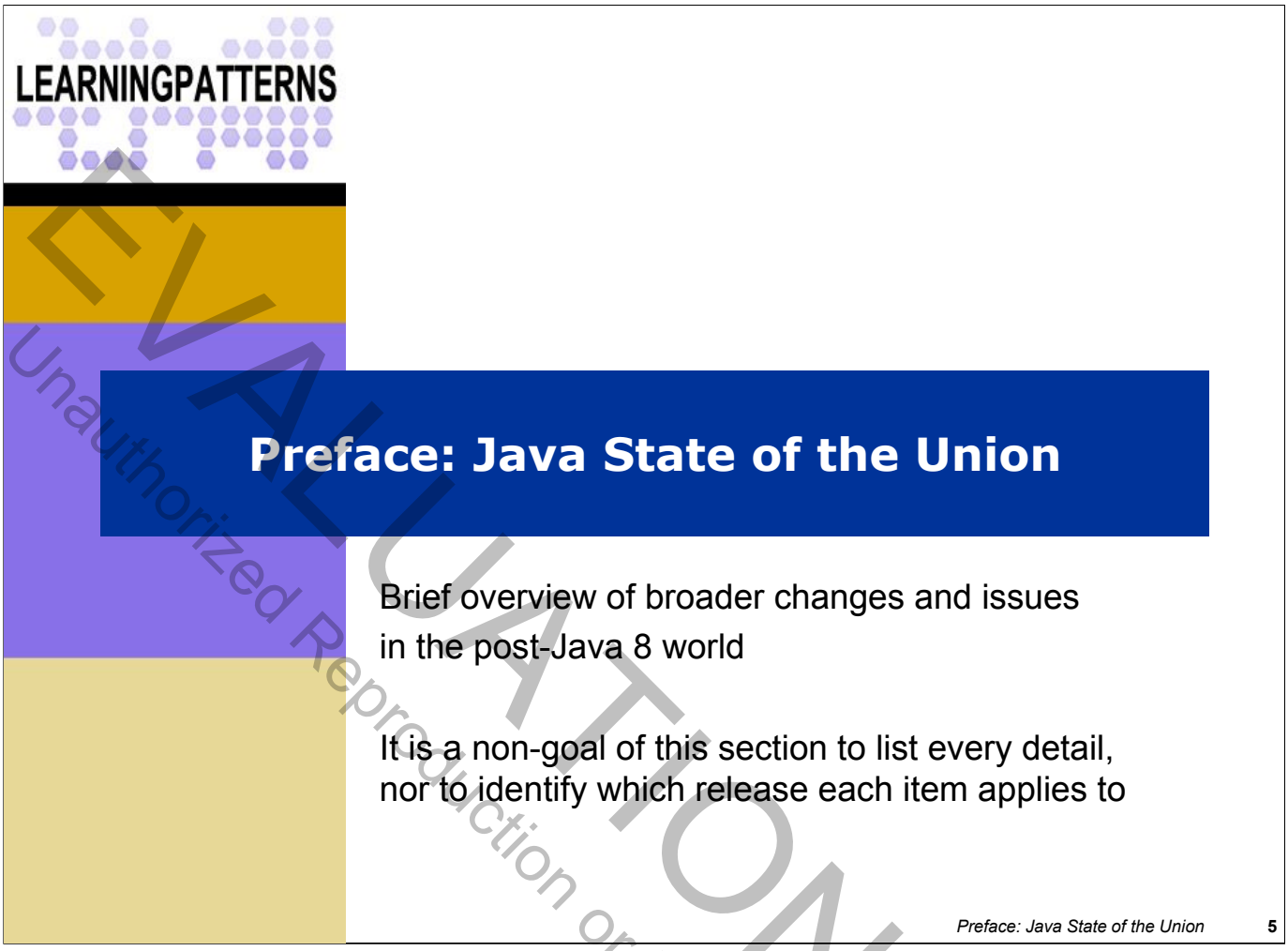
(1) If we had additional information about a particular item in the slide, it would appear here in the notes, with the appropriate superscript.

- We might also include related information that pertains more generally to the slide.

Labs

- The course has numerous hands-on lab exercises
  - Many use types from a fictional case study called **JavaTunes**
    - An online music store
  - The lab instructions are separate from the main manual
- Setup files are provided with skeleton code for the labs
  - Students add code focused on the topic at hand
  - There is a solution zip with completed lab code
- Lab slides have an icon in the upper right corner of the slide
  - The end of a lab is marked with a stop sign

Notes:



Notes:

New Java Release Cycle

- New "major" version every 6 months – **March** and **September**
  - Started with JDK 9 in Sep 2017
  - Two kinds of releases, detailed below
- **Feature Release**
  - Implementation enhancements, possible new language features
  - Not intended for long-term use
  - **Superseded** after 6 months, with release of next version
- **Long-Term-Support (LTS) Release**
  - This is what you're accustomed to working with
  - Plan is to mark a release as LTS every **3 years**
- Details in Oracle Java SE Support Roadmap [see notes]

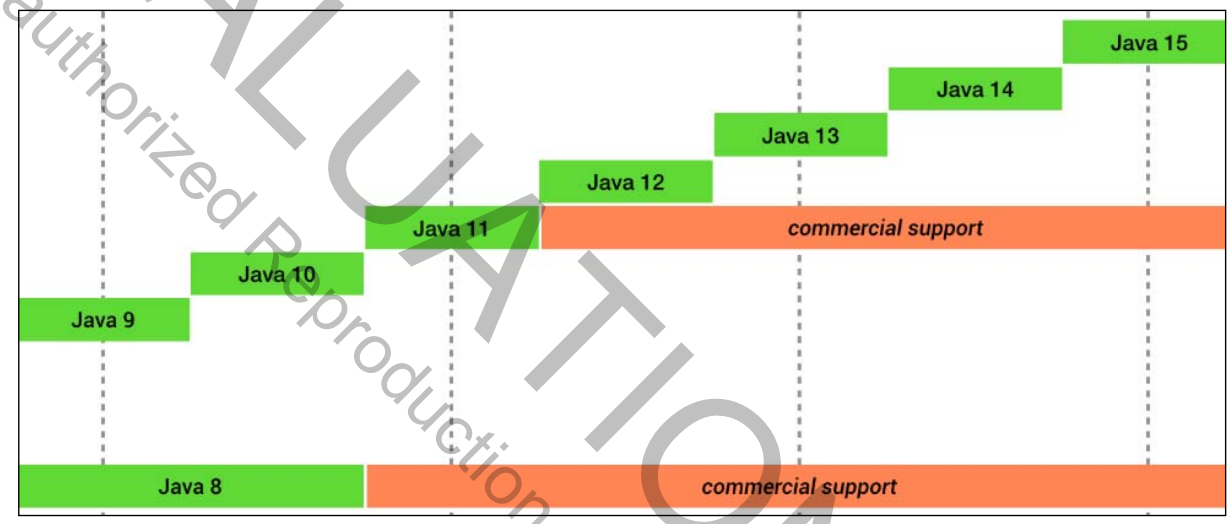
Preface: Java State of the Union 6

Notes:

- From the **Oracle Java SE Support Roadmap** at *http://oracle.com/technetwork/java/java-se-support-roadmap.html*.
  - For product releases after Java SE 8, Oracle will designate a release, every three years, as a Long-Term-Support (LTS) release. Java SE 11 is an LTS release. For the purposes of Oracle Premier Support, non-LTS releases are considered a cumulative set of implementation enhancements of the most recent LTS release. Once a new feature release is made available, any previous non-LTS release will be considered superseded. For example, Java SE 9 was a non-LTS release and immediately superseded by Java SE 10 (also non-LTS), Java SE 10 in turn is immediately superseded by Java SE 11.
- This is a **very** important document, packed with detail (and myriad footnotes). If you are responsible for maintenance and updates at your organization, you really need to read it in its entirety.

Versions and Support – Illustrated

- Except for LTS releases, these are really just 6-month "technology previews"
  - Official term is "feature release"
- You can see that LTS releases span several feature releases



Preface: Java State of the Union 7

Notes:

JDK Update Schedule – By Example

- Non-LTS and LTS releases are more frequent now
  - **9.0** **Sep 2017** **March** and **September** each year
    9.0.1 Oct 2017 one month later, coincides with below
    9.0.2 Jan 2018 **Jan-Apr-Jul-Oct** update schedule
  - **10.0** **Mar 2018** JDK 9 is **dead**
    10.0.1 Apr 2018
    10.0.2 Jul 2018
  - **11.0 LTS** **Sep 2018** next LTS in **3 years**, JDK 10 is **dead**
    11.0.1 Oct 2018
    11.0.2 Jan 2019
    11.0.3 Apr 2019
    11.0.4 Jul 2019
    11.0.5 Oct 2019
    etc., until next LTS
    - **12.0** **Mar 2019**
    - **13.0** **Sep 2019**
    - **…**
    - **17.0 LTS** **Sep. 2021**

Preface: Java State of the Union 8

Notes:

- Look it over closely – see how it works?
- This release schedule offers some nice benefits, in that updates are more frequent, and predictable. (Though the versioning and non-LTS vs. LTS releases will take some getting used to.)
- As a consequence, with more frequent releases, it can be more difficult to quickly find the applicable documentation for a specific release. Of course you can "google it," but there is a better way, built in to any JDK 11+ distribution:
  - Release Notes and other docs for all releases can be found by simply starting at your JDK installation, and opening *<jdk-install>/README.html*, then click and browse from there.
    - Goes all the way back to JDK 6!
    - Can be hard to find on some installations (e.g. Mac)
  - Can also go to https://java.com/licensereadme, which forwards to an oracle page with links to all Java installations going back to JDK 6.
  - Oracle page is https://www.oracle.com/technetwork/java/javase/terms/readme/index.html

Java Modules and JDK Version Numbers

- We divide the world into two: **pre-** and **post-modules**
  - **Java 8** and earlier is **pre-modules**
  - **Java 9** and later is **post-modules**
- **Java 11** is the current LTS release post-Java 8
  - It's the base version and main focus of this course (1)
  - Many post-11 releases - we cover some "goodies" from them
- For simplicity in the course, we'll use terms like ***"Java 8 code"*** and ***"Java 9 code,"*** etc.
  - "Java 8 code" = any code pre-modules, on the classpath
    - For example, your existing production systems
  - "Java 9 code" = any code organized into a module
    - Also called ***modular code***, deployed in a ***modular JAR*** file

Preface: Java State of the Union 9

Notes:

(1) You should **not** be using Java 9, 10, 12, 13, or any non-LTS release in production.

- These are all short-term "technology preview" or "feature" releases.
- Support for this type of release ends 6 months after the release date, coinciding with the release of the next version. If used in production, that would mean you lose support after 6 months. Not a good idea!
- This is why we use Java 11 as the main focus of the course. We do, however, show some features from later versions so you know what will be coming in Java 17 (when it does arrive). None of them are compelling enough (so far) to use a non-LTS release in production.

- "Java 8 code" includes **all code** written and compiled with **any JDK earlier than 9**.

What's In

- **Java Modules!**
  - **All code** now resides in a module (of some type)
    - **Including** all your Java 8 code written years ago – hmm....
  - We now have ***modulepath*** in addition to the classpath
  - New JDK command-line tools, enhancements to existing ones

JPMS = Java Platform Module System

- **JShell** – REPL shell (Read-Eval-Print Loop) for quick-coding
  - And running scripts
- New **HTTP Client**
  - Replaces outdated `HttpURLConnection`
- **Reactive Streams**
  - Non-blocking publish-subscribe style stream processing

Preface: Java State of the Union 10

Notes:

- The Java Platform Module System (JPMS), initially called Project Jigsaw, appeared in Java 9.
  - It is by far the Number One Most Important New Feature, and a complete game-changer, with the potential to affect nearly every aspect of Java development and runtime execution – or not...
  - As we'll see, even if you "opt out" of creating your own modules, you absolutely still need to know how Java Modules works, as well as understand the other major changes that have taken place in the Java 9 platform.

- As we'll soon learn, the Java runtime now treats all code as "in-a-module," including Java 8 code written long ago.
  - This is done via ***automatic modules*** and ***the unnamed module***, which exist primarily for migration purposes.
  - Over time, the use of these two types of modules should decrease, as existing code (and libraries) gets "modularized" into true ***named application modules***, and new code is written in modules from the beginning. Details forthcoming!

What's In

- Other enhancements
  - **Collection factory methods**
  - **Multi-release JARs** (MR-JARs)
  - **Type inferencing** – Java has a **`var`** now
  - Miscellaneous capabilities
    - **Process API** updates, new **Stack-Walking API**
    - Java 12+: **Switch Expressions**, **Compact Number** Formatting

```
List.of("list", "of", "strings")
```

- Implementation and monitoring enhancements [see notes]
  - **G1** is the default collector + new Epsilon GC (no-op), and ZGC
  - Java Flight Recorder donated to open source
- New compiler option **`--release`**
  - Replaces legacy `-source` and `-target` flags

Preface: Java State of the Union 11

Notes:

- The JVM continues to be improved in terms of scalability (both down and up) and performance. These enhancements come in forms that we as engineers can get our hands on and configure, such as garbage collectors, and also as compile and runtime changes at the bytecode generation and execution level, respectively. We won't address the latter at all. On the GC front, in addition to improvements in G1 (which is now the default), there are some interesting new ones available:
  - The **Epsilon** garbage collector is a **no-op**, particularly relevant for short-lived programs that don't benefit from the cost of GC. *http://openjdk.java.net/jeps/318*.
  - **ZGC** is a low-latency concurrent collector targeted at very large heap applications, particularly installations of big data software like Hadoop. *http://openjdk.java.net/jeps/333*.
- Java Flight Recorder has been donated to open source (no longer requiring a commercial license). *http://openjdk.java.net/jeps/328*.

What's Out

- CORBA packages (see ya!)

hey, what's CORBA?

- Other overlapping APIs from Java EE
  - Java 9-10 contained them in the **`java.se.ee`** module, which was marked "deprecated for removal"
    - `java.activation`
    - `java.corba`
    - `java.transaction`
    - **`java.xml.ws`**
    - `java.xml.ws.annotation`
    - **`java.xml.bind`**
  - **Physically gone in 11**
    - Along with their associated JDK command line tools [see notes]

these are their **module names** from **`java.se.ee`** in JDK 9-10

- Most likely to affect you is removal of JAXB and JAX-WS

Preface: Java State of the Union 12

Notes:

- The following JAX-WS tools have been removed:
  - `wsgen`
  - `wsimport`
- The following JAXB tools have been removed:
  - `schemagen`
  - `xjc`

What's Out

- CMS garbage collector is deprecated
  - Support to cease in JDK 14
  - G1 is now the default, and long-term replacement
    - Been around several years now
- Some items still "there" but you can't use them (1)
  - Internal JDK APIs that are now encapsulated in modules
    - Physically, their JARs (*rt.jar*, *tools.jar*, etc.) are gone
- Some items just physically need to go (see ya!)
  - Rarely-used GC combinations that were deprecated in Java 8 (2)
  - Endorsed Standard and Installed Extension mechanisms
    - `java.endorsed.dirs` and `java.ext.dirs` (1)
  - Outdated JDK command-line tools

Preface: Java State of the Union 13

Notes:

(1) JEP 260 (Encapsulate Most Internal APIs) states: "...make most of the JDK's internal APIs inaccessible by default [in JDK 9], but leave a few critical, widely-used internal APIs accessible, until supported replacements exist..."

JEP 220 (Modular Run-Time Images) removes *rt.jar*, *tools.jar*, etc., stating that, "some popular libraries make use of non-standard, unstable, and unsupported APIs that are internal implementation details of the JDK and were never intended for external use."

JEP 220 also removed both the Endorsed Standard and the Installed Extension mechanisms. Upgradeable Modules are intended to replace the Endorsed Standard mechanism, and Installed Extension is removed with no replacement ("we have seen little evidence of its use").

(2) The following GC combinations were deprecated in Java 8 – *http://openjdk.java.net/jeps/173*:

- Incremental CMS, DefNew + CMS, ParNew + SerialOld.

A summary of the JVM flags that configure these combinations can be found here: *http://openjdk.java.net/jeps/214*.

What's Out

- **Applets** deprecated in Java 9 and removed in 11
- **Java Web Start** removed in Java 11, with no replacement
- **JavaFX** extracted out to OpenFX – ***http://openjfx.io***
- *Java Client Roadmap Update* document describes all of this
  - Included in our lab bundle
- Java VisualVM (`jvisualvm`) and Java Mission Control (`jmc`)
  - Extracted out, separate downloads now
  - Java Flight Recorder donated to open source
  - Mostly done to align Oracle JDK with OpenJDK
    - As of Java 11, these two are now "functionally equivalent, with cosmetic differences"
- No more JRE
  - Instead, build a (smaller!) custom runtime image with **`jlink`**

Preface: Java State of the Union 14

Notes:

- **Note** that with all the client-side UI pieces being extracted out, Swing and AWT are **still in**.
  - Don't need 'em in your application? See "No more JRE" in the slide above...

- Note that while JVisualVM (`jvisualvm`) and Java Mission Control (`jmc`) are no longer part of JDK 11, and Java Flight Recorder has been donated to open source (no longer requiring a commercial license), the old standby JConsole (`jconsole`) is still included.

Deprecated – for Removal

- Number of deprecated classes / methods is always increasing
  - Java 6 416 items
  - Java 7 425
  - Java 8 464
  - Java 9 627 for removal 24
  - ...
- Java 9 introduced new "attributes" of deprecation
  - **"since"** and **"for-removal"** now included in a deprecation

```
@Deprecated(since="1.2", forRemoval=true)
```

- Java 9 marked 24 items for actual removal in future releases
  - Some going all the way back to JDK 1.1 and 1.2 (!)
    - Time to finally take this seriously and change your old code...
- Java 11 took this to conclusion, **physically removing them**

Preface: Java State of the Union 15

Notes:

- The figures above are cumulative, and were obtained via Java 9's new JDK tool **`jdeprscan`**:
  - `jdeprscan --list --release 6` (release is 6|7|8|9)
  - `jdeprscan --list --release 7`
  - `jdeprscan --list --release 8`
  - `jdeprscan --list --release 9`
  - `jdeprscan --list --release 9 --for-removal`
    - The `--for-removal` option is only applicable for release 9 or later (when "removal" was introduced).

- The main use case for `jdeprscan` is for you to run it on **your code**, and it will tell you all the places you're using deprecated items. (So now you really have no excuse...)
  - The commands listed above specify the `--list` option, which simply lists all the deprecations in the Java platform.

When to Move to Java Modules?

- The Java Module System is designed to be completely "opt in"
  - Some are moving right away
  - Some are waiting for more maturity / direction from others
- Using 3rd party libraries as modules will evolve (for the better)
  - Adoption has been slow, perhaps one reason to wait a bit
- Our take:
  - Get familiar with modules (it's here to stay)
  - Get familiar with the modularity in the Java platform itself, and how this affects your code (because it does!)
  - If you make the leap to "modularize" your own code:
    - Start small, e.g., common / core libraries, smaller applications
    - You'll need to deal with your third party dependencies, too

Notes:

Thinking in Java 9+

- JDK command-line tools have evolved over time
  - More options / flags than ever before
  - Many of these can now be expressed in several forms:

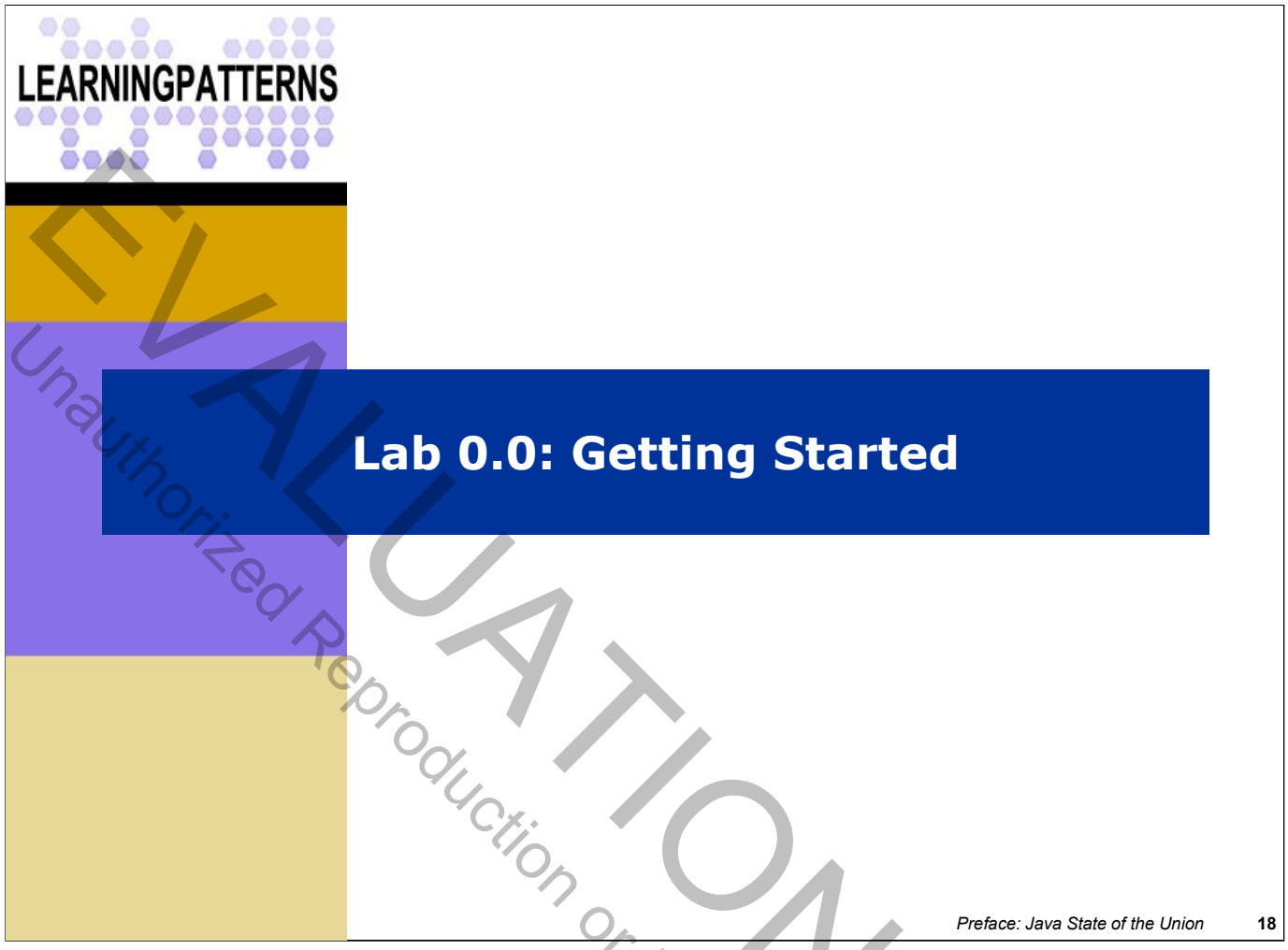
```
$ java -cp
$ java -classpath
$ java --class-path     // this is the new form

$ java -p
$ java --module-path    // this is the new form
```

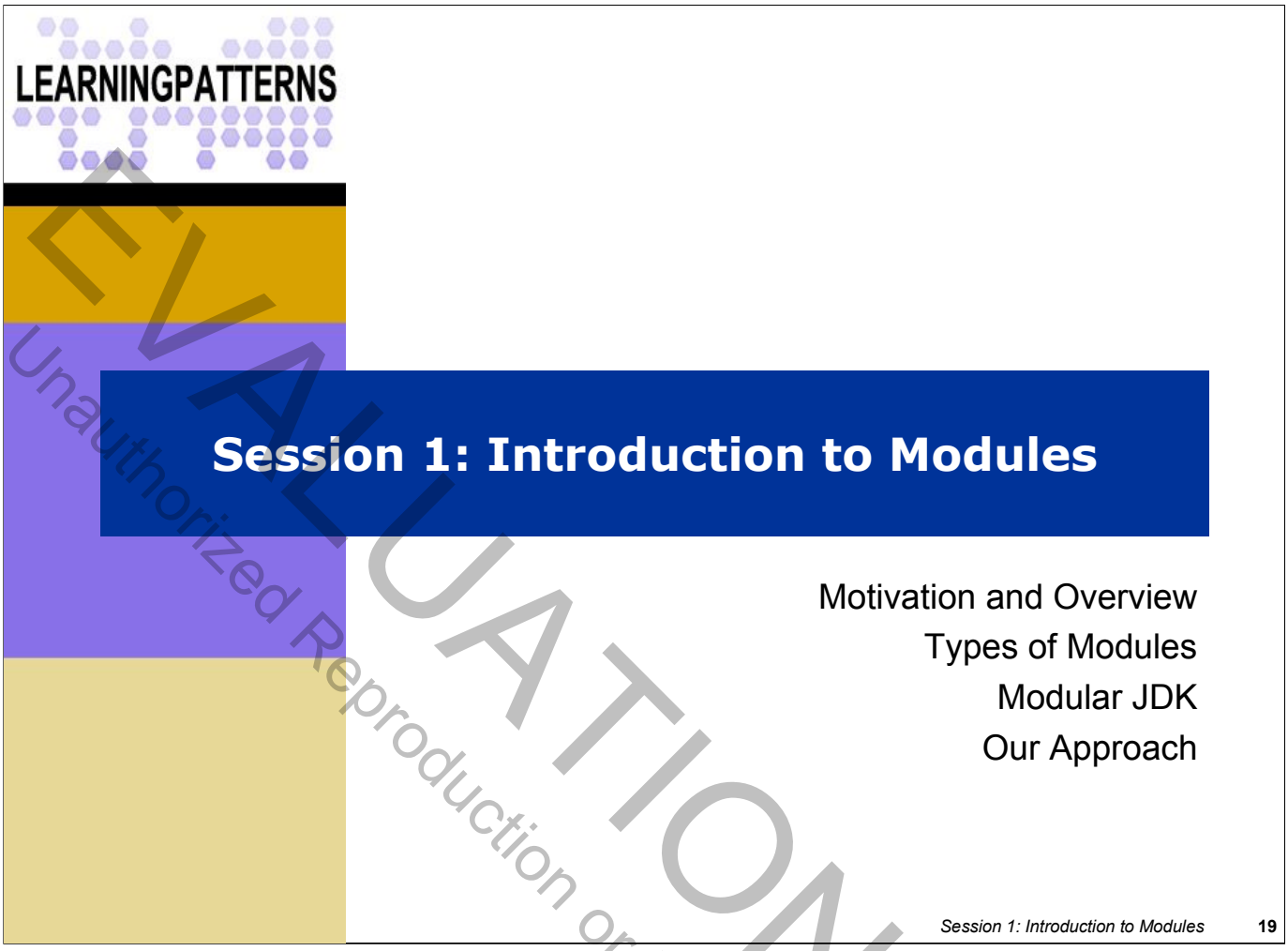
  - **NOTE:** since some of the new options offer only the new form, we will be favoring this one (in the slide examples and labs)

- Course focus is on "modular" code, i.e., code in a module
  - BUT: all the other platform improvements are available, even if you don't "opt in" and modularize your code

Notes:



Notes:

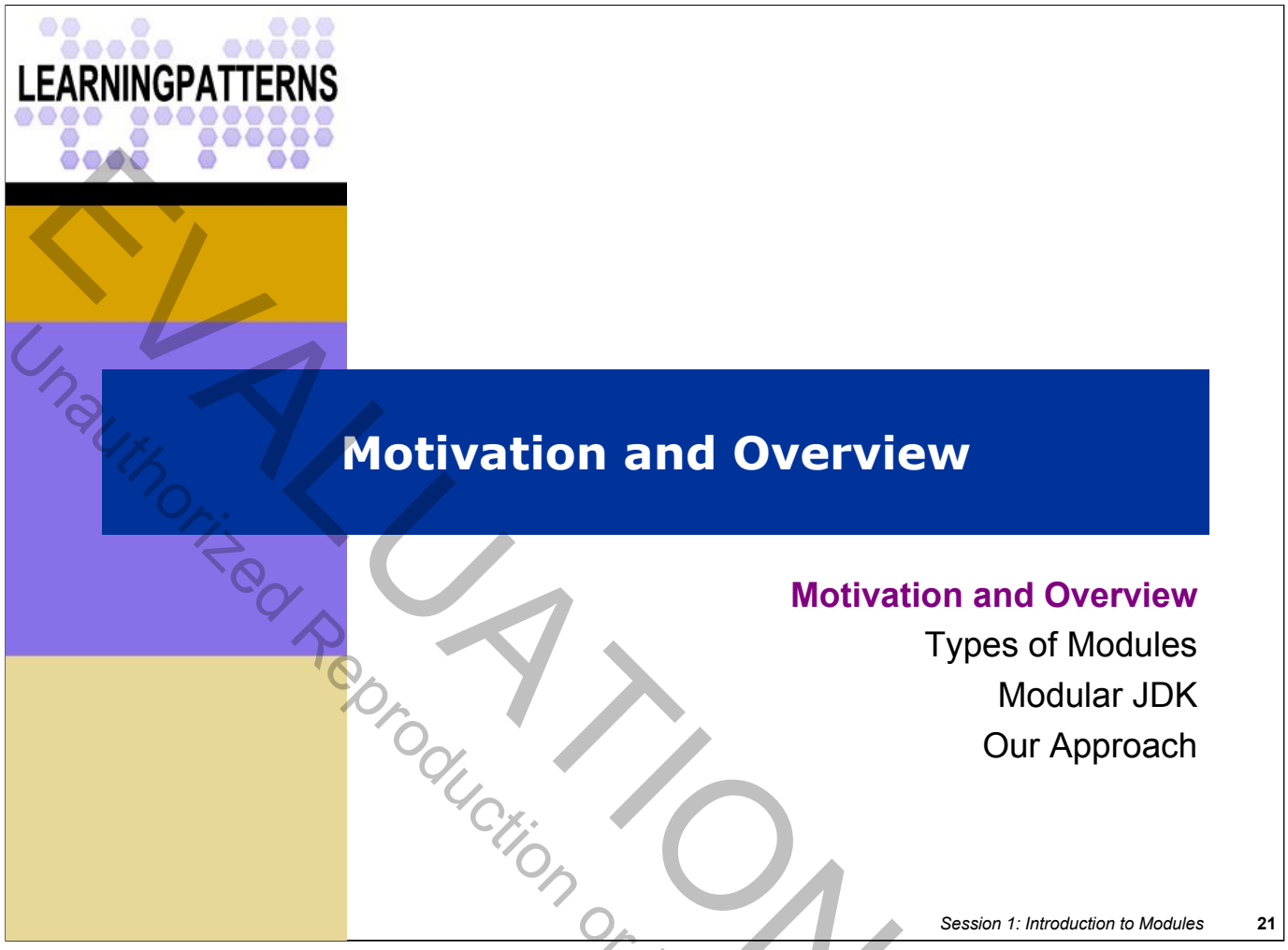


Notes:

Session Objectives

- Describe shortcomings and problems of the classpath
- Illustrate the need for modular encapsulation
- Introduce Java modules
  - Definition and structure, module descriptors, modular projects
  - How they reference each other, and the modulepath
- Describe the 4 types of modules
- Describe the modular JDK and understand how it supports both modular code and legacy (non-modular) code
- Outline our approach to learning modules in the course

Notes:



Notes:

Issues before Java Modules

- **No strong encapsulation**
  - Everyone can access a public class
  - Private access can be bypassed (reflection)
- **Unreliable configuration**
  - No way to list dependencies of a class
  - No versioning support
  - Classpath Hell aka JAR Hell
    - Can't tell in advance if classpath is correct...or complete
    - Runtime exception if not
- **JRE bloat** – large monolithic runtime needed for all apps
  - But large parts are often unused
- Java 9 introduced **modules** to address the shortcomings

Notes:

- Wanted: a class needs to be public (so your other packages see it), but not everyone sees it.
- Java does have explicit `import` statements, which do two things:
  - Let you use the unqualified name of a class.
  - Declare a dependency on the imported class.
- **However**, `imports` are a **compile-time** construct – at runtime, there's no telling which JAR(s) on the classpath contain the class in question, possibly leading to unforeseen problems...and Classpath Hell. The runtime classpath is a **flat list** – the first JAR that contains the class "wins."
- From *http://labs.consol.de/development/2017/02/13/getting-started-with-java9-modules.html*:
  - You probably enough know about unreliable configuration if you have ever experienced `NoClassDefFoundErrors` at runtime because of missing dependencies. Or spent hours and days trying to track down irreproducible bugs in your production environment, just to find out that somehow two versions of a 3rd party dependency have managed to sneak into your classpath. Most of those issues originate from the fact that applications written in Java 8 or below do not know about their own dependencies. Sure - `javac` won't compile our code if any compile-time dependencies are missing, and of course we have build tools and IDEs to support us with those. However at runtime, the JVM needs to lookup required classes in the classpath again, in some cases resulting in different classes being loaded by the classloader. When these things happen, they usually do during runtime. Ouch. And more often than not, tracking them down is a painful experience.

A Natural Progression of Organization

- Class definition collection of related data and functions
- Package collection of related classes
- **Module** **collection of related packages** but of course!

- Is this really needed? Look at any decent-sized library
- Spring
  - database-related packages: 33
  - web-related packages: 103
  - test-related packages: 30
- Java Platform API
  - UI packages: 33 + 15 (if you count sound and imaging)
  - XML packages: 35

Notes:

Package-Level Encapsulation

- Packages are created for organizational purposes
- Result is often **several subpackages per functional area . . .** or ... stated differently, **per *module***

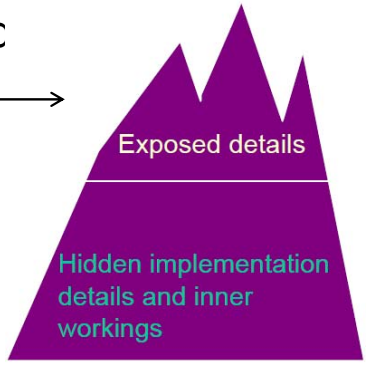
```
com.company.app.db
com.company.app.db.api
com.company.app.db.provider.jdbc
com.company.app.db.provider.jpa
com.company.app.db.provider.ext
```

- These packages need to **interface with each other** – so some classes are made **public**
  - Some developers will even write all their classes to be public . . .
- BUT every public class is now exposed to the **entire universe**

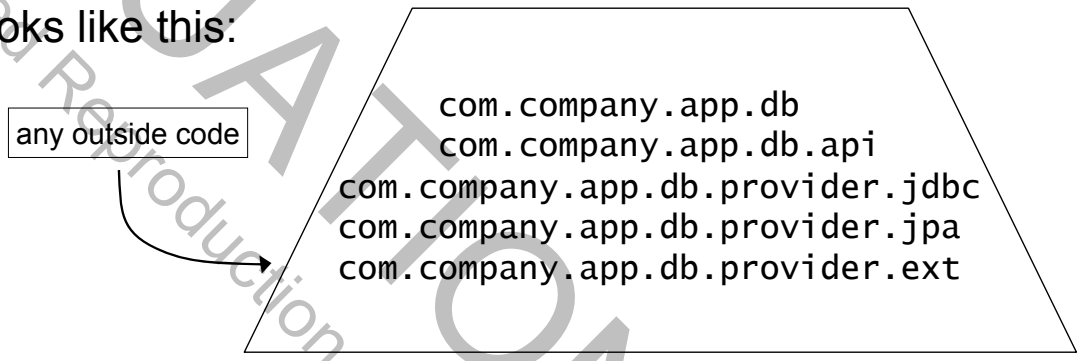
Notes:

Modular Encapsulation – Not in Java 8

- We all understand exposed vs. encapsulatec
  - A graphic from our Intro Java course



- In Java 8, there is no way to group a set of related packages
  - Exposing some
  - While hiding others
- Result looks like this:



Session 1: Introduction to Modules 25

Notes:

Internal Use Only – Case Study

- The following is direct from the Spring API documentation
- Package **org.springframework.objenesis**
  - Spring's repackaging of Objenesis 3.0 (with `SpringObjenesis` entry point; ***for internal use only***)
  - This repackaging technique avoids any potential conflicts with dependencies on different Objenesis versions at the application level or from third-party libraries and frameworks
- Why are they doing this?
  - "this technique avoids potential conflicts with dependencies"
  - In other words, they've been to JAR Hell too many times!!!

more on this later

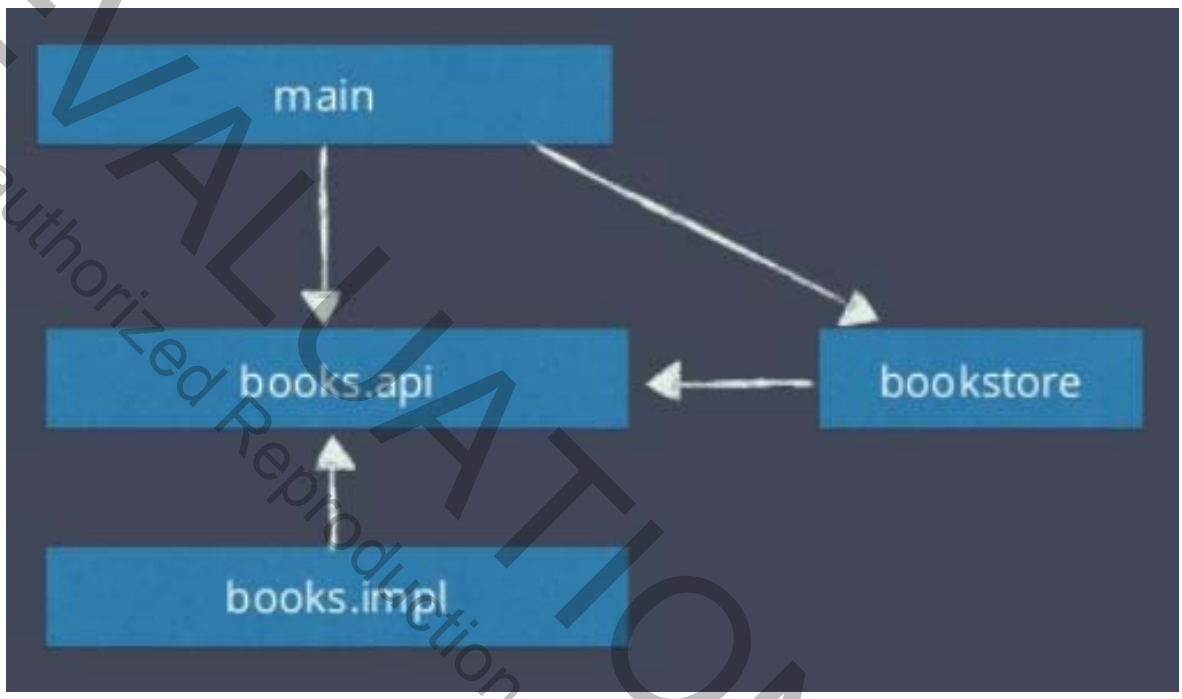
- **NOTE**: Java Modules does NOT address the "version problem"

Session 1: Introduction to Modules **26**

Notes:

Conceptual Model vs. Runtime Model

- Conceptual picture of your application
  - Easy to see how it all fits together . . .



Session 1: Introduction to Modules 27

Notes:

- Taken from *http://slideshare.net/SanderMak/migrating-to-java-9-modules*.

Conceptual Model vs. Runtime Model

- Actual view of your application

```
CLASSPATH=lib/antlr-2.7.7.jar:lib/cdi-api-1.1.jar:lib/classmat
e-1.3.0.jar:lib/commons-dbcp-1.4.jar:lib/commons-logging-1.2.j
ar:lib/commons-pool-1.5.4.jar:lib/dom4j-1.6.1.jar:lib/el-api-2
.2.jar:lib/geronimo-jta_1.1-1.1.1.jar:lib/hibernate-commons-ann
otations-5.0.1.Final.jar:lib/hibernate-core-5.2.2.Final.jar:li
b/hibernate-jpa-2.1-api-1.0.0.Final.jar:lib/hsqldb-2.3.4.jar:l
ib/jandex-2.0.0.Final.jar:lib/javassist-3.20.0-GA.jar:lib/java
x.inject-1.jar:lib/jboss-interceptors-api_1.1_spec-1.0.0.Beta1
.jar:lib/jboss-logging-3.3.0.Final.jar:lib/jcl-over-slf4j-1.7.
21.jar:lib/jsr250-api-1.0.jar:lib/log4j-api-2.6.2.jar:lib/log4
j-core-2.6.2.jar:lib/slf4j-api-1.7.21.jar:lib/slf4j-simple-1.7
.21.jar:lib/spring-aop-4.3.2.RELEASE.jar:lib/spring-beans-4.3.
2.RELEASE.jar:lib/spring-context-4.3.2.RELEASE.jar:lib/spring-
core-4.3.2.RELEASE.jar:lib/spring-expression-4.3.2.RELEASE.jar
:lib/spring-jdbc-4.3.2.RELEASE.jar:lib/spring-orm-4.3.2.RELEAS
E.jar:lib/spring-tx-4.3.2.RELEASE.jar
```

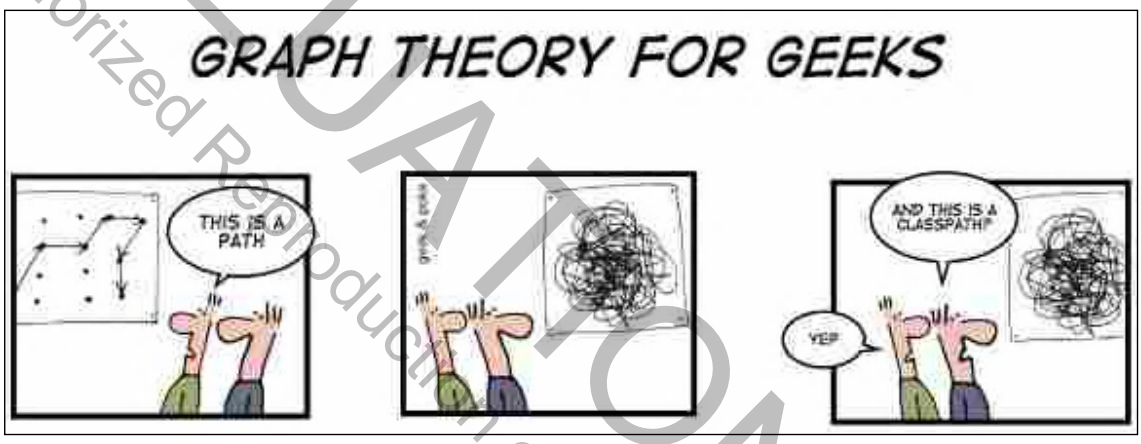

Notes:

- Taken from *http://slideshare.net/SanderMak/migrating-to-java-9-modules*.

- Take a close look at the JARs in this classpath – recognize any?
  - Some of these JARs **turn up everywhere**, and your application **will not run without them**, yet you're not using them from **your** code.
  - Must be that the libraries you call into are using them.
    - Now, without Maven or Gradle to help you, how would you know that?
  - Seriously, do you even know what `antlr` is? (No, it's not a Java API for hunting deer.)

Classpath Hell

- We've all been there at some point...



Notes:

Adopters of the Module Paradigm

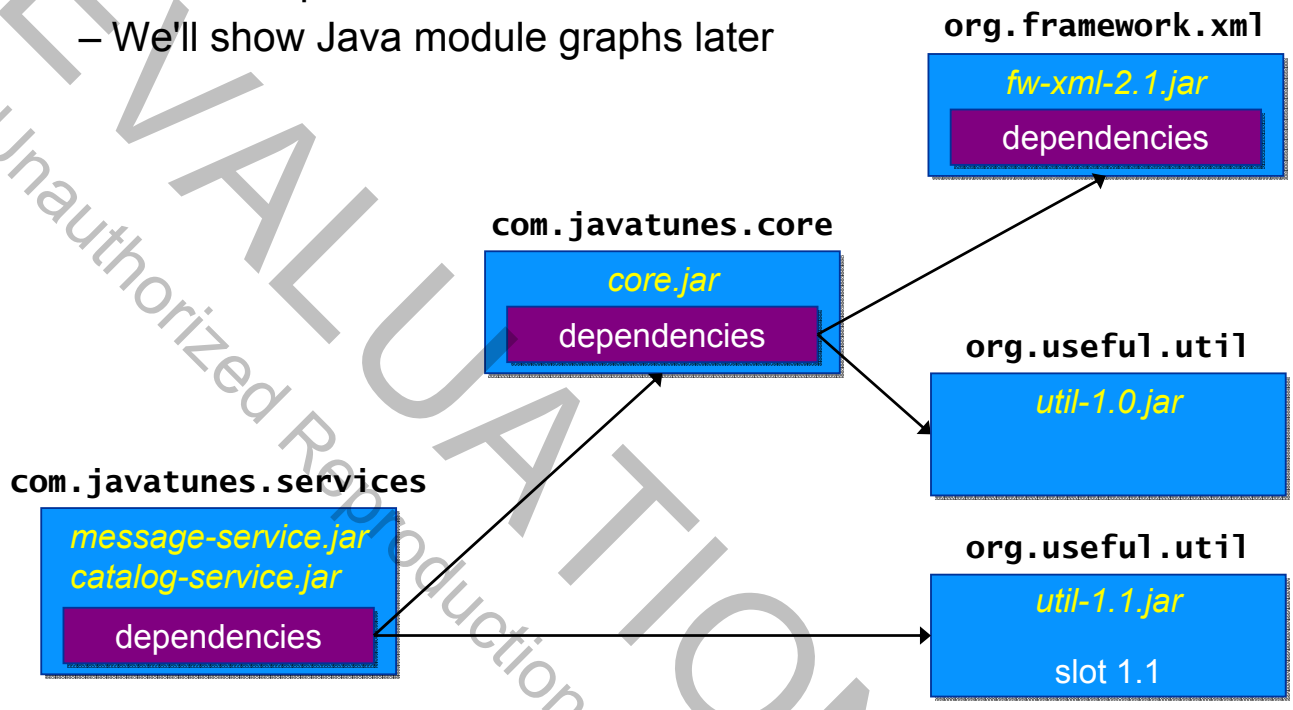
- **JBoss Application Server** has its own module system
  - **Different purpose than Java modules** – it's about packaging and deployment of applications and dependent libraries (JARs)
  - Game-changing architecture for Java EE app servers
- In JBoss, **all dependencies must be explicitly listed**
  - Including your applications' dependencies
    - Burden on the developer, but you get **exactly** what you want
    - Not just the first one found on the classpath at runtime
- This has several high-level similarities to Java modules

Notes:

- On any appserver, having multiple versions of the same library is not uncommon – different applications use different versions, and the server itself might use yet another version. There is also usually a tension between *WEB-INF/lib* and *<server-root>/lib*, i.e., which has precedence in the hierarchical classloader model, leading to yet another dismal place – **Classloader Hell**.
- JBoss Modules is a powerful deployment and classloading model.
  - It helps JBoss users solve dependency problems for applications. Effectively, there is no linear classpath (searched left-to-right, first-class-found-wins) – all classes are resolved via module lookup. Nothing is "left to chance."
  - It also drastically speeds up server start time and class initialization at runtime, because classes are found so quickly and efficiently – via module lookup instead of flat classpath.
- Disclaimer: this is not an endorsement of JBoss. We mention it only because it provides a useful example of how a module-based system can solve myriad problems that have been plaguing Java developers for years.

Module Graphs

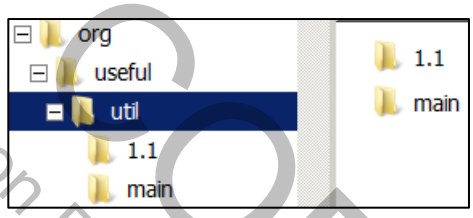
- Modules and their dependencies form a ***module graph***
  - This example is from JBoss Modules
  - We'll show Java module graphs later



Session 1: Introduction to Modules 31

Notes:

- In JBoss, a module is a named set of JARs.
  - This is defined in a module descriptor called *module.xml*, and the JARs are usually colocated with it. This module descriptor also lists the other module(s) that this module depends on.
  - These files are stored in a Maven-like repository structure, which **must** match the module name. In this screenshot, you can see the two version slots for `org.useful.util` – "`main`" and "`1.1`".



Introducing Java Modules

- **Module = named set of packages + *module descriptor***
  - ***module-info.java*** at root of its package hierarchy

```
module com.company.app.db {
  exports com.company.app.db;                    // exposed packages
  exports com.company.app.db.api;
  requires java.sql;                             // dependencies on
  requires java.persistence;                     // other modules
}
```

- **Exported packages** are visible to other modules
  - **IF** the other module **`requires`** this one

- This module ***reads*** **`java.sql`** + **`java.persistence`** (1)
  - **Read** a module = has access to its **exported** packages
  - Specifically, the **public classes** in those exported packages

Notes:

- Module = named set of packages.
  - Packages that are **exported** are visible to other modules.
    - Provided that those modules **read** this module.
  - All other packages in the module are **encapsulated**.
    - Hidden, and inaccessible to other modules.

- IMPORTANT: a module **exposes packages** and **depends on** (requires) other **modules**.

(1) This module **`requires`** modules `java.sql` and `java.persistence`
  - We say that module `com.company.app.db` **reads** modules `java.sql` and `java.persistence`.
  - The two modules are said to be read by this one

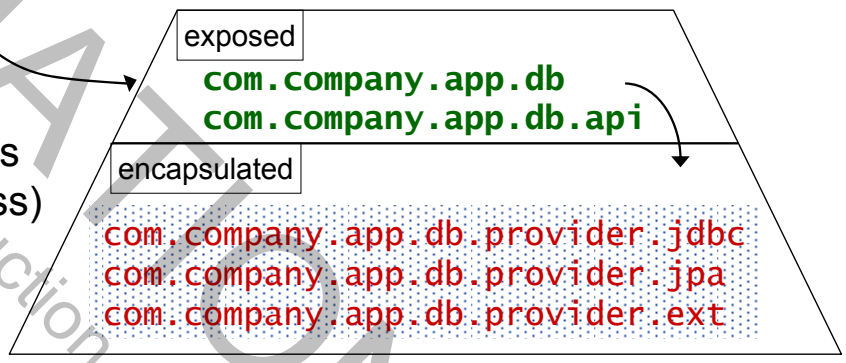
Modular Encapsulation – Realized in Java 9

- The 'client' module reads the 'db' module
  - Note which packages it has access to

```
module com.company.app.client {
  requires com.company.app.db;
}
```

```
module com.company.app.db {
  exports com.company.app.db;
  exports com.company.app.db.api;
  requires java.sql;
  requires java.persistence;
}
```

- Inside the module:
  - Normal visibility rules (class vs. public class)



Notes:

- Likewise, the 'db' module reads modules `java.sql` and `java.persistence`.
  - So it has access to the exported packages in those modules.

Module Name vs. Package Name

- This can get confusing – you'll just have to adapt
- Recall: **module = named set of packages**
  - Java SE platform modules are named `java.something`
  - Myriad packages are named `java.something`
- **Module `java.sql`** exports these **packages**:
  - `java.sql`
  - `javax.sql`

see the possible confusion?

- This can get confusing – you'll just have to adapt

deliberate reprint

- See notes for additional thoughts on module naming

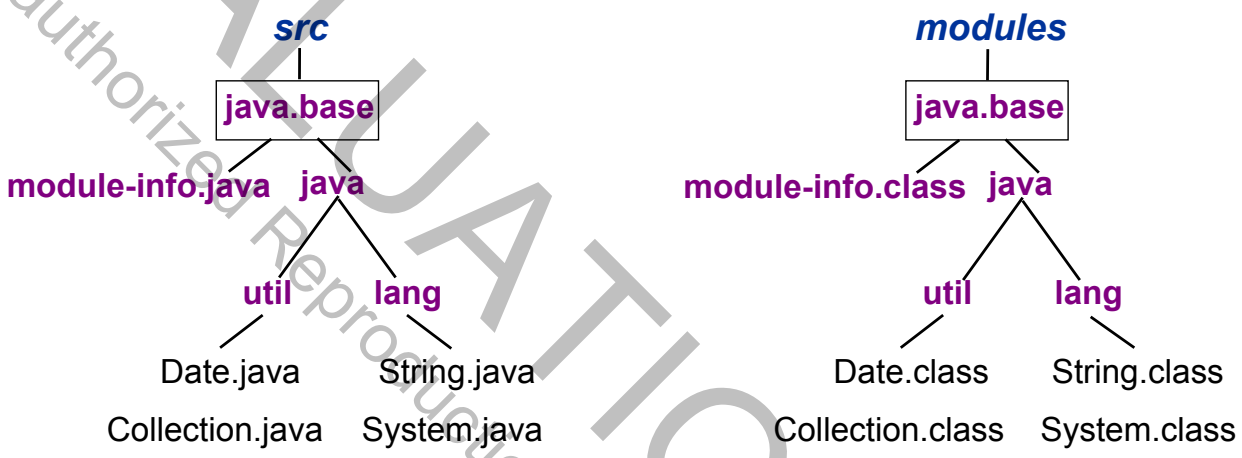
Session 1: Introduction to Modules 34

Notes:

- In many cases, the module name will match some part of its packages' names.
  - Arguably, the module name "`java.sql`" in the slide **is** a good name for these 2 packages.
- As another example, in our earlier demo module in the slides, **`com.company.app.db`** is certainly a suitable module name for these packages:
  - `com.company.app.db`
    `com.company.app.db.api`
    `com.company.app.db.provider.jdbc`
    `com.company.app.db.provider.jpa`
    `com.company.app.db.provider.ext`
- This is a common naming pattern – use the top-level package's name as the module name.
- You will also see module names that adhere to this basic pattern, but omit the "com" or "org" prefix, etc., resulting in something like `company.app.db`, or even just `app.db`.

Modules and Directory Structure

- **Convention: module base directory = name of module**
  - Module descriptor lives in module base directory (**required**)
  - Package directories live under the module base directory
  - Packaged into a standard JAR – a ***modular JAR***
  - Below are some packages from the `java.base` module



Session 1: Introduction to Modules 35

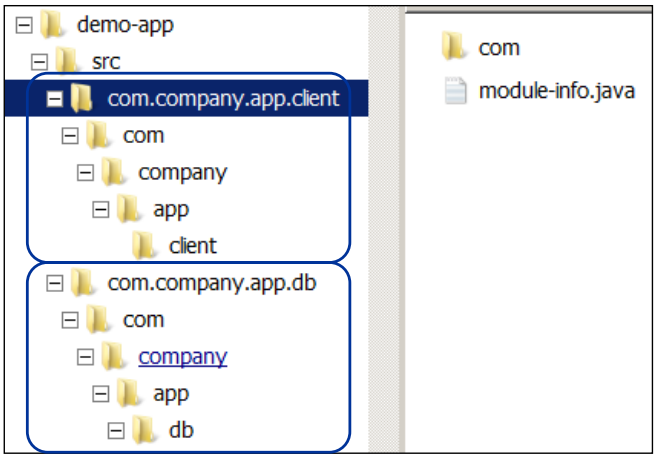
Notes:

- It is not strictly required that the module base directory be named after the module, but this is a **very common** convention, and we **strongly recommend** doing it this way.
  - There are some very good reasons for this convention, only one of which is clarity and organization. We'll see other benefits later in the course, in lab.

- There are two structures for file organization in modular projects – **single-module** and **multi-module**.
  - In the **single-module** structure, all the source code, including package directories and the *module-info.java* file, resides directly under the single base directory, e.g., ***src***.
    - In this model, you can only have **one module per source tree**, since you can only have one *module-info.java* file in a given base directory.
  - In the **multi-module** structure, you can have **multiple modules in one source tree**, each one rooted at its own base directory named after the module.
    - In this model, it is **required** that the module base directory = the module name.

Modular Projects and the Module Path

- Partial layout of demo app



- JDK tools have been updated to support modules
  - Commands below are run from **demo-app** directory
- See notes for more details

no spaces!

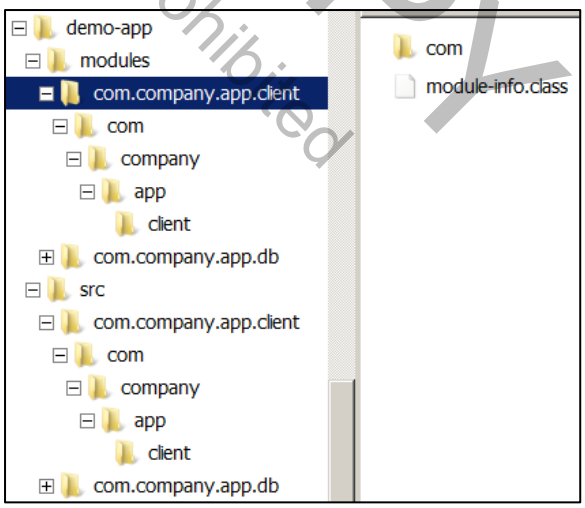
```
# compile both modules, each one treated as a single unit of code
$ javac -d modules --module-source-path src
        --module com.company.app.client,com.company.app.db
```

```
# module-path supports multiple entries using separator char for OS
# main-class is specified via --module module-name/main-class
$ java --module-path mlib;modules
       --module com.company.app.client/com.company.app.client.Main
```

Session 1: Introduction to Modules 36

Notes:

- Each module in the project is contained in its own module base directory, named after the module.
  - This is the **multi-module** structure. (See notes previous page.)
  - When using this layout, each module directory tree can be compiled as a "unit of code," and the compiler will create a mirror image of this structure, rooted under a **`modules`** directory (because we specified a '`-d modules`' argument which specifies the destination directory).
  - javac: **`module-source-path`** option specifies locations of source files for multiple modules
    - **`module`** option specifies the modules to compile
- When launching the application (`java`), the **`--module-path`** argument specifies location(s) of modules. Similar to classpath, with a few differences:
  - The locations must refer to modules, i.e., JARs with *module-info.class* at the root (modular JAR), or an exploded version of this.
  - It can also refer to a **containing directory** of modules, i.e., you can set `--module-path` to refer to a ***lib*** directory, and all modular JARs in that directory are automatically found. Nice!
    - This means that, unlike the classpath, a module path is typically not a mile long!
  - **`--module`** specifies main class in form module/mainclass



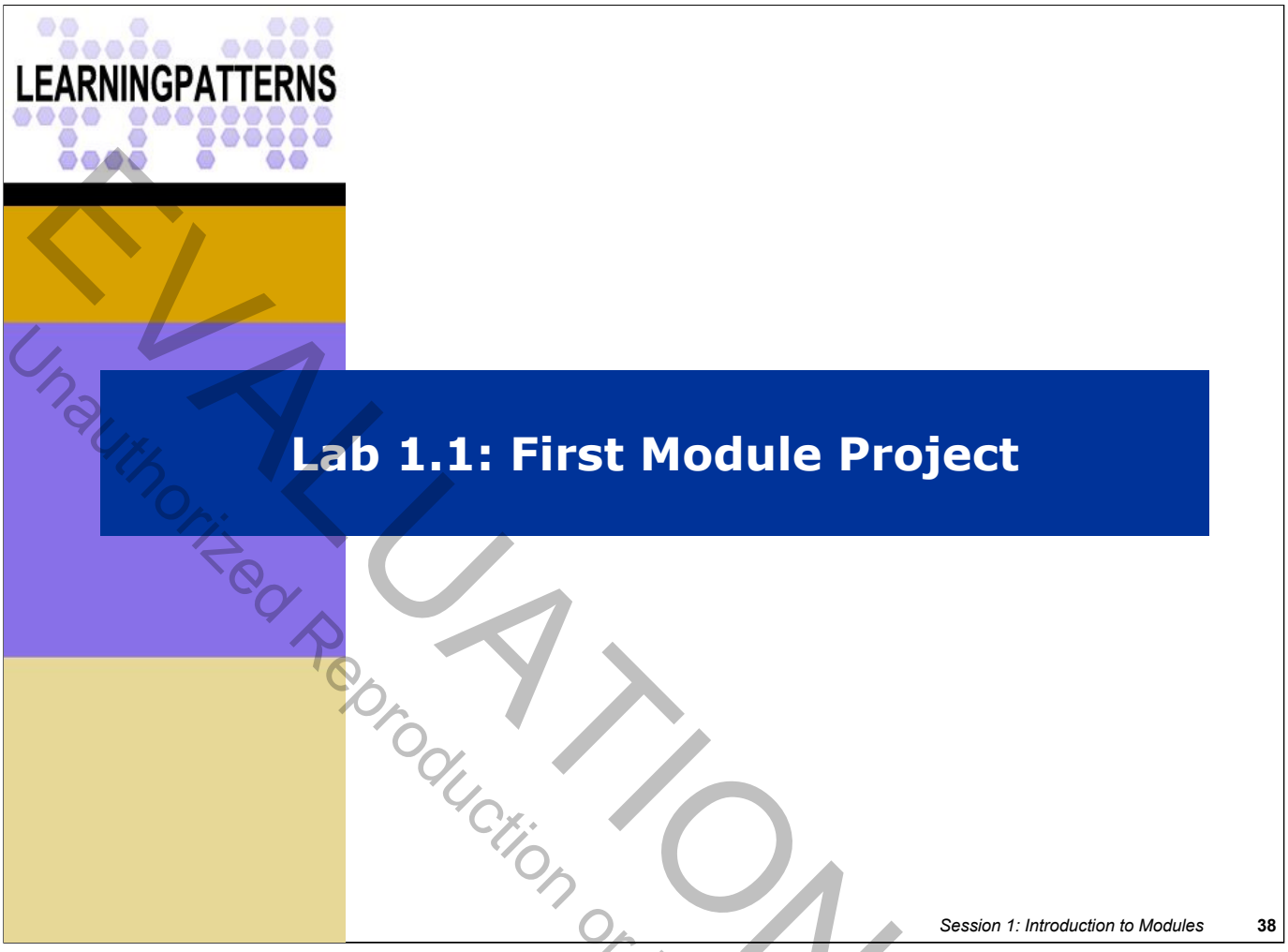
Modules Can Be Controversial

- Disagreements during specification phase
  - Remember the JCP, Oracle is not the boss...
  - May 2017: committee rejected module system in its current state
  - June 2017: approved in a 2nd vote (after changes were made)
- Modules **can** break existing code
  - While designed to be an opt-in technology, there are cases in which Java 8 code **will not run** out of the box on newer JVMs
- Is it better?
  - Most would agree that Classpath Hell / JAR Hell needs to go
  - Controversy remains wrt how much **net benefit** modularizing your code provides – are we heading toward Module Hell? 😐
  - More on this later

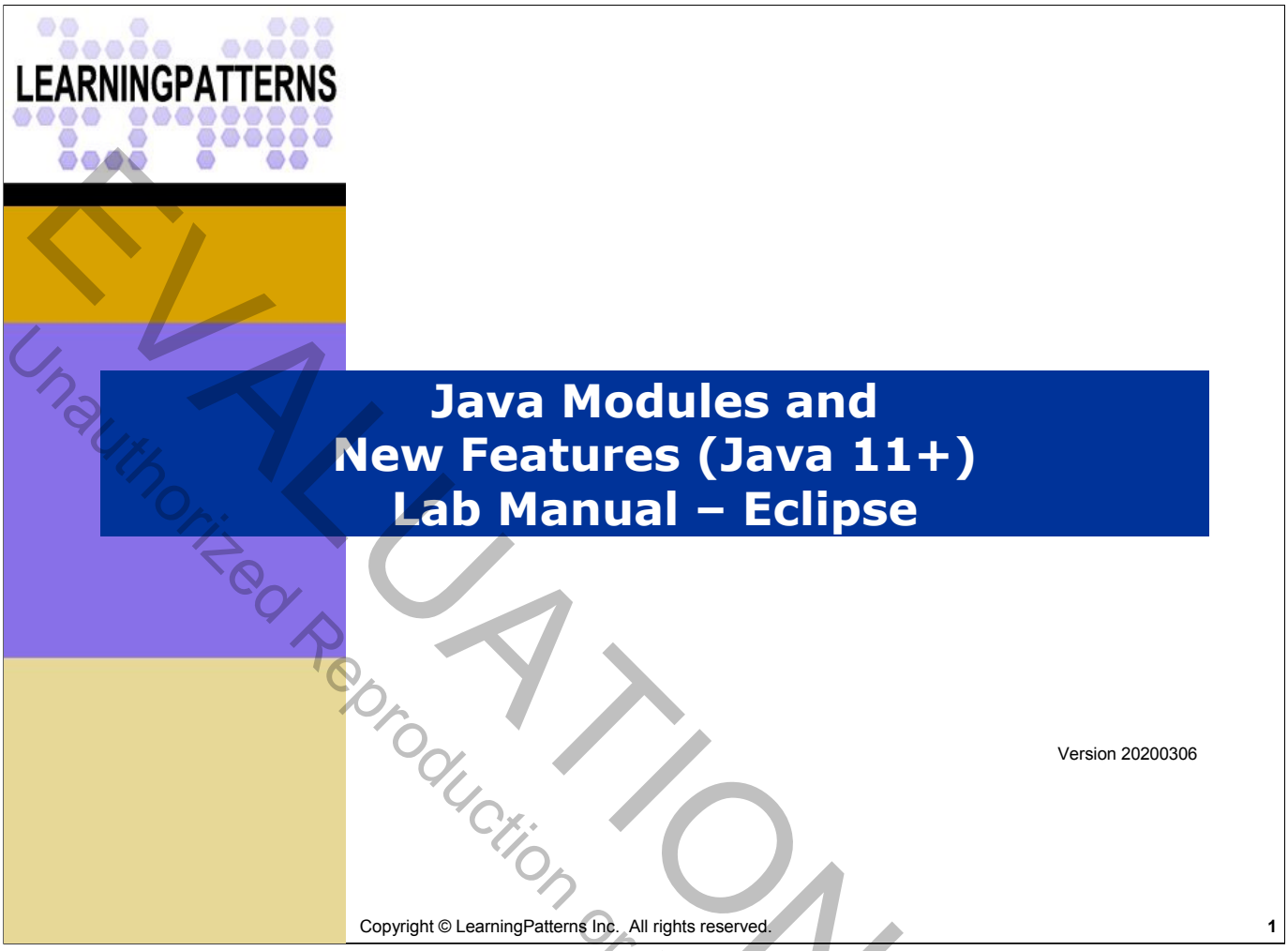
Session 1: Introduction to Modules 37

Notes:

- ***Project Jigsaw*** (JEP 261: Module System) took a long time to completely materialize into implementation. It was initially planned for Java 7, obviously that didn't happen. Same story for Java 8. Finally arrived in Java 9.
  - To be fair, it was **no small feat** to introduce such a **drastic change to Java**, while still **maintaining compatibility**, and providing a **means of migration** to the new module system.
    - And keep in mind how large the Java platform has grown.
    - You'll get a better sense of the sheer size of this effort once you start to modularize just a single application(!)
  - Add to this the committee nature of developing new Java specifications – these things can take awhile...
- The most likely future scenario is that, over time, this modular approach will become the norm. The migration issues will be over, your team will understand how it all works (there is certainly more complexity involved), and the tools and IDEs will have fully adapted.
  - Currently, however, some critics say that more work needs to be done, especially with regard to version selection and integration with build tools (both of which the module system in its current form does NOT provide).



Notes:



Notes:

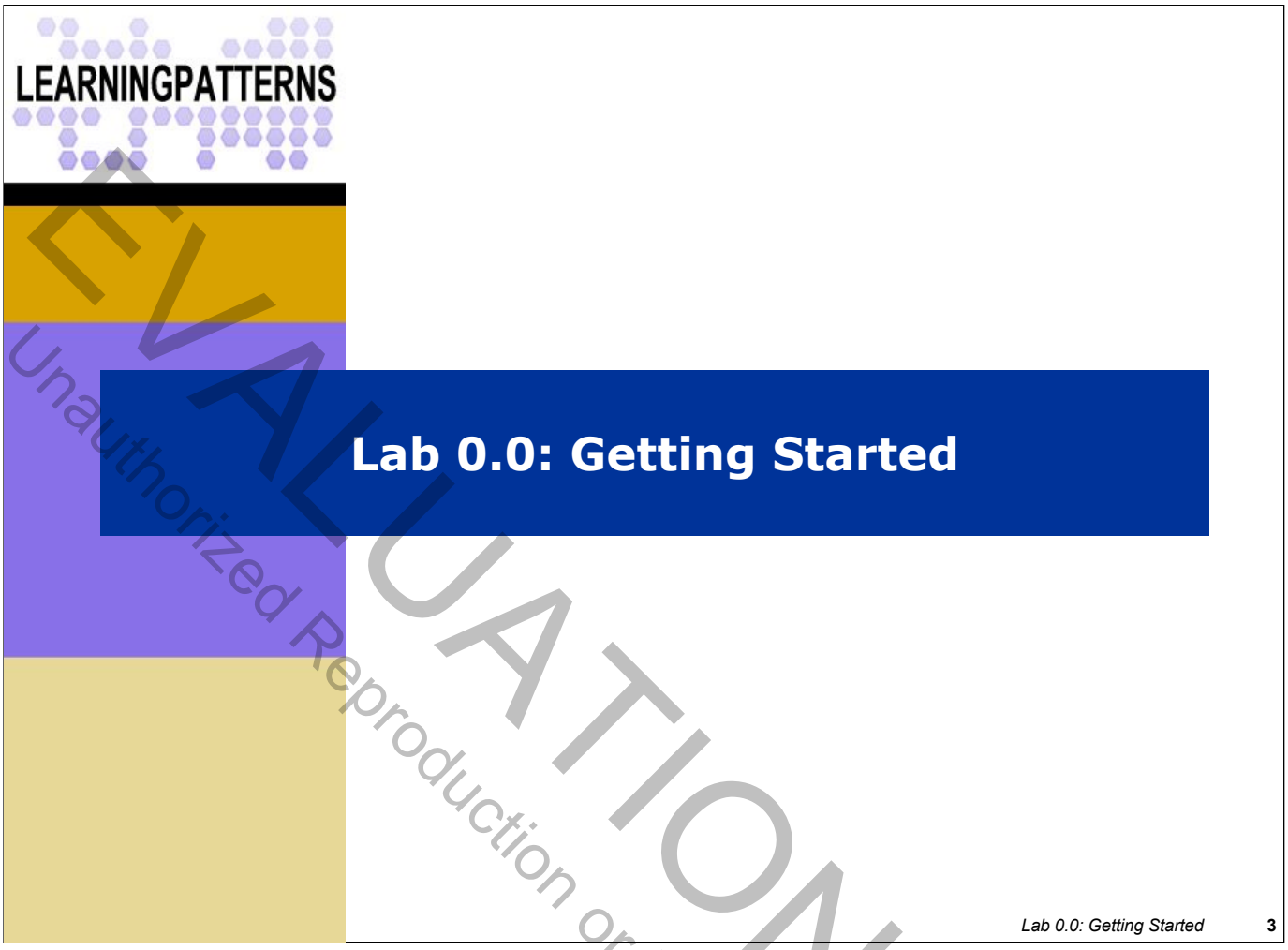
- Version 20200306

Release Level

- This manual contains instructions for creating and running the labs using the following software packages:
  - **Java SE 11 (LTS. Tested with 11.0.5)**
    - Later versions (e.g. Java 12) will **likely** work, but have not been tested
  - **Eclipse IDE for Enterprise Java Developers – 2019-06 or later**
- Instructions are geared towards Windows-based operating systems
  - You should easily be able to adapt to other environments, e.g., macOS

Introduction 2

Notes:



Notes:

Lab Synopsis

- **Overview**:
  - Examine new directory structure of JDK 11 installation
  - Extract and review the Java 11 API documentation (Javadoc)
  - Verify environment setup to do command line work with the JDK
    - JAVA_HOME and PATH
  - Work with some very simple Java 8 demo applications on Java 11
    - Do they work unchanged on Java 11?
- **Builds on previous labs**: none
- **Approximate time**: 20-30 minutes

Lab 0.0: Getting Started **4**

Notes:

Information Content and Task Content

Lab

- **Informational content** is presented in the normal way – the same as in the student manual
  - Like these bullets at the top of the page
- **Tasks** you need to do are in a "Tasks to Perform" box

Tasks to Perform

- Look at these instructions, and notice the different look of the box
  - Make a note of how it looks, as future labs will use this format
- Make sure that you have **Java 11 installed**
  - Likely in a directory such as ***C:\Program Files\Java\jdk-11.0.5*** (1)
  - If not, you'll need to install it – it can be downloaded from: ***http://oracle.com/technetwork/java/javase/downloads***
- OK – now **get out your setup files**, we're ready to start working

Lab 0.0: Getting Started 5

Notes:

(1) On macOS, the JDK is typically installed under */Library/Java/JavaVirtualMachines/jdk-11.0.5*.
  - On Linux, it may be in various places. Ask your instructor if you can't find it.

- The setup files may be on your workstation, or they may be provided by your instructor.

Extract the Lab Setup Zip File

- To set up the labs, you'll need the course setup zip file
  - It has a name like: ***LabSetup_Java11New_20200306.zip***
- Our base working directory for this course will be ***C:\StudentWork\Java11New***
  - Gets created when we extract the lab setup zip
  - Includes a directory structure and files needed in the labs
  - All instructions assume that this zip file is extracted to ***C:***

    If you choose a different directory, please adjust accordingly

Tasks to Perform

- Unzip the lab setup file to ***C:***
  - Creates the ***StudentWork/Java11New*** directory structure, containing files that you will need for doing the labs

Lab 0.0: Getting Started 6

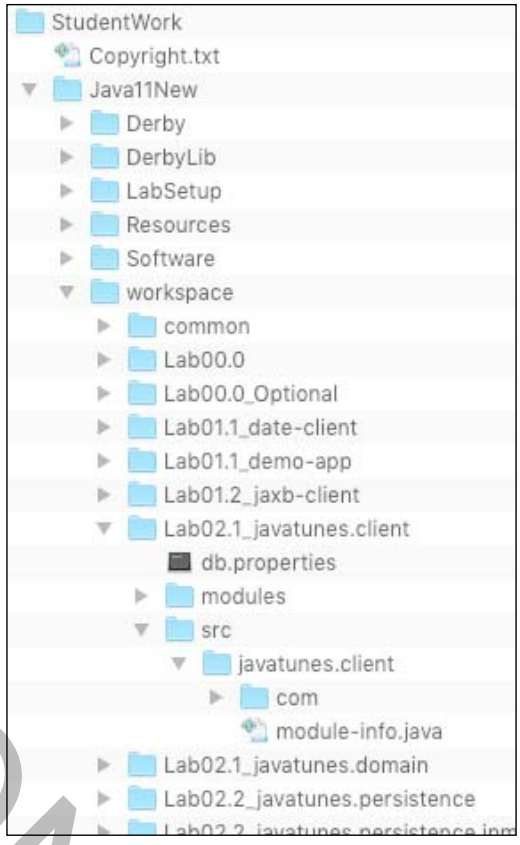
Notes:

- The setup files may be on your workstation, or they may be provided by your instructor.

Review Lab Folder Structure

Lab

- **StudentWork\Java11New** contains
  - **Derby/DerbyLib**: Database files
  - **LabSetup**: Additional Lab Files
  - **Resources**: Documentation
  - **Software**: Utilities for labs
  - **workspace**: Lab code (both Eclipse based and command line)
- **StudentWork\Java11New\workspace** will contain the following folders:
  - **LabNN** : Folder for Lab NN, containing:
    - **build files**: For non-Eclipse builds
    - **src/** or **src/module.name**: Java source
    - **bin** or **modules/modules.name** : compiled code



Lab 0.0: Getting Started 7

Notes:

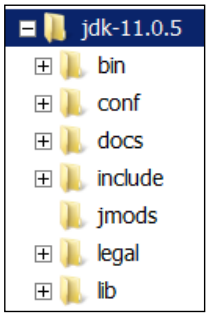
- Some labs are not modular.
  - Those labs will be laid out with source code under the src folder, and compiled code directly under bin (all in package folders, of course).
  - This is the standard Eclipse Java project layout.
- Some labs are modular.
  - Those labs will be laid out with source code under the *src/module.name* folder, and compiled code directly under *modules/module.name folder* (all in package folders, of course).
  - This is the Java standard layout for modular projects, which is not always required, but strongly recommended.
  - Eclipse does not always automatically set all of this up for you (unfortunately), but we will set it up as we create individual projects.

Review the JDK Installation

Lab

Tasks to Perform

- Using Windows Explorer, navigate to the JDK installation
  - On Windows, this is *C:\Program Files\Java\jdk-11.0.5* or similar
  - We refer to this as ***<jdk-install>***, ***<jdk-home>***, or ***<jdk>***



- Take a few minutes and look around
  - NOTE: you won't have a *docs* directory yet
  - Platform modules in ***jmods*** directory, as ***.jmod*** files
    - Used in building custom runtime images with `jlink`
  - The actual runtime image of the platform modules is ***lib/modules***
    - 100MB+ single binary file
    - No more *rt.jar* and the like, as in previous versions
- Open ***<jdk-install>/README.html*** and follow the link
  - From here, you can get to all the documentation, for several releases

Notes:

- On Mac, the Java install location is generally */Library/Java/JavaVirtualMachines/jdk-11.0.5.jdk/Contents/Home*
- On Linux, locations can vary, but some common ones are:
  - User home folder
  - /usr/lib
  - /usr/local
  - /opt
  - etc.
- To get the release notes, you can also go to java.com/licensereadme
  - From that page, follow the link to the **JDK 11 Readme**.

Extract the Platform API Docs

Lab

- We've provided the Java Core API docs (Javadocs) in the lab setup
  - In *StudentWork/Java11New/**Resources/jdk-11_doc-all.zip***
  - Nice to have it installed locally, no dependence on Internet connection
  - These docs are good for any Java 11 version, and for many later ones

Tasks to Perform

- Do an "extract to here" using a zip tool, which yields a ***docs*** folder
  - You can just browse them from here
  - Or, move that entire *docs* tree to *<jdk-install>*
    - Advise you **cut / move** the directory tree, NOT **copy** it (can be very slow)
  - Start at *index.html*
- Looks familiar, but the front page is **modules** now, not packages
  - NOTE the Search field upper right! ☺

Lab 0.0: Getting Started 9

Notes:

- On a Windows system, you may not be able to extract the zip directly into the Java installation directory, because it's under the special "Program Files" directory. As a workaround, you can extract to a temp directory, or do an "extract to here," then **move** (don't copy) the resulting ***docs*** directory to *<jdk-install>*.
- Though these are Java 11 docs, most of it will remain the same for later technology release versions – e.g. Java 12 or 13.

Verify Environment

Lab

- We'll need to use the JDK tools from the command line
  - Several new tools, and new options to existing tools
- Setting this up was requested to be done before the start of class
  - Be sure to take care of it now if needed

Tasks to Perform

- Set **JAVA_HOME** and modify **PATH**
  - JAVA_HOME environment variable should point to your *<jdk-install>*
  - PATH environment variable should include JAVA_HOME/bin
    - This is `%JAVA_HOME%\bin` on Windows, `$JAVA_HOME/bin` on macOS
- Open command prompt and test – see notes

```
$ java  -version
$ javac -version
```

Lab 0.0: Getting Started 10

Notes:

- These steps are SOP (standard operating procedure) for any Java developer, and should be familiar to you.

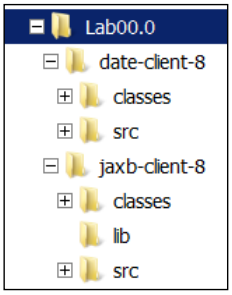
```
C:\StudentWork\Java11New>java -version
java version "11.0.5" 2019-10-15 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.5+10-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.5+10-LTS, mixed mode)

C:\StudentWork\Java11New>javac -version
javac 11.0.5
```

Compatibility with Existing Java 8 Code

Lab

- We've provided two very simple demo applications written in Java 8
  - *StudentWork/Java11New/**workspace/Lab00.0***, each in its own directory
    - ***date-client-8***
    - ***jaxb-client-8***



- They've been working great for years!
  - What happens when we move them to Java 11?

Tasks to Perform

- Open a command prompt to ***date-client-8***
  - Execute the appropriate **run** script for your OS – ***run.cmd*** or ***run.sh***
    - Should run ok – three dates, from the '`util`', '`sql`', and '`time`' packages
  - Review the `run` script in an editor – nothing special going on
- Try to compile/run it on JDK 11, using our **`build/run`** scripts (1)

Lab 0.0: Getting Started **11**

Notes:

(1) It should compile fine. Then review the build script – again, nothing fancy.
- Try running it again via the run script. Should run as before. These are JDK 11-compiled classes now.
- Looks like this one works fine on Java 11!

Compatibility with Existing Java 8 Code

Lab

Tasks to Perform

- Now we'll examine the **jaxb-client-8** application
  - In your command prompt, change directory to ***jaxb-client-8***
  - Again, execute the appropriate **`run`** script for your OS
  - Looks like we've got a problem!

```
Exception in thread "main" java.lang.NoClassDefFoundError:
javax/xml/bind/JAXBContext
```

  - Now run it via **`run-11`** script – works this time
    - You'll see a "book marshalled" message
    - The marshalled books are in the *books* subdirectory, in XML files

- Review and compare the two scripts: `run` and `run-11`
  - The `run-11` script adds a JAXB implementation (and its dependencies) to the classpath – this is needed, because **Java 11 removed JAXB**

- Now run and review both build scripts: **`build`** and **`build-11`** (1)

Lab 0.0: Getting Started 12

Notes:

(1) The `build` script fails and the `build-11` script succeeds, for the same reason: `build-11` adds JAXB to the classpath.

Observations

Lab

- Modules are completely "opt-in"
  - You don't have to change your Java 8 code to be "modular" to run it
- There are several caveats though, and we've seen one here
  - **date-client** runs fine on Java 11, with no changes necessary
    - Can also recompile it with JDK 11 no problem
  - **jaxb-client** runs fine on Java 11, BUT we had to add JAXB back in
    - JAXB and other overlapping packages from Java EE have been removed
  - All in all, not too big a problem – just a few JARs added
- **NOTE**: this is all because it's a **classpath-based application**
  - Our application code is **not in a module**
  - Our code and the 3rd party libraries (JAXB) are on the **classpath**
  - We get to use everything defined in platform module **`java.se`** – this module represents the "entire" Java SE platform [see notes]

Lab 0.0: Getting Started 13

Notes:

- With caveats – some of which are things like JAXB being removed, some of which involve deep reflection into platform classes. We will examine this in more detail in the next section.

Additional Research [Optional]

Lab

- We've provided several useful ReadMe files and accompanying filesets in the lab bundle, in an organized folder structure
  - *StudentWork/Java11New/**workspace/Lab00.0_Optional***

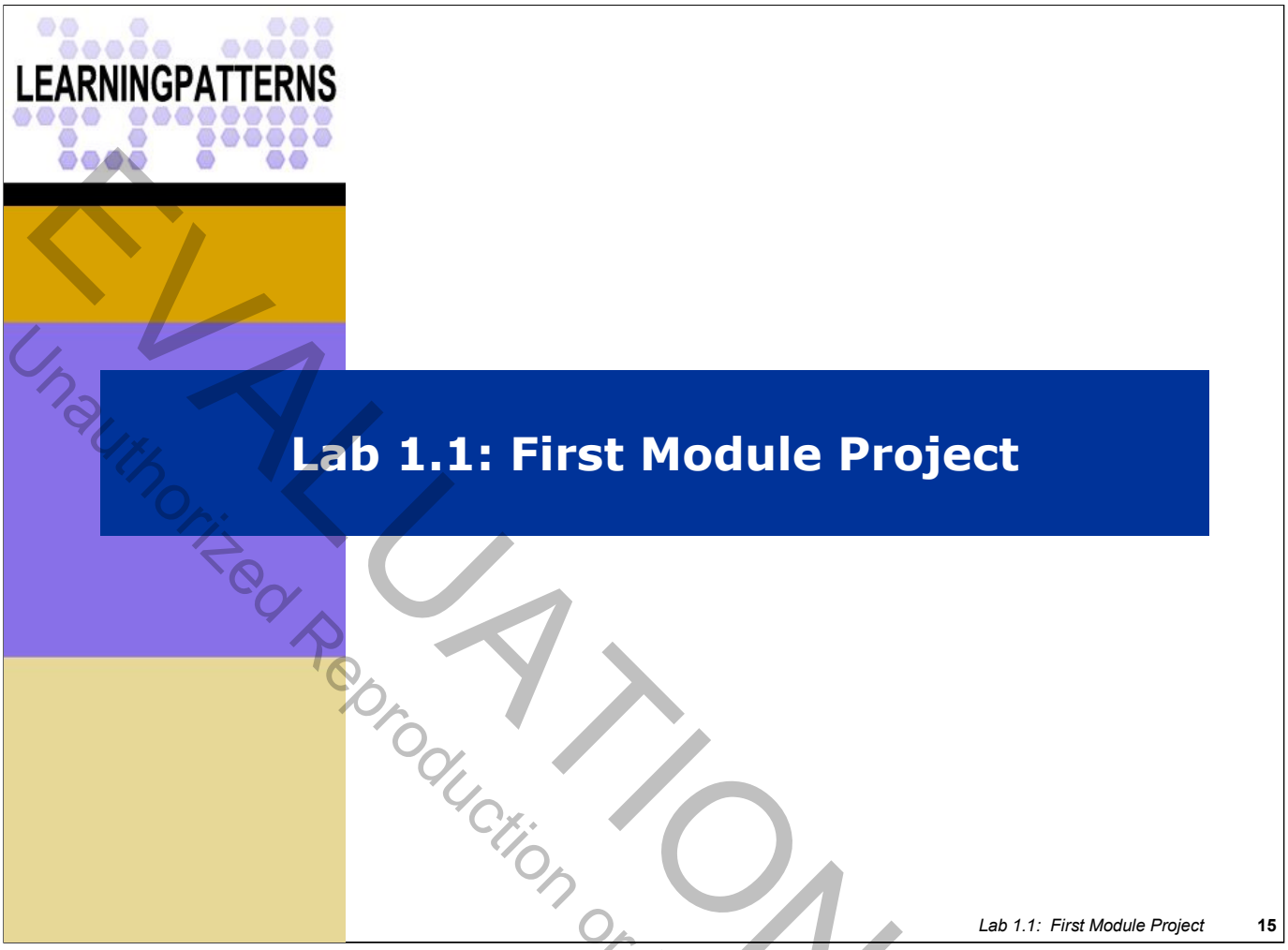
Tasks to Perform

- Outside of class time (lunch, evenings, etc.), we recommend you take a look
  - An important one is the ***java-support-updates-roadmaps*** folder
    - We strongly suggest you look at this one, even if you don't bother with any of the others – start at the usual place: *ReadMe.txt*

STOP

Lab 0.0: Getting Started 14

Notes:



Notes:

Lab Synopsis

- **Overview – this lab has two parts**:
  - **Part 1**: build and inspect a multi-module project, via command scripts
  - **Part 2**: import a Java 8 application into the IDE and make it modular
    - NOT a full-scale migration exercise, but illustrates some of the differences between "old" and "new" – Java 8 code vs. Java 9 code, and classpath vs. modulepath
- **Builds on previous labs**: none
- **Approximate time**: 25-35 minutes

Lab 1.1: First Module Project **16**

Notes:

Part 1: Multi-Module Project

Lab

- We've provided a "full version" of the demo app from the slides
  - The **com.company.app** demo used to illustrate the basics of modules
  - It's just a few source files, but it uses the customary project layout, and allows us to get familiar with modular projects

Tasks to Perform

- On the **filesystem**, use Windows Explorer and navigate to *StudentWork/Java11New/**workspace/Lab01.1_demo-app***
  - Expand the ***src*** folder and note the two module base directories
    - `com.company.app.client`
    - `com.company.app.db`
  - Each contains *module-info.java* and a standard package structure
    - Drill down into each one and open a source file or two for **brief** review
    - Standard `package` and `import` statements, very simple Java code
- Next, we'll work with some build and run scripts ⟶

Lab 1.1: First Module Project 17

Notes:

JDK Tools with Modular Project

Lab

Tasks to Perform

- Open a command prompt to the project base directory, ***Lab01.1_demo-app***
  - Open the **`build`** script (*.cmd* or *.sh*) for review – note how it's compiling **modules**, not sets of individual source files
  - Open the **`build-jar`** script for review – nothing new here (1)
- Execute the scripts: first `build`, and then `build-jar`
  - See instructor if you get errors
    - Likely a simple JAVA_HOME and/or PATH issue with your setup
  - Examine the filesystem
    - A ***modules*** directory structure of *.class* files that mirrors that of ***src***
    - An ***mlib*** directory with modular JARs (2)
- Open the **`run`** script, review it, then run the application (3)

Lab 1.1: First Module Project **18**

Notes:

(1) Reminder: since some of the new options to the JDK tools offer only the "--new-form", we will be favoring this one.
- For example, `jar` options `--create` and `--file` instead of the traditional `-cf`.
- Note also that we will break our own rule – in particular, we tend to favor `-classpath` over the newer `--class-path`. Old habits die hard...
- Of course, all the old flags are still supported.

(2) There is nothing special about our choice in using an "*mlib*" directory instead of something like *lib*. We're just emphasizing that these are modular JARs.

(3) The application gets executed twice – first with a modulepath of exploded modules, and then with modular JARs. This means it won't run unless you've run **both** `build` and `build-jar` beforehand.
- In both cases, the modulepath is set to the modules' **containing directory**, i.e., the directory above the individual modules.

Module Descriptor and Resulting JARs

Lab

Tasks to Perform

- Open both module descriptors using any text editor (not Notepad)
  - That is, each module's ***module-info.java*** file
  - 'client' module requires 'db' module and exports no packages
  - 'db' module exports 2 packages out of its 5 packages total
- Inspect the modular JARs
  - Open them with any zip utility, notice **module-info.class** at the root
  - Then, use the **jar** utility to inspect them () (1)

```
$ jar --file mlib/app-client.jar --describe-module
$ jar -f mlib/app-client.jar -d # shorter, same meaning
```

  - Notice the following in the output:
    - An "exports" or "requires" for each directive in the descriptor (obviously)
    - Each one "requires java.base mandated" – **this is important**
    - Encapsulated packages (not exported) are listed as "contains"

Lab 1.1: First Module Project 19

Notes:

(1) Remember! Forward slash (/) on unix, backslash (\) on Windows.
We generally show forward slash (/) in the slides.

We will revisit this technique in more detail later, which is especially useful when dealing with library or other JARs that were not created by you. This is something you'll encounter when doing a migration.

Modular Encapsulation – Demonstrated

Lab

- Let's see strong encapsulation in action
  - As structured, all the classes in this project need to be public *
    - Not a flawed design, but one of the consequences of Java 8 visibility rules
    - The classes in the '`provider`' packages are visible to the '`client`' package
    - This changes once we go modular

Tasks to Perform

- Open **`com.company.app.client.Main`** in an editor (not Notepad)
  - Look for the TODO and perform the task
  - Read the compiler error – does it make sense?
- Comment back out the offending line of code and rebuild the project
- You can close the command prompt
  - We'll be working in the IDE for this second part

Lab 1.1: First Module Project 20

Notes:

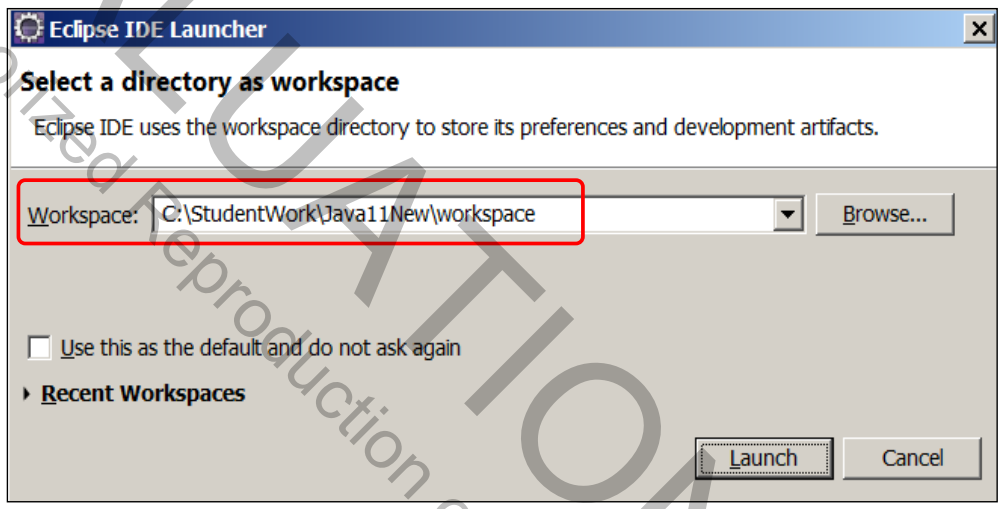
* Why would this be so?

Part 2: Module Project in the IDE

Lab

Tasks to Perform

- **Launch Eclipse**: go to ***C:\eclipse*** and run ***eclipse.exe***
  - A dialog box should appear prompting for a workspace location (1)
  - Set workspace location to *C:\StudentWork\Java11New**workspace*** (2)
  - If a different default workspace location is shown, change it
  - **Don't import any projects** – just open the workspace (3)



Lab 1.1: First Module Project 21

Notes:

(1) If the workspace dialog box does **not** appear, that means that Eclipse was previously launched on your system, a workspace directory was chosen, and the "Use this as the default..." box was checked.

- In this case, you should change workspace to that shown above via:
  File → Switch Workspace → Other... This will open the dialog box.

(2) **NOTE: you must use our provided workspace directory**, as we've supplied starter code and other files for some of the labs here.

(3) You should not do anything but open Eclipse using the correct workspace here.

- In particular, you should not import any project folders you see. We'll explain how to use these all correctly later, and it's important to wait until then.

- If Eclipse was installed elsewhere, adjust the path to the Eclipse executable accordingly.

- You can also put a shortcut on your desktop to start Eclipse.

The Eclipse Platform

- **Eclipse** (**www.eclipse.org**) is an open source platform for building integrated development environments (IDEs)
  - Used mainly for Java development – can be extended via plugins and used in other areas (e.g., C# programming)
- Originally developed by IBM
  - Released into open source
  - IBM's RAD product line is built on top of Eclipse
- Eclipse products have two fundamental layers:
  - **Workspace**: files, packages, projects, resource connections, configuration properties – all stored on the **filesystem**
  - **Workbench**: editors, views, and perspectives – the **UI** you work with
- We will set up the workspace and workbench, then do our lab

Lab 1.1: First Module Project **22**

Notes:

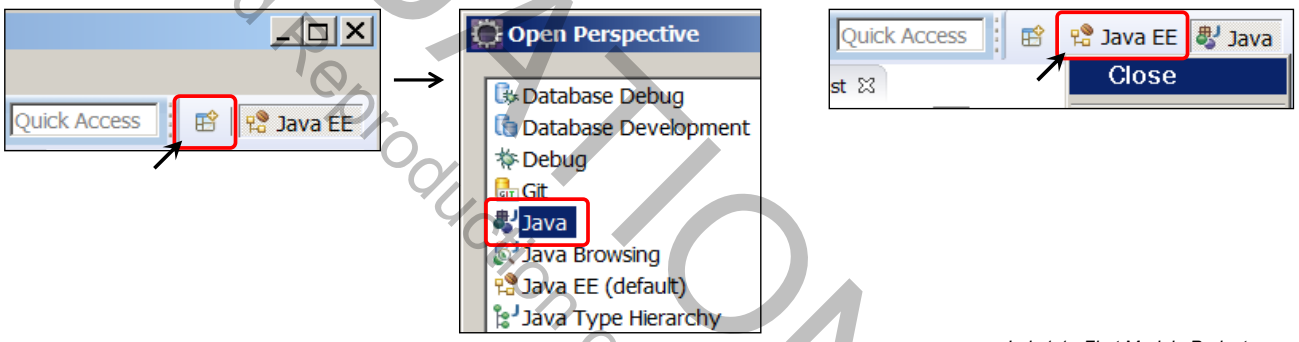
- The workbench sits on top of the workspace and provides visual artifacts (the UI) that allow you to access and manipulate the underlying workspace resources.

Workbench and Java Perspective

Lab

Tasks to Perform

- Close the Welcome screen and **open a Java Perspective**
  - Click the Perspective icon at the top right of the Workbench and select **Java** (see below left and below center)
  - See notes about perspective icons and theme (look and feel)
- **Close the Java EE perspective** by right-clicking its icon at the top right of the Workbench and selecting Close (see below right)
  - The Java EE perspective is the default for the Eclipse Java EE version



Lab 1.1: First Module Project 23

Notes:

- **NOTE**: the perspective switcher does not show the perspective name by default. To change this, right-click on any perspective icon and **Show Text** – see examples below.

without labels

with labels

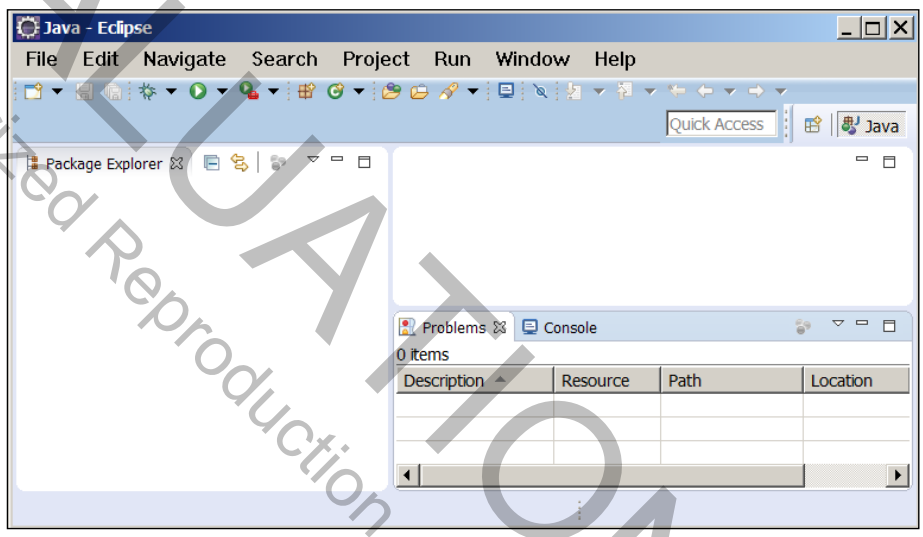
- Many of our screenshots use the "classic" theme, colors, and fonts.
  - This may not be what your version uses, which is OK. It shouldn't cause any confusion.

Customize the Perspective

Lab

Tasks to Perform

- Unclutter the perspective by closing some views
  - Close the Task List and Outline views
- Open the Console view: **Window → Show View → Console**
- You can save these changes to the perspective (see notes)



Lab 1.1: First Module Project 24

Notes:

- An Eclipse ***perspective*** is simply a collection of ***views***, shown as tabbed panes in the Workbench.
  - A perspective defines which views are included, where they are positioned, and what their relative sizes are in the Workbench.

- Eclipse has several predefined perspectives, which can be used as-is, or customized to suit your needs / preferences. There's a good chance you'll want to tweak the ones you're using.
  - To customize a perspective, open a predefined one, e.g., the Java perspective, close the views you don't want, open any views you do want, reposition and/or resize them, and then save the changes via **Window → Perspective → Save Perspective As...**
    - TIP: to preserve the factory-shipped perspective and create a brand new perspective based on it, use a different name, e.g., "Java [YI]", where "[YI]" are Your Initials in square brackets.
    - Window → Perspective → Save Perspective As... → Java [YI].

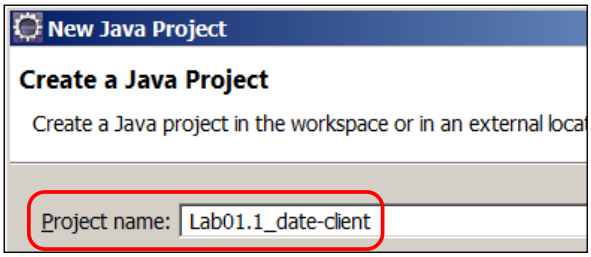
- Your instructor may also recommend additional customizations, which s/he will go over with you.
  - There are myriad settings, all available via **Window → Preferences**.

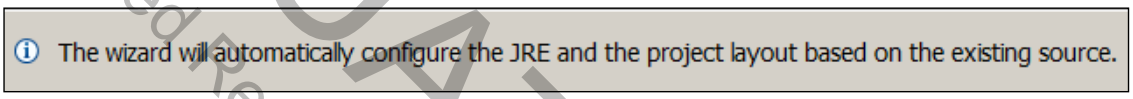
Create Module Project (1 of 2)

Lab

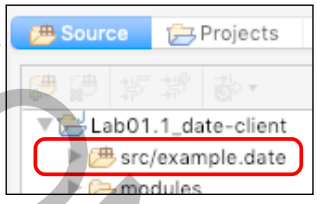
Tasks to Perform

- Check the Eclipse preferences to be sure you are using Java 11
  - Window → Preferences → Java → Installed JREs
- Create a new **Java** project
  - **File → New → Java Project**
  - Name it **exactly Lab01.1_date-client**
    - Becomes project root folder name (1)
    - Picks up any starter code - noted in Eclipse info message below





- Click **Next**, and in the **Java Settings** dialog, note the ***src/example.date*** source folder



Continue on next slide →

Lab 1.1: First Module Project 25

Notes:

(1) **IMPORTANT NOTE**: you **must** name the project **Lab01.1_date-client**, in order to pick up the starter code we've provided for this lab.

- That's because the starter code was preprovisioned in the *workspace/Lab01.1_date-client* directory (in appropriate subdirectories).
- By naming the project *Lab01.1_date-client*, the project directory defaults to *workspace/Lab01.1_date-client*, and the starter code is automatically recognized and included in the project's fileset.

- NOTE: we are deviating from Eclipse's default directory structure for source and output files for a standard Java project, which is normally *src* and *bin*, respectively.
  - We are instead using the common convention of ***src/module.name*** for source files, and ***modules/module.name*** for compiled *.class* files.
  - We have source in ***src/example.date***, and that **IS** picked up automatically by Eclipse as a source folder, replacing the default src/ folder.
  - Sadly though, it still defaults to bin for compile .class files.
  - We change that in the extra steps we're taking here. Later, we'll show how Eclipse can automatically set the correct output folder based on the existence of .java and .class files in the correct places.

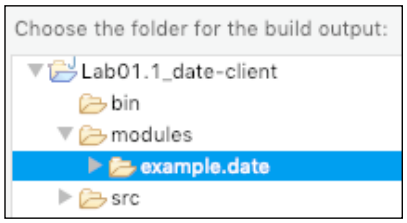
Create Module Project (2 of 2)

Lab

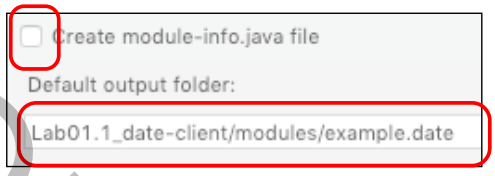
Tasks to Perform

- In lower left of Java Settings, **Uncheck "Create module-info.java"**
- Set the **Default output folder** by clicking the browse button

  - Browse to ***modules/example.date***
    - This already exists in our startup folders
    - If it didn't, you could just type it in, and Eclipse would create it
    - Click **OK**



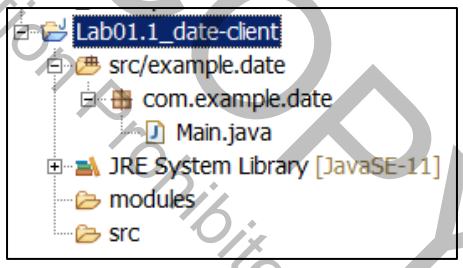
- You're set – Click **Finish**
  - See notes for screen shot of resulting project



Lab 1.1: First Module Project 26

Notes:

- Once done, it looks pretty much like a regular Eclipse project.
  - We've just changed the default source and output folders to follow module conventions.



Going Modular – Before and After

Lab

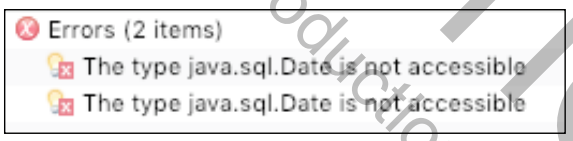
Tasks to Perform

- We can run our project as a Java 8 classpath-based application
  - Right-click ***Main.java*** → **Run As → Java Application**
  - Works as before (in Lab 0.0, when we ran it from the command line)

- Now we'll write a very simple module descriptor for our project
  - Right-click on source folder *src/example.date* → **New → File**
  - Name the file ***module-info.java*** and write the descriptor – see notes:

```
module example.date {
  // nothing to export or require
}
```

  - Save the file (which compiles it), and now we have errors in the project!



Lab 1.1: First Module Project 27

Notes:

- There is an Eclipse wizard for this task, which adds more intelligence to this operation, but we **don't** want that at this time – there's a reason...

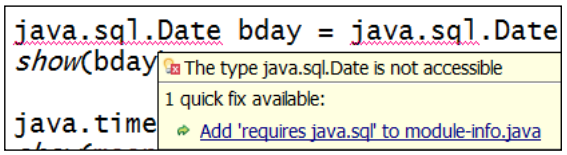
Going Modular – Before and After

Lab

- What's happening here and why?
  - **Every module implicitly reads platform module `java.base`**
  - **Before our code was put in a module, it implicitly read `java.se`**
    - This module gives legacy Java 8 code access to the entire Java SE platform
    - Minus the things that were removed in JDK 11, e.g., JAXB
    - But now we don't get that for free!

Tasks to Perform

- See Javadoc for each of these platform modules:
  - `java.base`, `java.se`, and `java.sql`
- You'll need to add a **`requires`** directive for the **`java.sql`** module
  - Before you do, examine the errors in *Main.java* and note the quick fix



- Re-run the application → should work now

see notes

STOP

Lab 1.1: First Module Project 28

Notes:

- Shouldn't we also do this for the other Lab 0.0 project, jaxb-client-8?
  - We could, but it's better to wait until we know more about the different types of modules (next).

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com