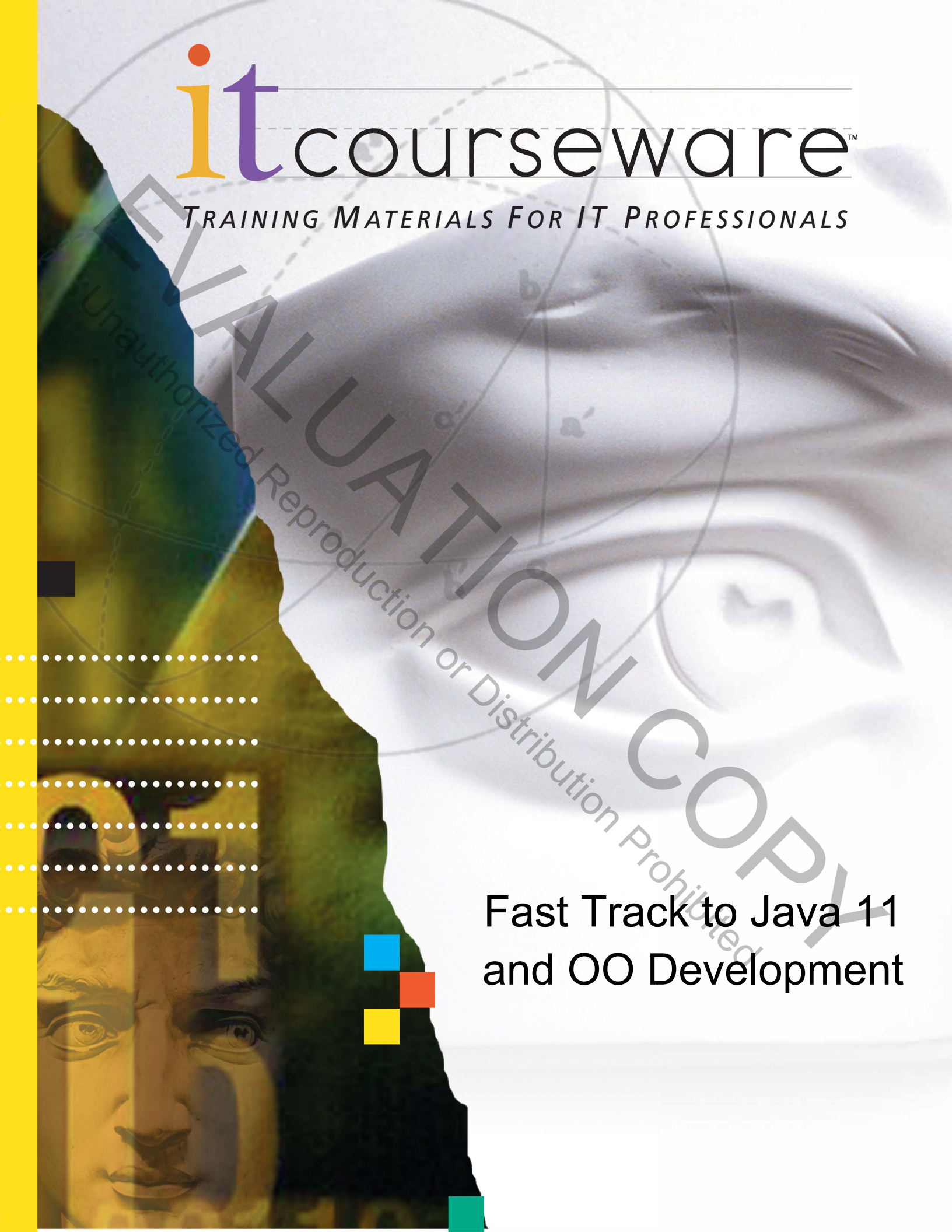


itcourseware™

TRAINING MATERIALS FOR IT PROFESSIONALS

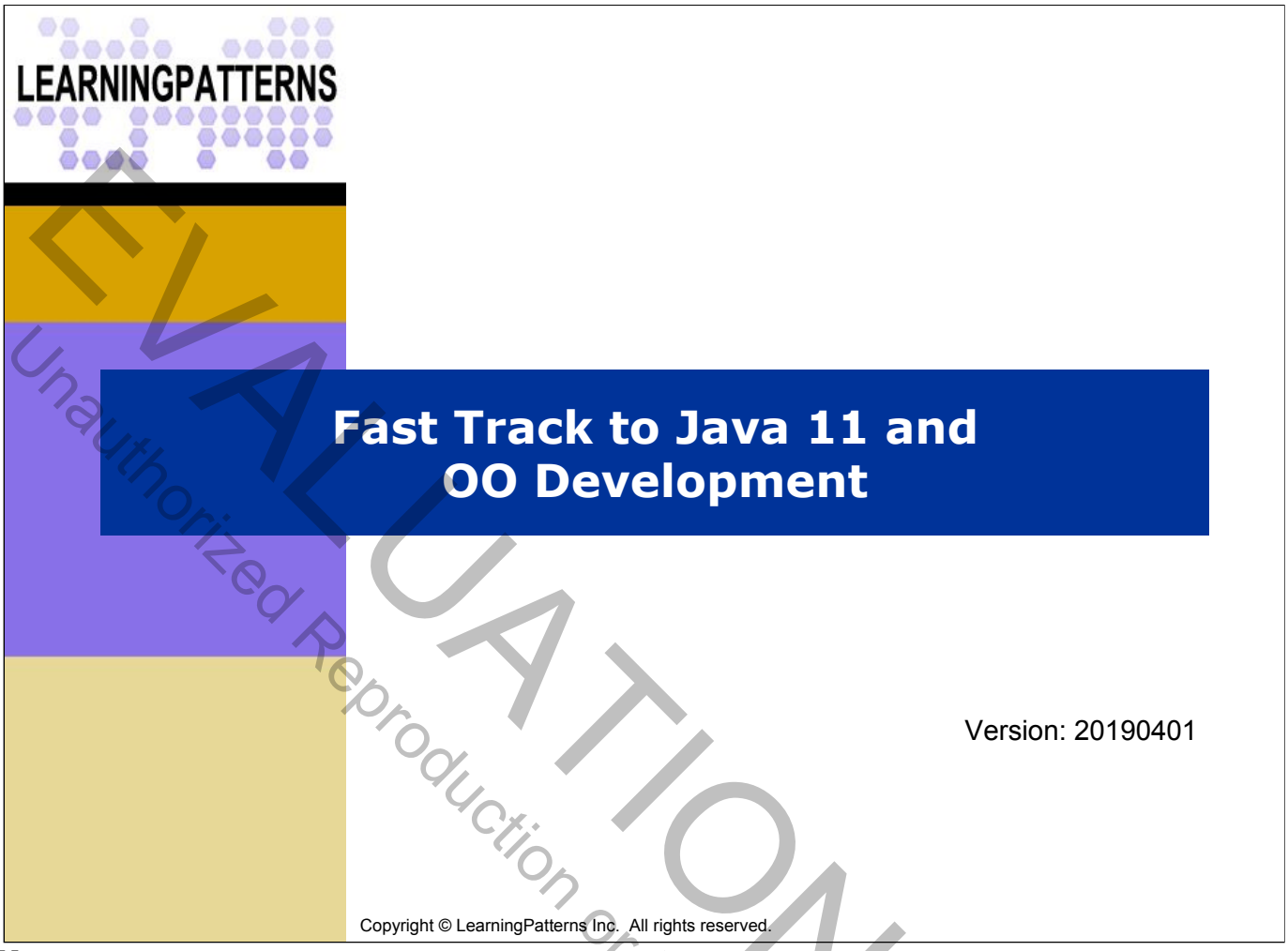
Fast Track to Java 11 and OO Development

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: Fast Track to Java 11 and OO Development



Notes:

- Version: 20190401

Course Overview

- Introductory Java course starting with basic principles providing:
  - A solid foundation in object-oriented concepts and good practices
  - Coverage of all core Java technology
  - Skills needed to write good Java-based systems
  - Coverage of database access with JDBC/JPA
- Be prepared to work hard and learn a great deal!
- There are numerous hands-on labs
  - Exercising all the important concepts discussed
  - Lab solutions are provided
- The course supports all recent versions of Java

Notes:

What We'll Learn About and Use

- Java's architecture, uses, and language basics
  - Compile and execute Java programs
  - Naming conventions, good coding style, good structure
  - Define and use classes and use the core Java API
- Object-oriented (OO) programming and the object model
  - Including composition, inheritance, and polymorphism
- Modules and packages to organize code
- More advanced capabilities - interfaces, exceptions
- Other APIs:
  - Database access with JDBC/JPA
  - Collections Framework
  - I/O

Notes:

Labs

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
  - The detailed lab instructions **are separate** from the main lecture manual
- Setup zip files are provided with skeleton code for the labs
  - Students add code focused on the topic they're working with
  - There is a solution zip with completed lab code

Notes:

Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:

```
JavaInstructor teacher = new JavaInstructor()
```

  - Code fragments use the same font, e.g. `teacher.teach()`
  - We **bold/color** text for emphasis
  - Filenames are in italics, e.g. *JavaInstructor.java*
  - Notes are indicated with a superscript number (1) or a **star ***
  - Longer code examples appear in a separate code box (below)
    - Code fragments may leave out detail (e.g. imports, or class def)

```
package com.javatunes.teach;
public class JavaInstructor implements Teacher {
   public void teach() {
      System.out.println("Java is way cool");
   }
}
```


Notes:

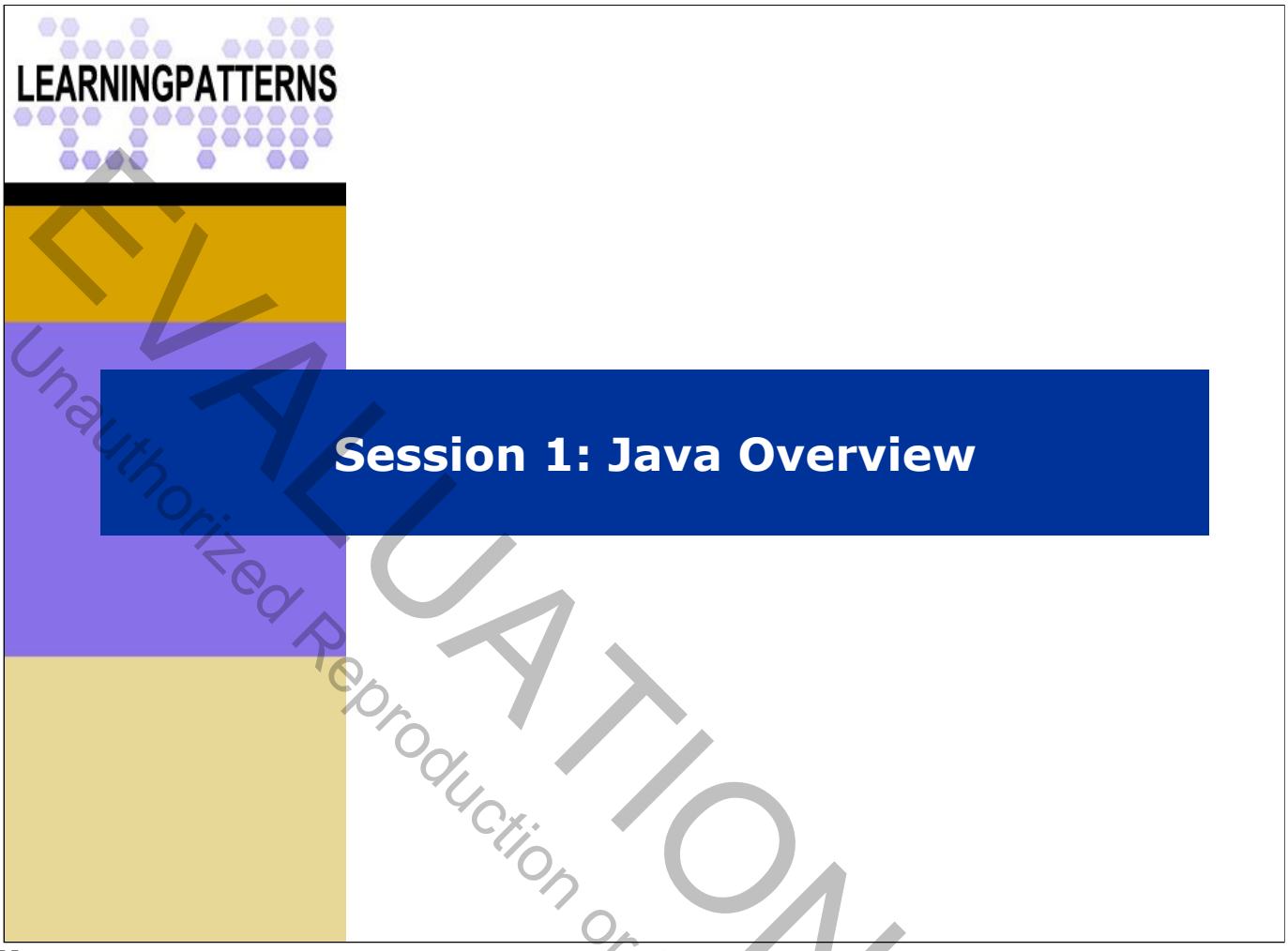
(1) If we had additional information about a particular item in the slide, it would appear here in the notes, with the appropriate superscript.

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

Session List

- Session 1: **Java Overview**
- Session 2: **Class and Object Basics**
- Session 3: **Classes and Objects in Detail**
- Session 4: **Flow of Control**
- Session 5: **More on Classes and Objects**
- Session 6: **Strings and Arrays**
- Session 7: **Packages and Modules**
- Session 8: **Composition and Inheritance**
- Session 9: **Interfaces**
- Session 10: **Exceptions**
- Session 11: **Collections and Generics**
- Session 12: **Database Access with JDBC and JPA**
- Session 13: **Additional Language Features**
- Session 14: **Java I/O** (optional)
- Appendix: **JDBC**

Notes:

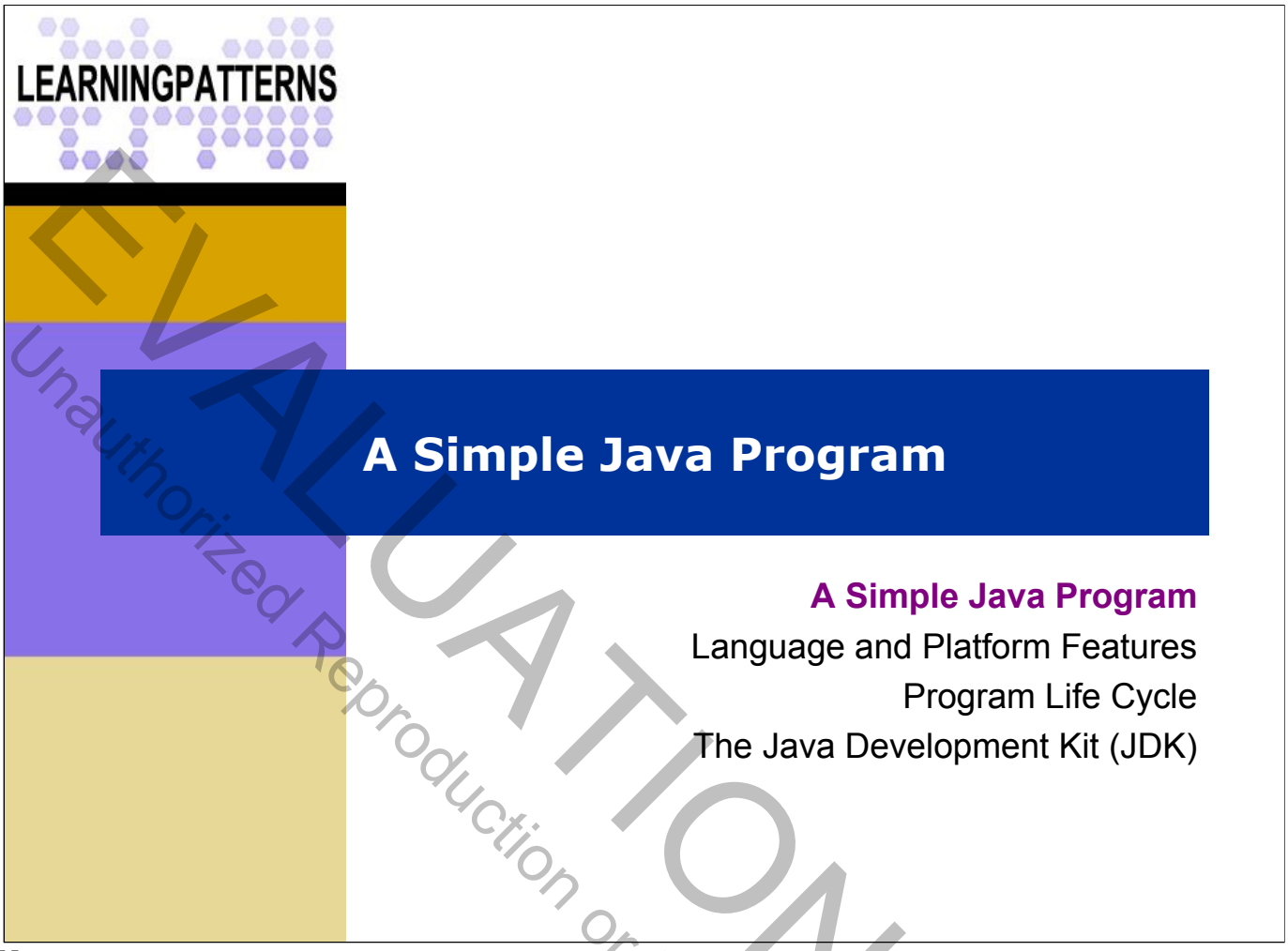


Notes:

Session Objectives

- Look at a Java program, and how it is compiled and run
  - And create and run it in the lab environment
- Understand how Java is a language and a platform
  - Know how it supports multiple environments
  - Be aware of the different Java platforms
- Define portability and explain how Java achieves this
- Know the Java program development and runtime lifecycle
- Understand the JDK (Java Development Kit) and be aware of some of its tools

Notes:



Notes:

A Simple Application Class

- We'll first review a simple Java app - `HelloWorld`
  - It displays "Hello World"
  - It's a non-graphic standalone application
- Java programs are a **`class`** containing a **`main()`** method
  - The `HelloWorld` source file must be named ***HelloWorld.java***
- **`main()`** is the starting point of an app
  - Must be declared as below, and must be defined within a class
  - `System.out.println` outputs to the console

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World");
   }
}
```

Session 1: Java Overview 10

Notes:

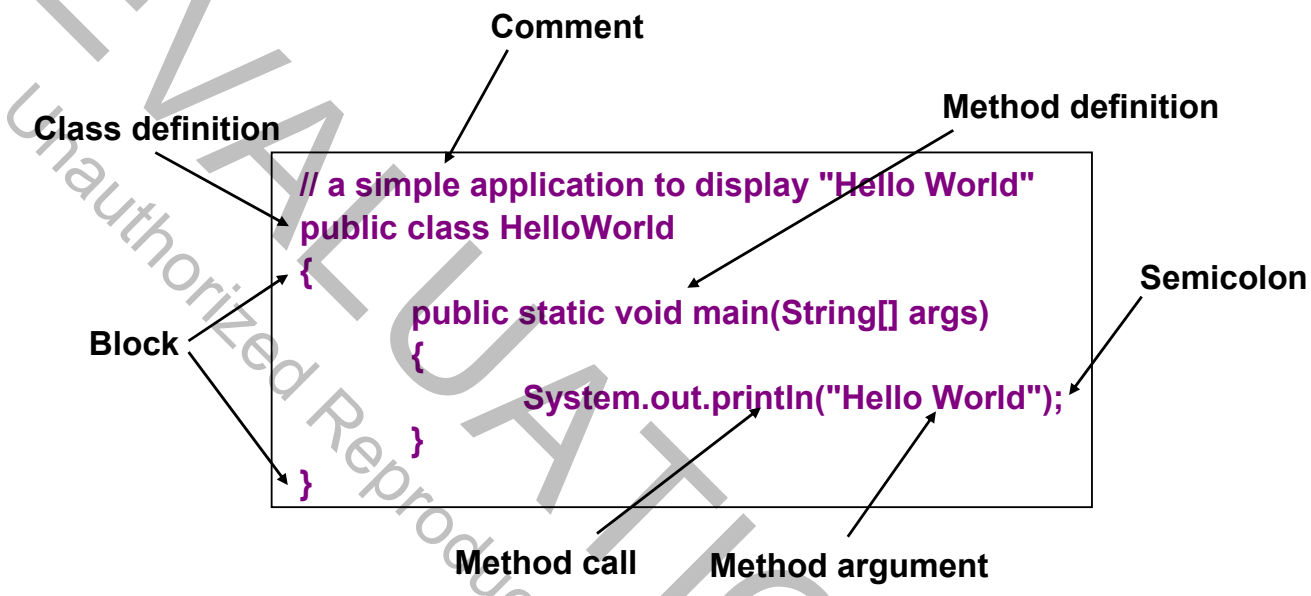
- Java classes can be deployed into several runtime execution environments, including the following.
  - Standalone applications that are run explicitly, like our `HelloWorld` application.
  - Servlets and other enterprise components which are run in a Java application server.
  - Applets, which are downloaded from a Web server and run in a Web browser.
    - Applets are no longer used, nor supported.
- We will concentrate on applications in this course.

- `main` is the special method which is the starting point for every standalone Java application.
  - Other components (e.g. servlets) work a little differently.
- The `main` method has to appear inside a class.
  - `main` takes a single argument – an array of `String`s.
  - This array holds arguments that are given on the command line when the application is run.

- `System.out.println` causes something to be printed to `System.out`, which is the standard output (the console) – more on this later.

HelloWorld in More Detail

- We'll cover these details in more depth later



Notes:

Compiling HelloWorld

- To use a Java class, **compile** the Java code with **`javac`**
  - `javac` is the Java compiler

```
C:\StudentWork>javac HelloWorld.java
```

  - Compiling creates ***HelloWorld.class***
    - Containing Java **bytecode** (more on this soon)
- Run the program with the Java Virtual Machine (**JVM**)
  - Via the **`java`** executable

```
C:\StudentWork>java HelloWorld
```

  - Producing "Hello World" in your console

Session 1: Java Overview 12

Notes:

- *HelloWorld.class* is the bytecode for the program produced by the compiler.
  - We'll talk more about bytecode soon.
- For now, we're giving you just enough information to just run the program.

Note on Comments

- Java has 3 kinds of comments
  - Single line comment: Starts with // ...
  - Multi-line using /* ... */
  - Javadoc comments using /** ... */ (1)

```
/**
 * This class prints "Hello World" to standard output
 */
class HelloWorld  {   // this comment starts mid-line
  public static void main(String[] args) {
    System.out.println("Hello World");
    // Easy to comment out code this way
    // System.out.println("Bye");
  }
}
/* This class still needs work.  We need to add many
   interesting things, and show how cool Java is.  */
```

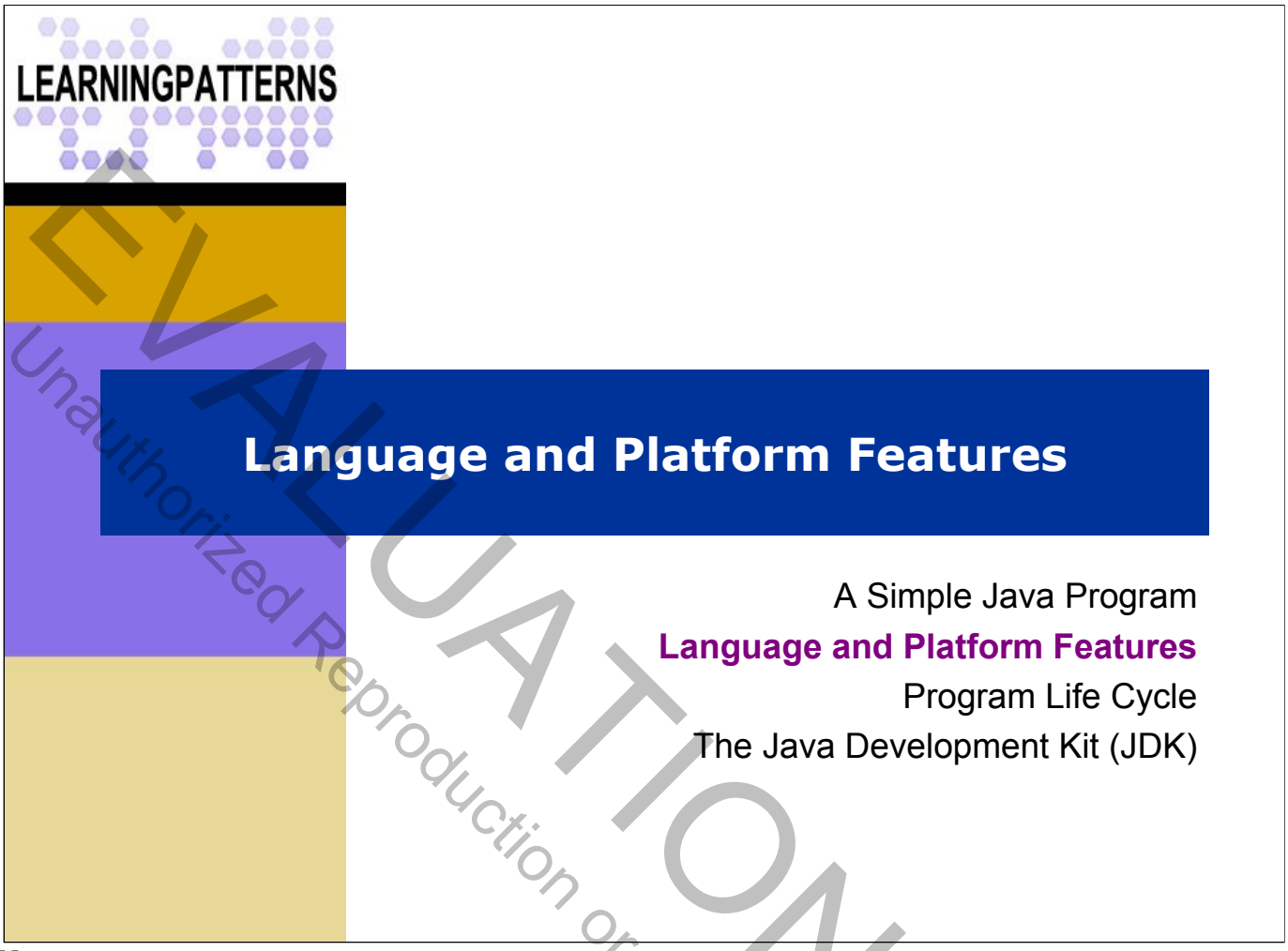
Session 1: Java Overview 13

Notes:

- Comments are text that is ignored by the Java compiler.
  - Comments can start anywhere on a line.

- **Javadoc** comments are used in generating API documentation directly from the source code.
  - These comments precede the items that they are documenting in the code.

(1) A tool called **`javadoc`** reads these comments.
  - As well as reading your source code.
  - `javadoc` also understands a number of parameters that you can embed in these comments to enhance the generated documentation.



Notes:

What is Java?

- **A programming language**
  - Strongly typed, object-oriented, general purpose language
- **A runtime system**
  - The JVM translates Java bytecode (output from the compiler) to native operating system code at runtime
- **A platform**
  - **JDK** (Java Development Kit) contains the JVM + other runtime facilities
    - Previous releases (Java 11-) also had a JRE (Java Runtime Environment)
  - The JDK is available for many operating systems
    - Windows, Linux, Mac OS, and many more
  - It includes the Java API, with many standard classes
    - e.g., `String` and `System`

Notes:

- The JRE (Java Runtime Environment) was at one point part of the Java system.
  - It was basically a slimmed down version of the JDK.
  - In Java 11, the JRE was removed, and only the JDK is provided. Users can create smaller custom runtimes using additional tools that are now provided (e.g. `jlink`)

- Sun Microsystems (acquired by Oracle) was Java's inventor, but there are many companies now involved in the development of Java specifications and technologies.
  - Most database, application server, and operating system vendors are heavily involved in Java.
  - There are also many vendors with IDEs and other tools to help with Java development, deployment, debugging, profiling, etc.
  - These companies collaborate with Oracle through something called the Java Community Process.
    - See http://jcp.org/ for more info on the JCP.
  - There is also an open source version available. See the link below for more details.
    - http://openjdk.java.net/

Java is Modern and Object-Oriented

- Supports many modern features
  - Networking, multithreading, database access, exceptions, internationalization, security, data structures, GUI, and more
- Java is **Object-Oriented**
  - **Object-Oriented Programming** is a way of thinking about and modeling systems
- **OO programming** creates types which model the real world
  - Forming an abstract blueprint of your system
    - e.g., Customer, Employee, PurchaseOrder, ShoppingCart, etc.
  - Types have properties (data) and methods (behavior)

Notes:

- The Java language itself remains relatively small – these features comes as the core Java API, a set of thousands of classes that have been designed with reuse in mind.
  - Instead of a large body of keywords and syntax to learn.
- OO Programming is a methodology or paradigm that is still evolving within the field of software engineering.
  - It's an approach that has a proven track record.

Java is Portable and Safe

- Java programs are **platform independent**
  - Java source compiles to the same **bytecode** across all platforms
  - Bytecode executes in the JVM at runtime
    - Where it's translated to Windows code, UNIX code, etc.
    - It runs anywhere there is a JVM (Write Once Run Anywhere)
- Java eliminates **error prone** features
  - e.g., pointer arithmetic and "go to" statements
- Java **helps you develop reliable software**
  - Garbage collection to prevent memory leaks
  - Array bounds checking
  - Compiler-checked exceptions
  - Removal of unsafe capabilities (e.g. raw pointer access)

Notes:

- The Java Virtual Machine (JVM) can be invoked explicitly to run a Java application, or it may be invoked implicitly by another application such as a Web server to run servlets or a Web browser to run applets.

- It is mostly true that Java code can be run, virtually unchanged, on any machine that has a JVM.
  - Hence the "Write once, run anywhere" slogan.
  - Java is consistent across all implementations – the Java Language Spec alone defines the language.
  - System administrators can keep one copy of code in one place, and it will run on all platforms.
  - However, Java code should still be tested in the target environments in which it is expected to run. There can be many subtle differences between environments that will affect your system.

- Java also has a very large, standard API of pretested, reusable code.
  - This helps you find developers who know the platform, and makes it easier to build systems with it.

Java has Multiple Platforms

- **Java Standard Edition (Java SE)**
  - Core APIs and some enterprise APIs (e.g. Web Services)
  - **Java SE Embedded** and "**compact**" profiles define subsets of Java SE with a reduced footprint
- **Java Enterprise Edition (Java EE)**
  - Platform for multitier enterprise applications
  - Depends upon Java SE
  - Adds enterprise capabilities like REST services, Messaging, etc.
- **Java Micro Edition (Java ME)**
  - Platform for small devices like cell phones
  - Subset of Java SE, with a smaller footprint
  - Lower resource usage than Java SE Embedded
- **Android**, a de-facto standard, is a different platform altogether (1)

Notes:

- Java SE Embedded and compact profiles define subsets of the full Java SE API.
  - These can run in smaller environments than the full Java SE environment.
  - They still require more resources than a Java ME environment.

(1) Android, while written in Java, actually uses a slightly different architecture.

- It does not use the standard JVM.
- Android programs are compiled into Dalvik executables (not bytecode).
- Dalvik is a VM written specifically for the Android platform, and works differently in a number of areas.

- The details of Android, Java ME, and Java SE Embedded/compact are beyond the course scope

- The name of the Java distribution has changed over many releases - mostly for marketing reasons.
  - Java 1.x release used the term JDK (JDK 1.0, 1.1).
  - Release 1.2 introduced the term Java 2 (Java 2 Release 1.2, Java 2 Release 1.3, Java 2 Release 1.4).
  - The release after Java 2 Release 1.4 used the term Java 5 (really just release 1.5 renamed).
  - The next releases built on that - Java 6, Java 7, and Java 8, … Java 11, …

Who Develops/Controls Java?

- **Sun Microsystems** introduced Java (1990s)
- **Oracle** acquired Sun in 2010
  - Still provide commercial versions of Java (free and for fee)
  - https://www.oracle.com/java/
- **OpenJDK**: Open source Java implementation
  - Started by Sun in 2006-2007
  - Became official Java reference implementation with Java 7
  - http://openjdk.java.net/
- **Java Community Process** (**JCP**): Mechanism for developing standard technical specs for Java
  - Via JSRs (Java Specification Requests)
  - https://www.jcp.org/

Notes:

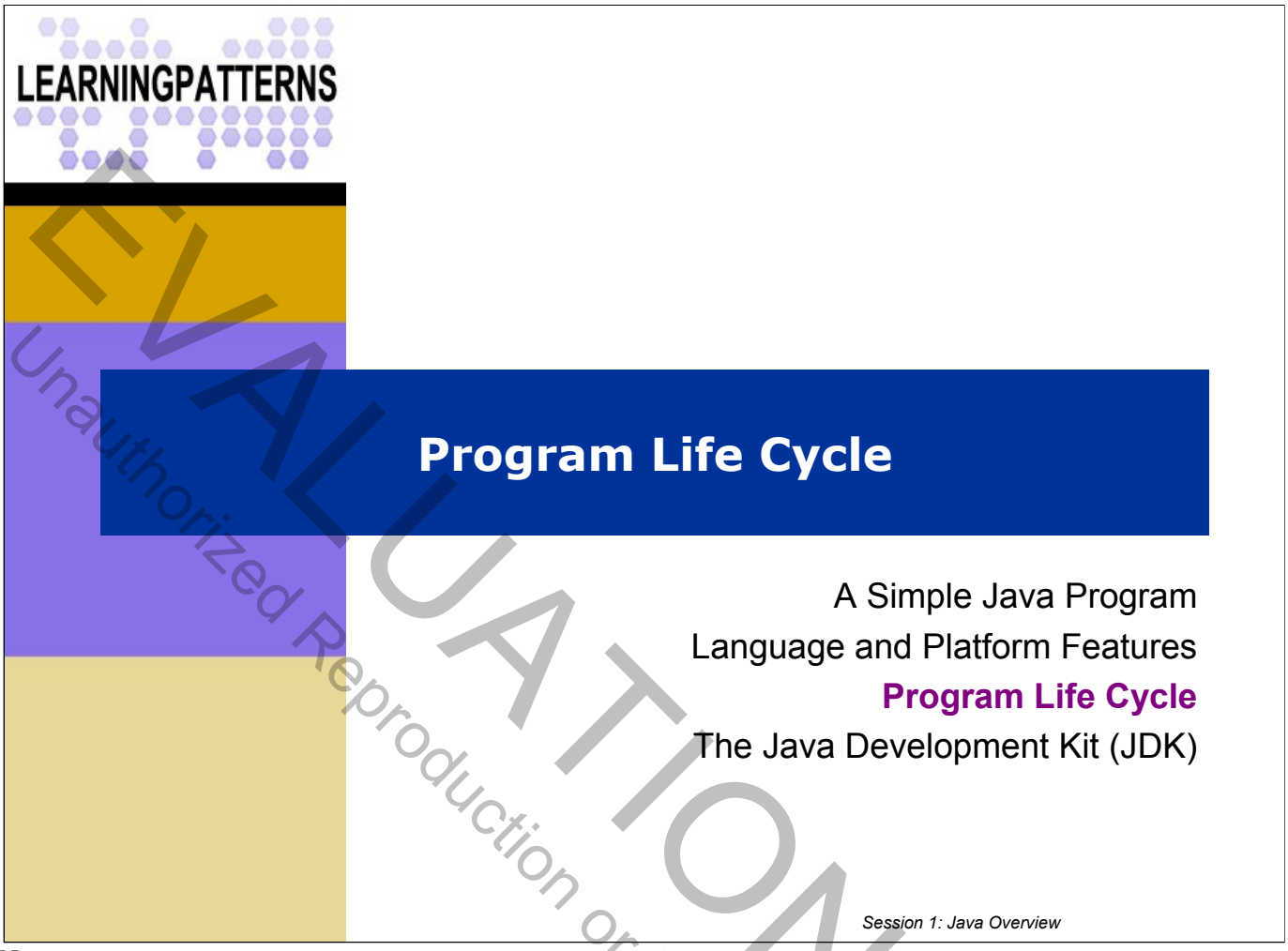
Current Java Release Cycle

- **Rapid releases** every six months (starting with Java 9)
  - Feature releases / technology previews
  - Only critical updates, which end with next release
- **Long-term support** (**LTS**) releases on a longer schedule
  - Public updates for years, plus extended support
  - Java 8 and earlier basically only had LTS releases
  - We summarize some release history below

Version	Release Date	End of Updates
Java 7	July 2011	April 2015
Java 8 (LTS)	March 2014	2019-20 (Oracle) 2022 (OpenJDK)
Java 9 (First rapid)	Sept. 2017	March 2018
Java 10 (Rapid)	March 2018	Sept. 2018
Java 11 (LTS)	Sept. 2018	2022+ (OpenJDK)
Java 12 (Rapid)	March 2019	Sept. 2019 (OpenJDK)

Notes:

- You'll also see Java 11 referred to as "18.9 LTS" because it was released in 09/2018, and is an LTS release. Similarly Java 10 is sometimes referred to as "18.3 non-LTS" because it was released in 03/2018 and is a non-LTS release.
  - This is mostly in Oracle/Open JDK documentation.
  - There is a large discussion on the naming of Java versions. It is mostly not relevant to this class, but for more info you can see the link below.
  - https://medium.com/codefx-weekly/why-java-18-3-and-whats-wrong-with-it-6844d1d5b9dc
- For more info on support from Oracle see the following link.
  - https://www.oracle.com/technetwork/java/java-se-support-roadmap.html

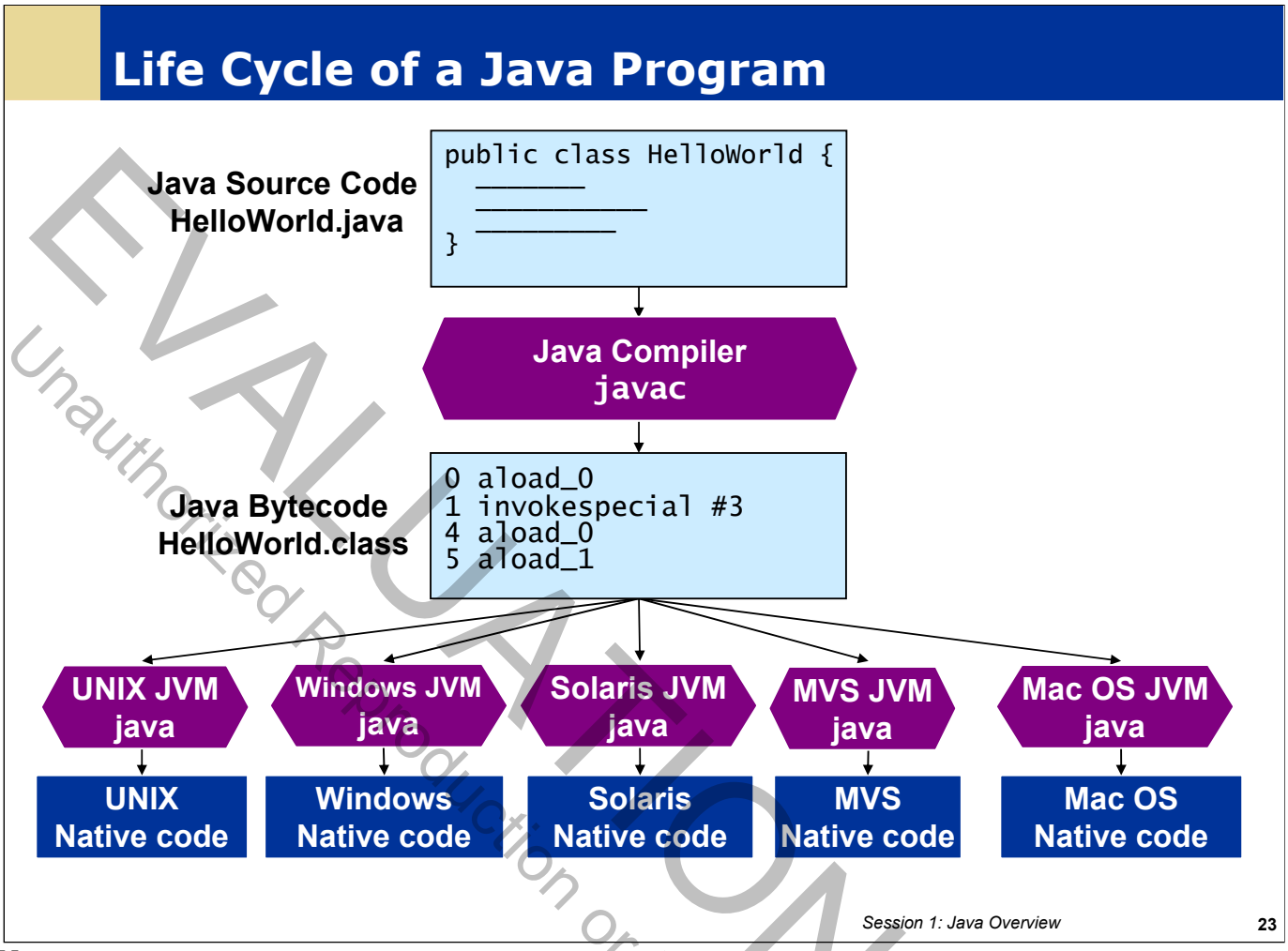


Notes:

Java Source and Java Bytecode

- Java source is in a .java file, e.g. ***MyClass.java***
- Source is compiled into Java bytecode
  - Stored in ***.class*** files, e.g. ***MyClass.class***
  - It's a **platform independent** intermediate representation
  - It's interpreted by the **JVM**
    - Think of it as native JVM instructions
  - Bytecode helps make "**write once run anywhere**" possible
- The JVM converts bytecode into the **native code**
  - Based on the target environment
  - It then runs the program

Notes:

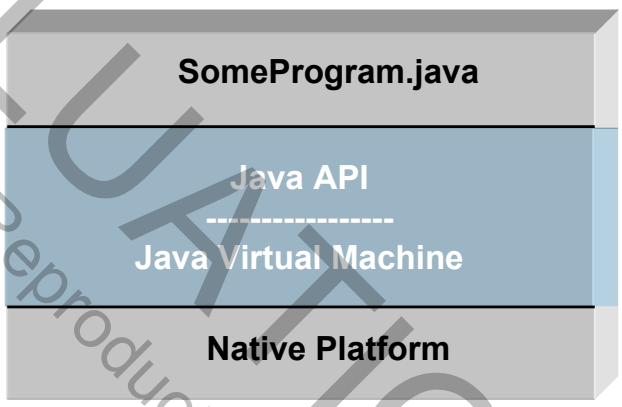


Notes:

- The JVMs for many platforms are available as standard distributions from Oracle.
  - As well as from OpenJDK.
- JVMs for other platforms are available from their vendors, as well as from OpenJDK.

Java Programs Insulated From Environment

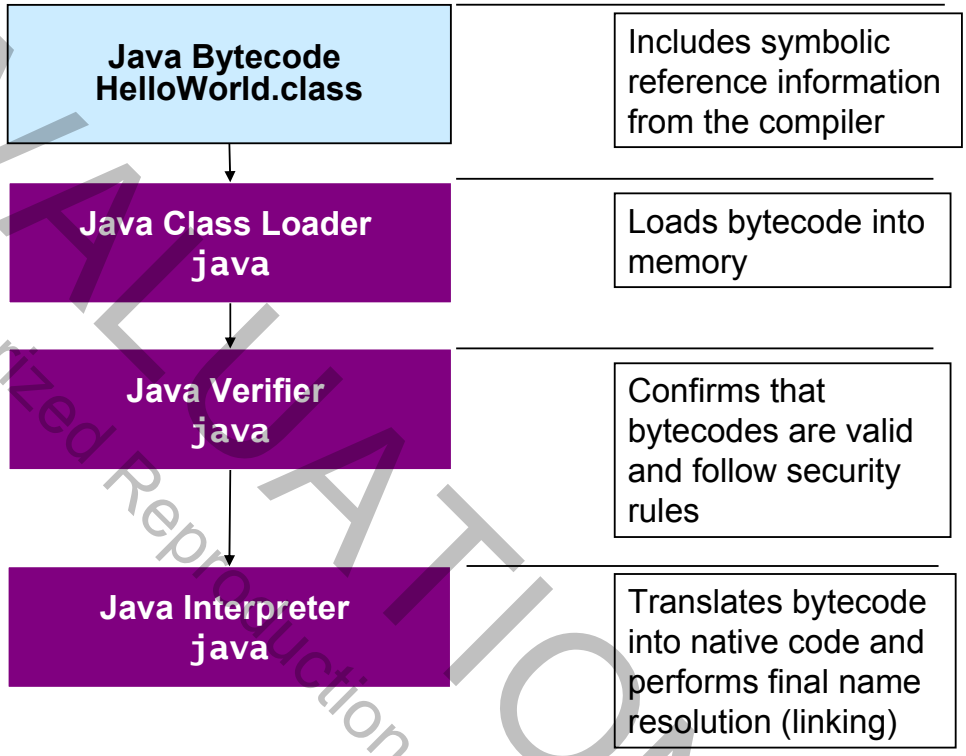
- Java provides a platform to run programs independently of the environment
  - A software-only platform of the Java API and JVM



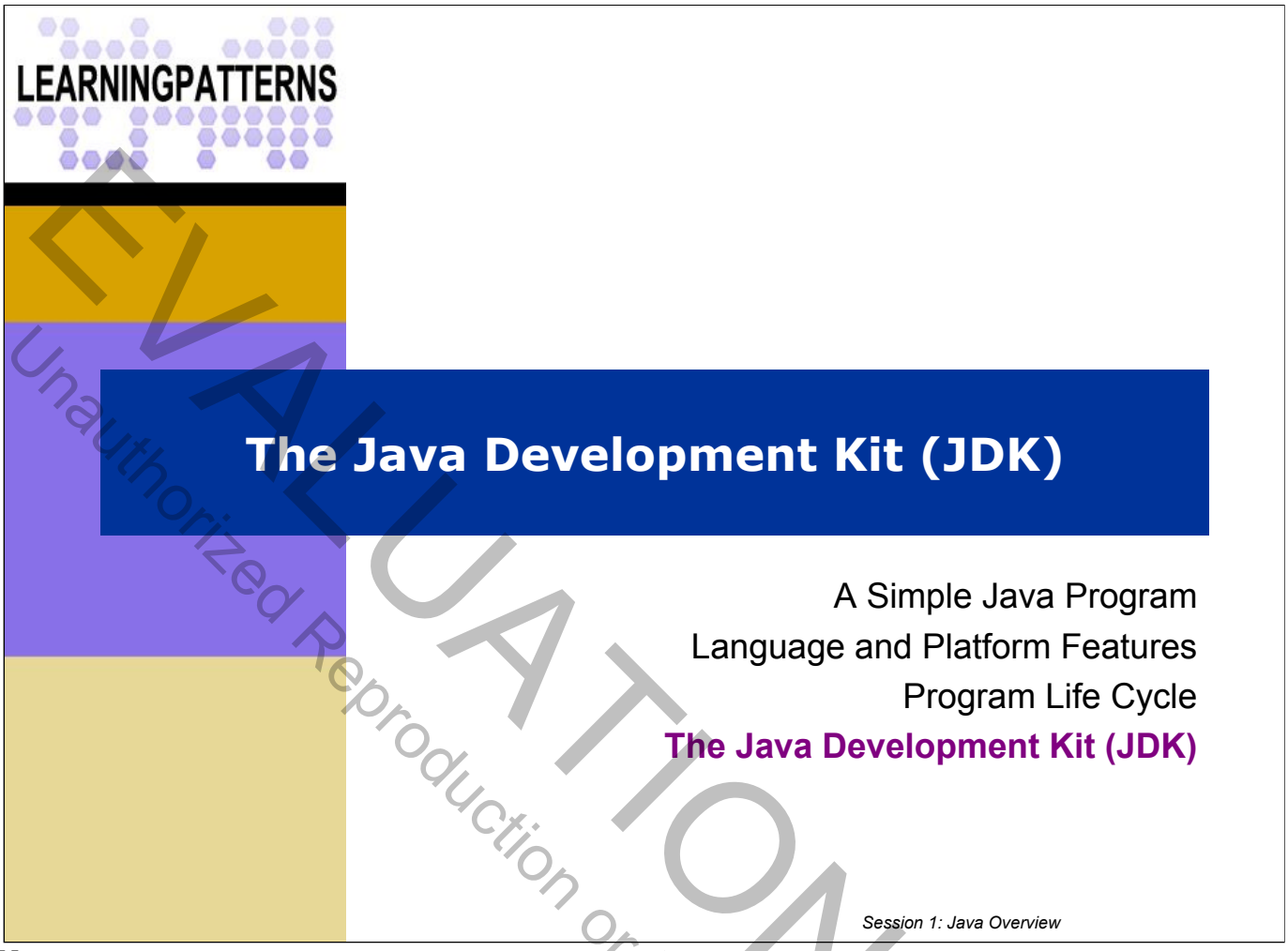
Notes:

Java is Dynamic - The Runtime Process

- Several phases take place at runtime



Notes:



Notes:

Java Development Kit (JDK)

- The Java Development Kit (or "JDK") provides all basic components for Java programming
  - Officially called the Java SE Development Kit by Oracle
- Major components include:
  - The Java core API libraries
  - Command line tools

`javac`	**compiler**
`java`	**runtime system**
`javadoc`	**documentation tool**
`jar`	**creates JAR (Java ARchive) files**
`javap`	**allows you to peek inside .class files**

Notes:

The Java API

- Java has a large **Application Programming Interface** (**API**)
- Organized into **packages** (related libraries), including:
  - **Core** (`java.lang`): Common functionality such as the `String`, `Double`, and `Exception` classes
  - **Utility** (`java.util`): Collections, internationalization, date/time, and logging
  - **I/O** (`java.io`): File I/O, buffers, File objects, abstract channels
  - **Persistence** (`java.sql`, `javax.persistence`): Database Access
  - **Networking** (`java.net`): Including TCP/IP and SSL
  - **GUI** (`java.awt`, `javax.swing`): Graphical User Interfaces (GUI)
  - **XML**(`javax.xml`, `javax.jws`): XML manipulation and Web services
- Packages are grouped into **modules**, which are covered later

Session 1: Java Overview 28

Notes:

- There is an enormous amount of functionality available with the Java SE platform.
  - These packages make it easy to build highly functional and portable programs immediately.
- We can't even include all the packages here, as there are well over a hundred.
  - And there are thousands of classes in the packages.

- There is even more functionality available in the Java EE platform.
  - It defines standards for enterprise technologies such as Web applications and distributed objects.

Downloading and Installing the JDK

- Try the following:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

- For Java 11 (Based on JDK 11.0.1 - 64 bit install)
  - Install file: *jdk-11.0.1_windows-x64_bin.exe*
  - Default installation directory:
    *C:\Program Files\Java**jdk-11.0.1***

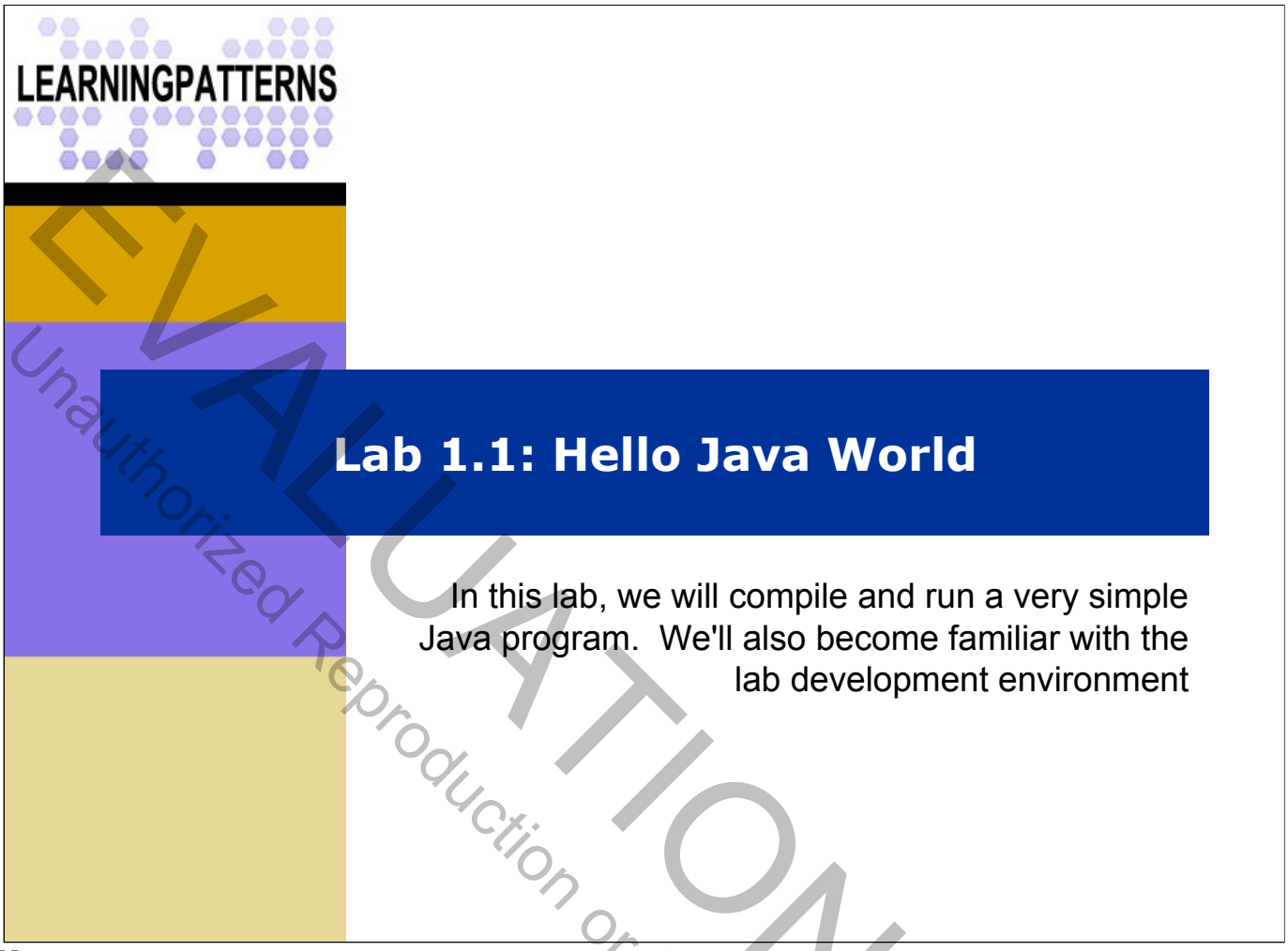
- **NOTE:** the core API documentation is not included
  - It can be downloaded separately as a zip file
  - Unzip it into the Java installation directory

Session 1: Java Overview 29

Notes:

- If you want the OpenJDK versions, try
  - https://jdk.java.net/11/

- As update releases come out, their install files and install folders generally include the update version as part of the name (e.g. 11.0.1).
  - This is true of older builds also.
  - Generally, the rapid releases (e.g. Java 9) do not have updates. Occasionally, if there is an urgent fix or security issue, there will be an update release.



Notes:

Review Questions

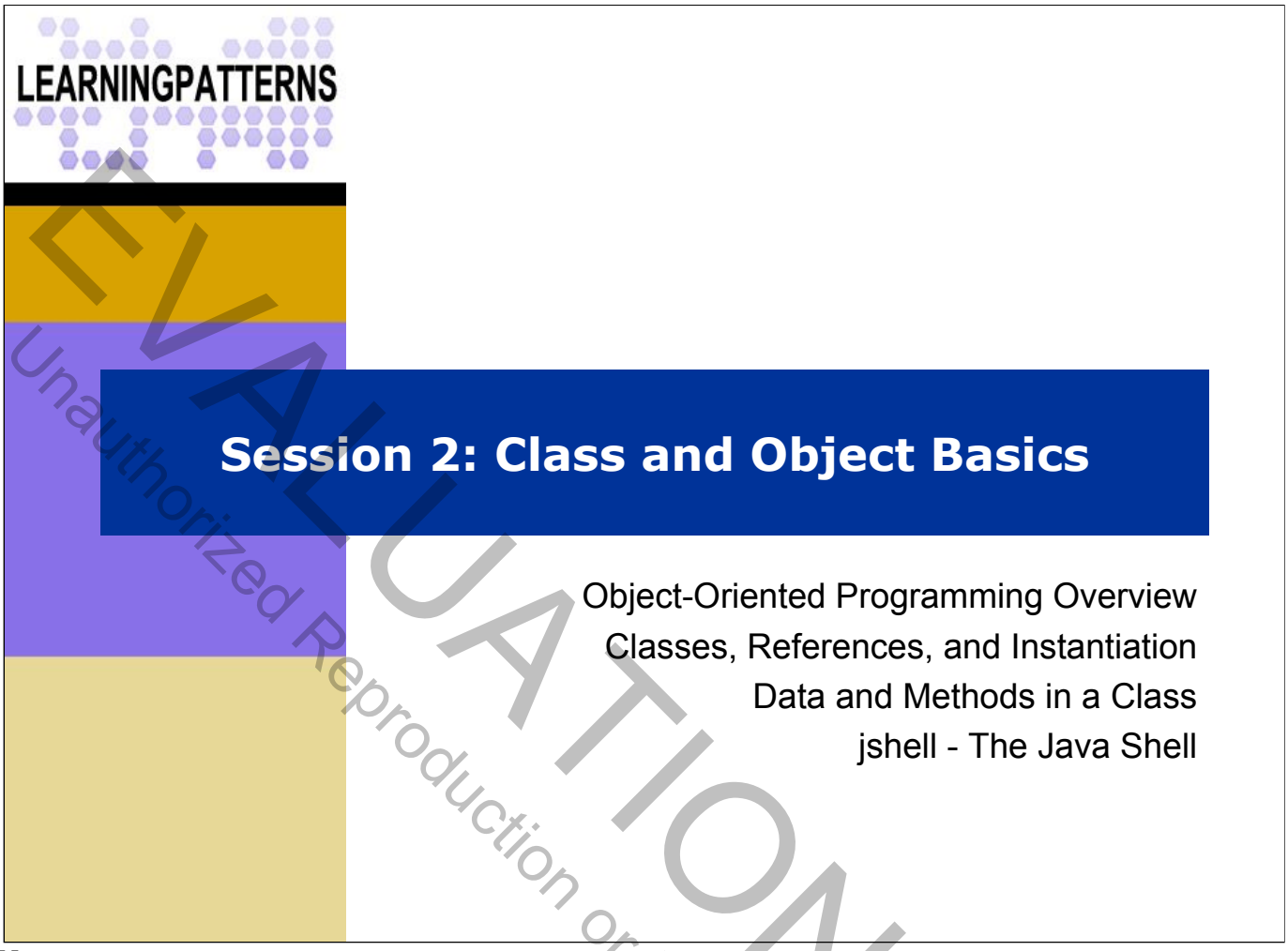
- Explain the Java program development and runtime lifecycle.
- What is the purpose of the main method?
- What is portability? How does Java achieve it?
- What is the role of the Java virtual machine?
- What is the JDK? Name some of its tools and what they are used for.

Notes:

Lesson Summary

- Java source files have a ***.java*** extension
  - They're **compiled into bytecode** (*.class* files) with javac
  - They **run in the JVM**, which converts them to the target platform
  - The JVM provides an **independent operating environment** for Java programs
- The **`main()`** method is the entry point for all Java programs
  - It appears in a Java class definition
- Java programs can run unchanged in different environments and operating systems.
  - Done by using the **same bytecode format** and **standard libraries** across all platforms
- The JDK provides tools for working with Java
  - Including **`javac`** (the java compiler) **`java`** (The JVM), and others

Notes:

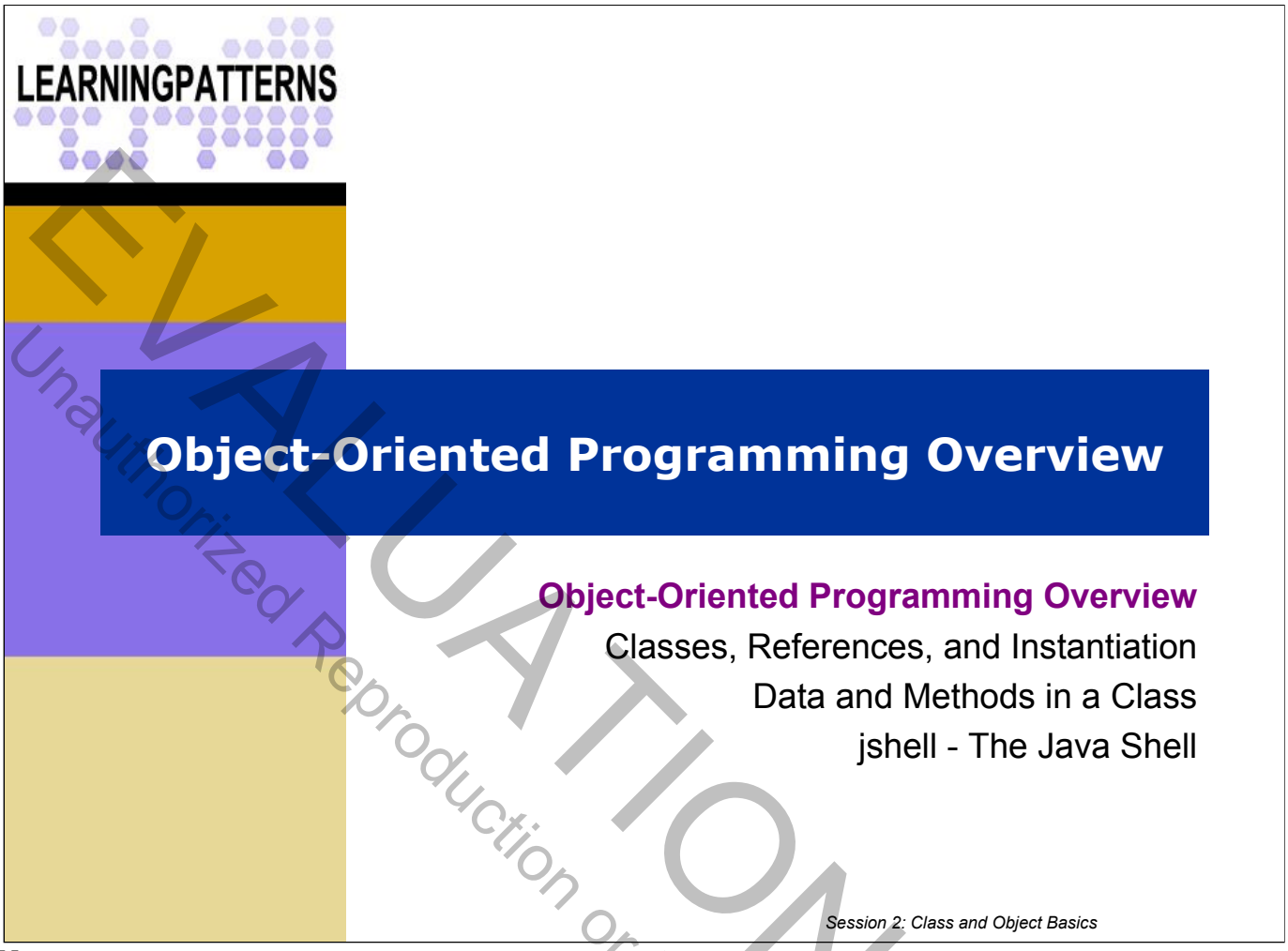


Notes:

Session Objectives

- Provide an **Object-Oriented Programming** (OOP) overview
  - Principles and advantages of OOP
- Discuss the major **characteristics of an object**
- Understand **class** vs. **object**
  - And **object** vs. **reference**
- Create/test a Java **class definition**
  - Instance variables, getter/setter methods, and constructors
  - Creating objects
- Explain what **encapsulation** is and why we use it
  - Do it in Java

Notes:



Notes:

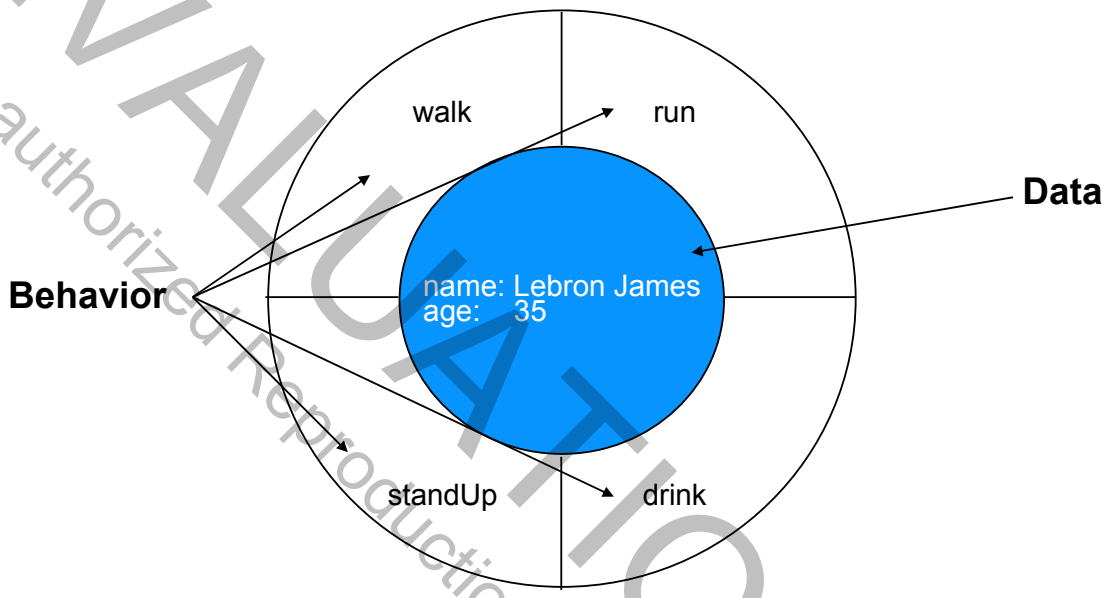
What is Object-Oriented Programming (OOP)?

- A way of thinking about and modeling systems
  - One of many methodologies within software engineering
  - Scientifically sound, with a proven track record
- Basic principal
  - **Ties data together with the code** that manipulates it
  - Organized in elements called **classes** or **types**
- A type **represents** a **concept**, **abstraction**, or **thing**
- **Objects** (instances of a type) **work together**
  - They perform the functionality of a software system

Notes:

What is an Object?

- A software representation containing **data** and **behavior**
  - e.g. a person's name and age, with various behavior



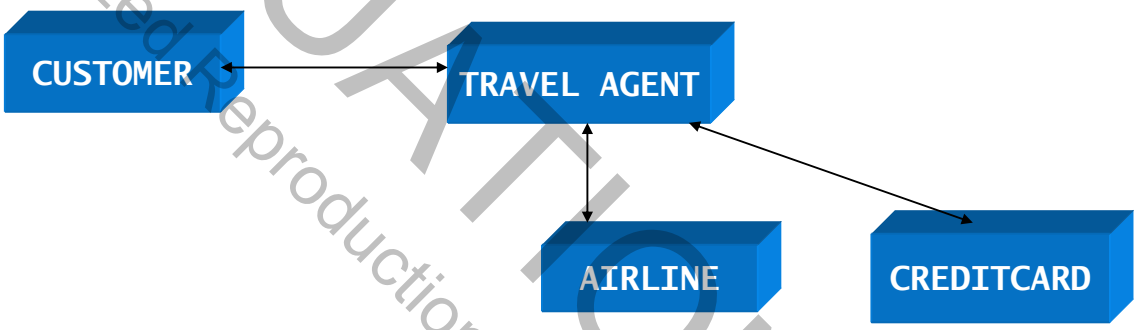
Session 2: Class and Object Basics 37

Notes:

- This is a common graphical representation of an object.
  - We'll see that there is a reason the data is in the middle.
  - It is a common, and recommended, practice that users of an object only interact with the object's behavior.
  - Users generally should not manipulate data directly.
  - We'll talk more about this later.

Important Object Characteristics

- **Type or Class**: The kind of thing an object is - its classification
- **Data or Properties**: Information associated with an object
- **Behavior or Methods**: Actions a type provides
- **Identity**: The unique existence of any object, independent of its characteristics
- OOP organizes a program around **types** and **objects**
  - Not around functions or processes
  - An object-based system is a set of collaborating objects:



Session 2: Class and Object Basics 38

Notes:

- These characteristics may be called different things by different people.
  - For example type vs. class.
- In general, we use the different terms fairly interchangeably.
- There are many views on what is important about an object.
  - These are core characteristics that most agree on.

About Object-Oriented Programming (OOP)

- **Core Principles** of OOP
  1. Everything is an object
  2. Objects make requests of each other to do work
     - Delegation, or divide and conquer
  3. Every object has its own data, which may consist of other objects
  4. Every object is an instance of a type – a type groups similar objects
  5. The type is the repository for an object's behavior
  6. Types are organized into an inheritance hierarchy

- Some advantages of OOP
  - Handles large, complex systems better
  - More flexible, maintainable, and reusable code
  - Models the real world - better accuracy in modeling
  - Scalable - ability to grow with the organization
  - Faster implementation (large API, simpler syntax, portability)

Session 2: Class and Object Basics 39

Notes:

- The OO advantages are achieved by adding another level of organization on top of structured code.
  - OO models also more closely model the real world.

- Alan Kay, inventor of Smalltalk, articulated the concept of recursive design, followed by his six principles of object-oriented programming.

- In a pure OO language, like Smalltalk, an objects memory consists completely of other objects.
  - In some OO languages, like Java, an object can also hold primitive data, for example a number, which is not a full fledged object.

What's a Type?

- Generally, the way that you classify objects, for example:
  - **Concrete**: Person, Car, Planet, Star
  - **Concepts**: Number, Democracy
  - **Events**: Phone Call, Earthquake
- The types you use depend on your **problem domain**
  - In Java, types are represented by classes and interfaces
- **Determining your types** is a key step in creating a system
  - What these are is not clear-cut
  - Discovering them, along with their interactions, is a creative process

Notes:

- Almost anything can be a type.
  - A type's characteristics will also depend on the problem domain.
  - How you name your types and interact with them will also be dependent on the domain.
- There are many techniques for determining the types in a system
  - They are beyond the scope of this introductory course.

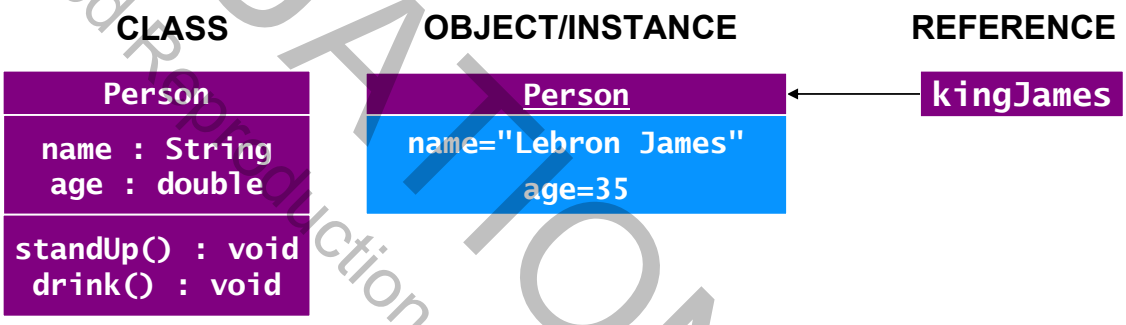
Types, Instances, and Property Values

- A type is a **blueprint** for objects (but not the object)
  - A person's definition (the `Person` type) is not actually a person
- Actual objects are called **instances** of their type(s)
  - You, the students, are instances of the `Person` type (one would hope)
  - Creating instances of a type is often called ***instantiating*** that type
- Each instance has its own values for its properties
  - You are all `Persons`
  - But you all have your own names, ages, incomes, etc.

Notes:

Classes and Objects

- A **class** defines a new type in Java
  - It's the **blueprint** describing its properties and behavior
  - Properties hold **information**
  - Methods perform the **behaviors** of an object
- An object, or instance, is the actualization of a class
  - As shown in the UML-based diagrams below (1)



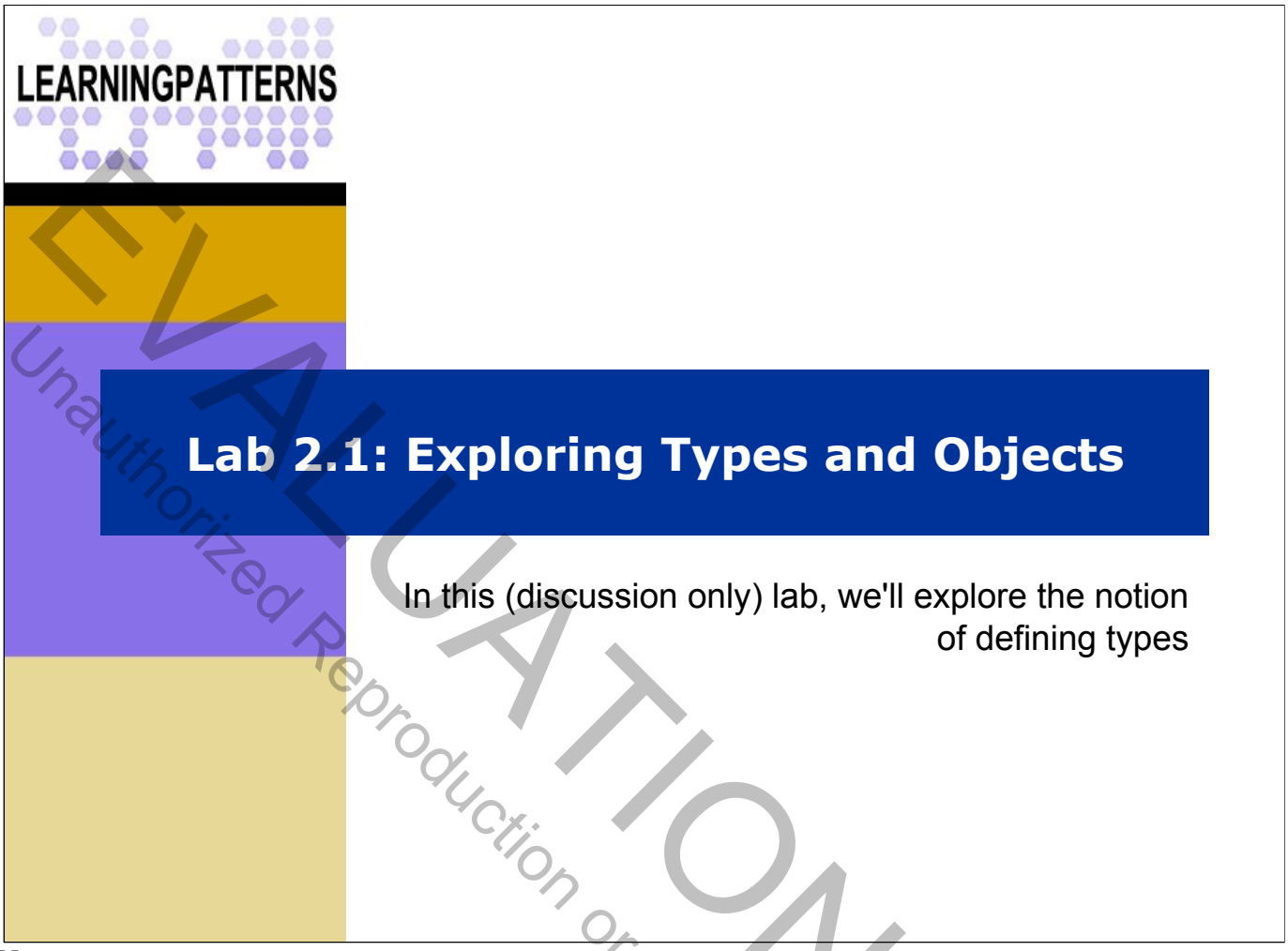
Session 2: Class and Object Basics 42

Notes:

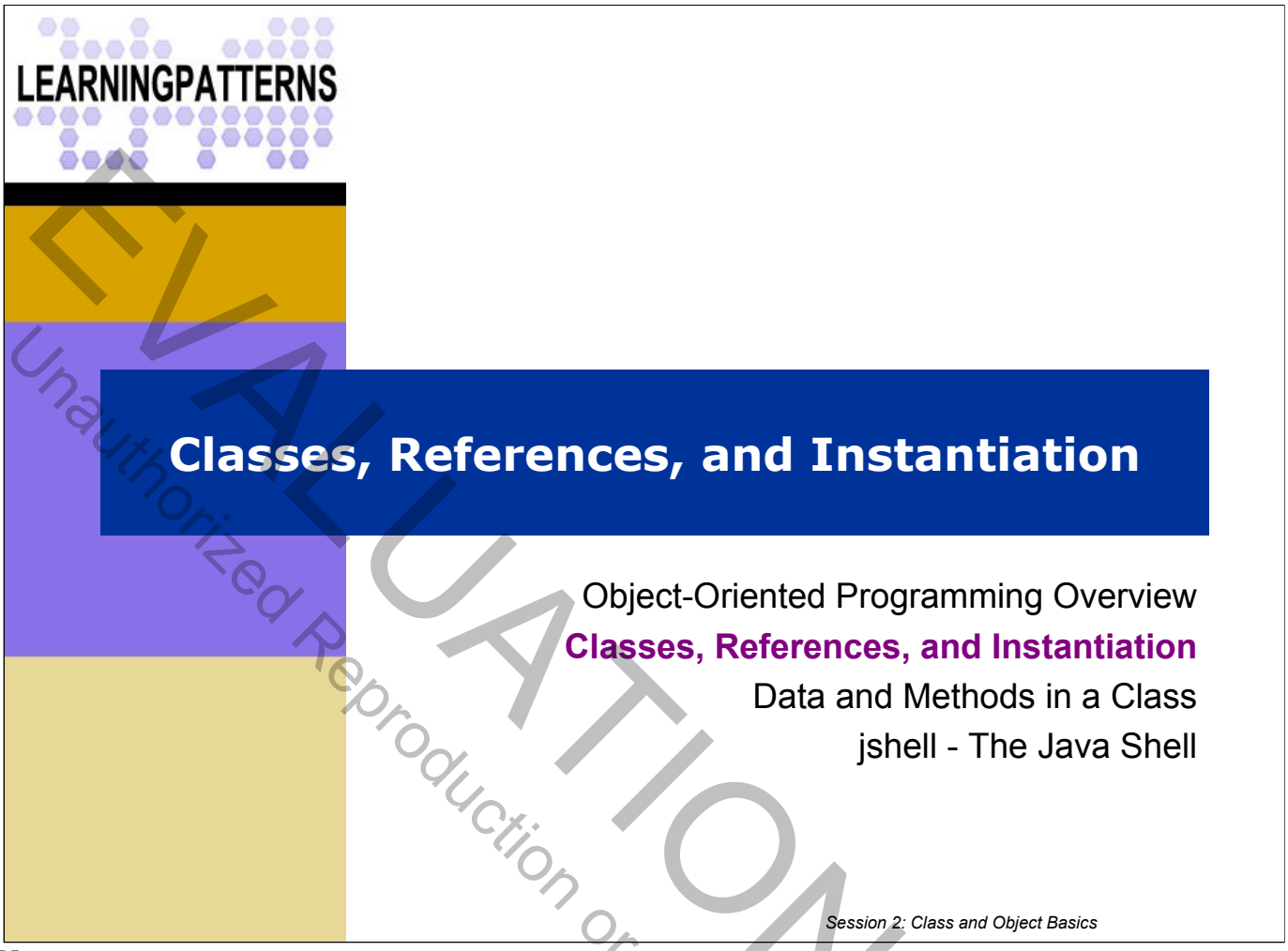
(1) Note that the method of diagramming classes used above is similar to class diagrams used in UML – the Unified Modeling Language.

- UML has become a standard notation for modeling OO systems.

- If `Person` is a class it might contain `name` and `age` properties.
  - Our `Person` class might have methods to `standUp()` and `drink()`.
- Another example: a cookie is an instance of a cookie recipe.
  - The recipe can be used to make an unlimited number of cookies.
  - The cookie is the object and the recipe is the class that describes how to make cookies.
- The rectangle underneath the Class header denotes the class and consists of the following.
  - The classname (`Person`)
  - The data fields, with their type (e.g. `name : String`)
  - The methods with their return value (e.g. `standup() : void`)
- The rectangle under the Object/Instance header denotes an instance (indicate by underlining the classname `Person`) and consists of the following:
  - The classname (`Person`)
  - The data fields, with their values



Notes:



Notes:

The Class in Java

- A **class** defines a new Java type
  - Defined in a ***class definition*** (see below)
- It's a **central concept** in Java
  - Data is defined in its **fields**, or **instance variables**
  - Behavior/code is defined in its **methods**
  - There is no global data, nor global methods

```
// define a class to represent an alarm clock
public class AlarmClock
{
  // More detail to come …
}
```

- The `class` keyword introduces the class - named `AlarmClock`
  - Curly braces `{ }` mark out the body of the class
    - Data and methods are declared within the body
  - We'll talk about the `public` keyword later

Notes:

- There are a few directives, such as the `import` and `package` directives, that appear outside a class.
  - They don't actually create anything, or execute code. We will discuss them later.

A Class Definition is a Blueprint

- A class definition is a ***blueprint*** to create instances
  - Writing it doesn't create instances
  - It just shows what they'll be like when you do create them
- A class definition is like a **cookie cutter**
  - **It's not a cookie**
  - To make a cookie, you cut cookie dough with the cutter
  - To make an object, we cut one out from object dough (memory)
    - The class definition is the cookie cutter
- Creating the object is also known as ***instantiating*** the class

Notes:

Creating and Referencing Objects

- Use **new** and the class name to ***instantiate*** instances
  - **new AlarmClock()** creates an instance of `AlarmClock`
  - It is created on the "heap" (1)
- Objects live in a computer's memory
  - Unlike cookies on a cookie sheet that we pick up and eat
  - How do we pick up an `AlarmClock` instance? With variables.
- **Variables** can reference an object instance
  - They have a **name** and a **type**
  - Below, the left hand side declares an `AlarmClock` variable
  - It refers to a newly created `AlarmClock` instance

```
AlarmClock anAlarmClock = new AlarmClock();
```

Session 2: Class and Object Basics 47

Notes:

- When you're starting out with Java, it can be hard to understand when you're creating objects.
  - As opposed to just passing them around.
  - The **new** operator is a sure indication that an object is being created.

(1) The heap is a common pool of memory that is available to your program.
  - Think of the heap as a huge ball of memory dough that you can pull chunks off of to make objects.

- Note that we'll often use the terms **instance** and **object** interchangeably.

- Java does type checking on assignments like that above.
  - Not considering inheritance, this means that the types must be the same.

```
// let's say you had another class - Watch
AlarmClock aClock = new Watch();  // ERROR
```

- Statements are made up of expressions, which declare variables and compute values.

Identity and Object References

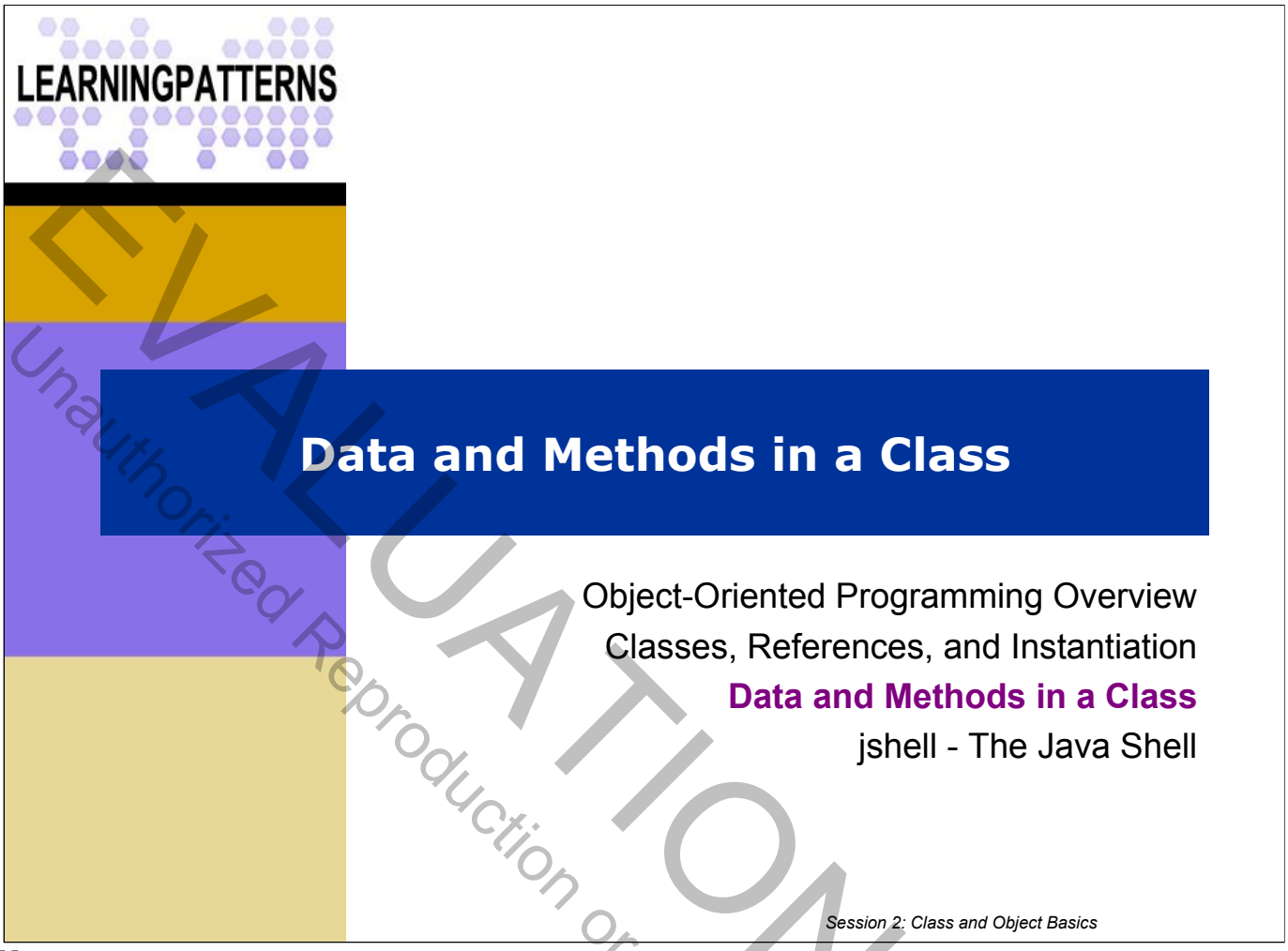
- ***Object references*** refer to objects
  - They're handles on objects that we use to access and work with them
  - Multiple references can refer to one object
- Each object has a unique existence or ***identity***
  - Independent of any properties or references
  - A property may be a unique identifier, but that's not the identity
    - Your existence doesn't depend on your unique social security number

Mini-Lab

- Every look at the instructor - you all agree they exist, right?
  - Does their identity depend on their name (a property)?
- Now, point to the instructor (It's OK that it's not very polite)
  - You're all referring to one person - you are **references** to them
  - But you're NOT the instructor
  - Similarly, object references are not the object - just handles on it

Notes:

- At an implementation level, an object reference can be different things.
  - In Java, it is just an opaque reference whose internals are hidden.
  - In other languages, e.g. C++, it can be a pointer.



Notes:

Storing Data in Objects

- A Class declares data in its ***fields*** (or **instance variables**) (1)
  - Each instance stores its own values
  - These values can **vary** from **instance to instance**
- If not initialized, fields are initialized to a **default value** (2)
  - Zero for numeric values, null for references

```
// class AlarmClock with field for snooze time
public class AlarmClock {

  // Variable declaration
  int snoozeInterval;
}
```

Session 2: Class and Object Basics 50

Notes:

(1) The language spec uses the term field.

- General literature uses both field and the term instance variable, which was used in other OO languages like C++.

(2) Default values include the following.

- Integer primitive types: **`0`**
- Floating point primitive types: **`0.0`**
- boolean primitive type: **`false`**
- char primitive type: **`\u0000`**
- class type: **`null`**

About Fields

- Each instance contains a copy of its fields
  - Each instance has space for all its fields
  - This makes sense – every person has their own name, each clocks stores its own time
- You can initialize a field's value as shown below
  - Instead of using the default value
  - The field is initialized immediately after the instance is created

```
public class AlarmClock
{
  // declare and initialize a field/instance variable
  int snoozeInterval = 5;
}
```

Session 2: Class and Object Basics 51

Notes:

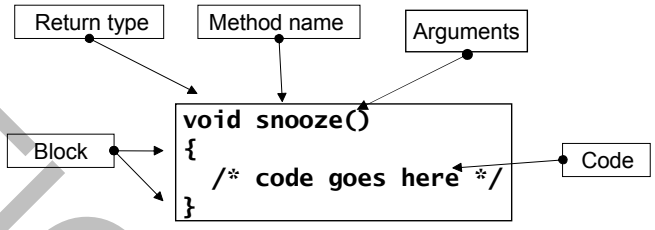
- Don't be confused about when the initialization of the field occurs.
  - It only happens when you create an instance of the class.
  - Until then, the field declaration is just a blueprint - like the shape of the cookie cutter.
  - When you create the instance (cut out the cookie) then its field gets the value you have as an initializer (as the cookie gets the shape of the cookie cutter you use).

Behavior and Methods

- Behavior is defined in ***methods*** within the class body
  - They associate executable code with classes

```
public class AlarmClock {
  // snooze() is often called a business (or behavioral)
method
  void snooze() {
    System.out.println("ZZZZZ");
  }
}
```

- Methods can be passed arguments and return values
  - A `void` return means nothing is returned
  - More on this later



Session 2: Class and Object Basics 52

Notes:

- All method definitions must appear within the body of a class definition.

- `snooze()` is a normal method of our class.
  - It's sometimes called a business method to differentiate from other kinds of methods like accessor methods that we'll cover shortly.
  - Calling it a business method is really just documentation; it's just a method, just like accessor methods.

Invoking Methods

- Invoke methods with an instance, followed by the ***dot operator*** **.** and the method name
  - You **always** need an instance for a regular method

```
referenceVariableName.methodName()
```

```
// Here's a program that uses AlarmClock

public class EarlyMorning
{
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    AlarmClock yourClock = new AlarmClock();

    // We have two instances - call snooze() on one of them
    myClock.snooze();
  }
}
```


Notes:

Data Access in Methods

- Methods can access instance variables
  - Just refer to them by name (e.g. `snoozeInterval`)
  - Uses the value of the invoking instance
    - Since there is **always** an instance used with a normal method call
  - The program at bottom will output

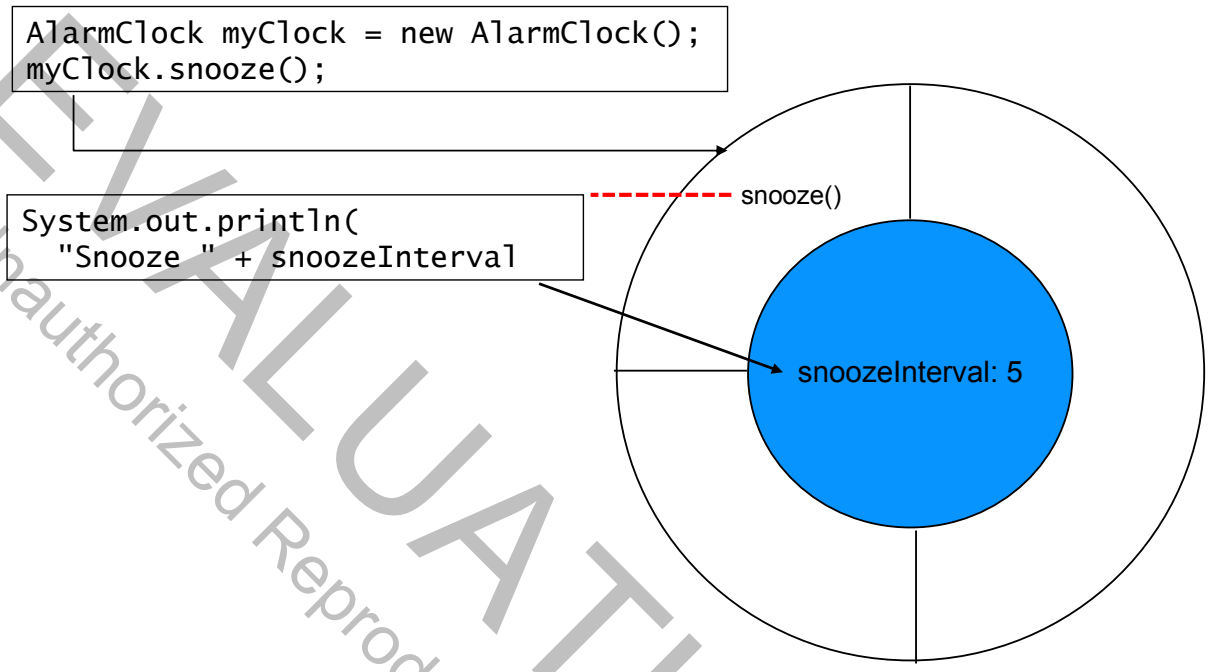
    Snooze 5

```
public class AlarmClock {
  int snoozeInterval = 5;
  void snooze() { System.out.println("Snooze " + snoozeInterval };
}
```

```
public class EarlyMorning {
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    myClock.snooze();
  }
}
```


Notes:

Pretty Pictures



Notes:

Java Naming Conventions

- ***Identifiers*** start with a **letter**, **underscore _**, or **dollar sign $**
  - Subsequent characters can be digits
  - Useable as variable names, class names, method names, etc.

- **Java naming conventions** (recommended, not required)
  - **Classes**: Generally **nouns**, spelled as **CamelCase**
    - `String`, `System`, `HelloWorld`, `TextField`
  - **Variables**: Spelled as **camelCase**
    - `salary`, `hairColor`, `maxUpdateInterval`
  - **Method**: Generally **verbs**, spelled as **camelCase**
    - `getSalary()`, `dye()`, `setHairColor()`, `service()`

Session 2: Class and Object Basics 56

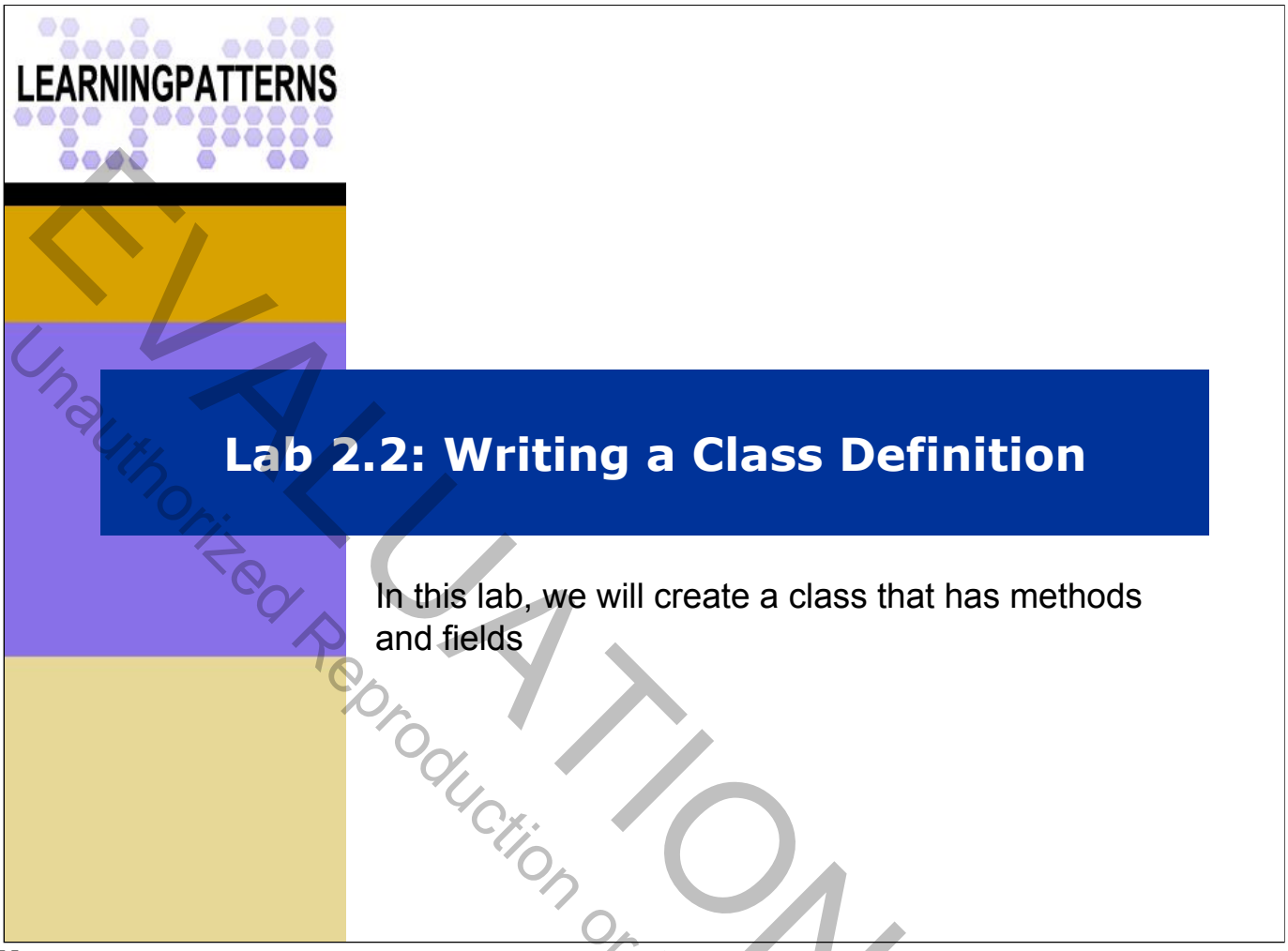
Notes:

- For Oracle's recommended Java coding conventions, see the following link.
  - http://www.oracle.com/technetwork/java/index-135089.html
- For Google's (much more up to date conventions) see this link.
  - https://google.github.io/styleguide/javaguide.html

- Java reserves a number of identifiers to be used as keywords.
  - Keywords can't be used as variable names.
  - Here are a few of them.

```
break, case, catch, class, else, extends, final, finally, import,
  interface, if, public, return, super, switch, throws
```

  - `true` and `false` are `boolean` literals, and technically not keywords.
  - `null` is the "null literal" and is also not technically a keyword.



Notes:

More About Variables

- Variable declarations have the basic form:

***TypeSpecifier Name = VariableInitializer*$_{opt}$**

```
int n;                  // no initialization
int j = 0;              // with initialization
int k = j + 1;          // initialize with expression
float r = 0.0F;         // float (decimals default to double)
```

  - The type indicates the kind of data and legal values
    - Sometimes called the *compile-time* type
  - Variable assignments are checked at compile time
    - To make sure a compatible value is assigned

- The lines of code above are ***statements***
  - The smallest independent units in a Java program
  - Simple statements like these **end with a semicolon**

Session 2: Class and Object Basics 58

Notes:

- **Integer values** default to type `int` (recommended usage).
- **Floating point values** default to type `double` (recommended usage).
  - That is why you need the `F` in the last example above. Without it, you are trying to assign a 64-bit value (`0.0`) to a 32-bit `float` variable. The `F` in `0.0F` indicates a 32-bit value.

- All variables must be declared somewhere in a class.
  - e.g., they might be declared as fields in the class itself, or as local variables in a method.
  - There are no global variables.

About Java Primitive Data Types

- All data in Java has a type
- The possible types are:
  - Class or interface type, e.g., `String`
  - A **primitive type**:

Type	Size (bits)	Range
`byte`	8	-128 to 127
`short`	16	-32,768 to 32,767
`int`	32	-2,147,483,648 to 2,147,483,647
`long`	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	32	±1.4E-45 to ±3.40E+38
`double`	64	±4.9E-324 to ±1.797E+308
`char`	16	Unicode character set
`boolean`		`true` or `false`

Session 2: Class and Object Basics 59

Notes:

- All numeric types are signed, that is they can hold both positive and negative numbers.
- `float` and `double` conform to the 1985 IEEE 754 standard, including `NaN`, `Infinity`, etc.
- To ensure portability, primitive types have the same strictly specified representations on all platforms.
- Java uses the Unicode character set – this 16-bit international character set includes:
  - The characters "A" through "Z" and "a" through "z".
  - Other familiar characters, such as "$" and "_".
  - A number of foreign language character sets.
  - ASCII is a subset of Unicode (the first 256 characters).

Numeric Literals

- **Integer** literals can be:
  - **Decimal**/Base10 (default) **16**
  - **Octal**/Base8, with prefix of **0** **020**
  - **Hexadecimal**/Base16, with prefix **0x** or **0X** **0x10**
  - **Binary**/Base 2 with prefix **0b** or **0B** (Java 7+) **0b10000**
- **Floating point** literals can be:
  - Decimal **31.38**
  - Scientific notation, using **E** (preferred) or **e** **3.138E+1**
- Default for integer literals: **32-bit `int`**
- Default type for floating point literals: **64-bit `double`**
  - Use **F**, **D** to explicitly indicate `float`, `double` values (`2.5F`, 1.2**D**)
  - Use **L** to explicitly indicate `long` values (10**L**)

Session 2: Class and Object Basics 60

Notes:

- You might need to use **L** or **F** to override the default for numeric literals (`int` and `double`).

```
// Use an L to indicate that a literal is a long
// The number below is too big for a 32 bit int
long big = 21474836470L;

// Use an F to indicate a literal is a float.
// Without it, you may try to assign a
// 64-bit value to a 32-bit float variable
float salary = 456.76F;  // This works
float salary = 456.76;   // This does NOT work
```

Non-Numeric Literals

- **boolean** literals: `true` or `false`
- **char** literals: character in single quotes: `'?'`
  - Or 4-digit hex Unicode character number: `'\u0063'`
- Several **special characters** use escape sequences
  - `\n` for a new line and `\"` for a literal double-quote
  - Some others are listed in the notes
- Java allows underscores between numeric literal digits
  - For readability - there are restrictions (1)

```
long creditCardNumber = 1234_5678_9012_3456L;
```

Session 2: Class and Object Basics 61

Notes:

Escape Sequence	Name
`\f`	form feed
`\t`	tab
`\n`	line feed
`\r`	carriage return
`\"`	double-quote
`\'`	single-quote
`\\`	backslash
`\udddd`	Unicode number

(1) You can place underscores only between digits; you cannot place underscores:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

Strings

- String literals are sequences of doubly-quoted characters:

```
"Hello World"
```

- Java handles strings in a special way
  - Class **`String`** defines the string representation
  - Declare a string reference with `String`
    - Can initialize it with a **string literal**
    - Doesn't need the **new** keyword
    - The only place in Java objects are created without using new

```
String hi = "Hello";
```


Notes:

Arithmetic Operations

- Java provides standard arithmetic operations, including:
  - Addition (+), Subtraction (-), Multiplication (*), Division (/)
  - String concatenation (+)
  - Numerical Equality (==)
  - Bitwise operators
- Java also has logical operators, that include:
  - Logical OR (| and ||) Logical AND (& and &&)
  - NOT (!)

Session 2: Class and Object Basics 63

Notes:

- If you want a full list of the operators, see the Java Language Specification.
  - For anyone with programming experience, they should be unsurprising.

Primitive Type Conversion and Casting

- Conversion / casting take place at compile time
  - **Upcasts** are implicit and automatic (from smaller to larger size)
  - **Downcasts** must be explicit (from larger to smaller size)
    - Since you might lose data and/or precision

```
float diameter = 6.77F;
float pi = 3.1415927F;
double circum = pi * diameter;  // auto "upcast" to double

float approx = (float) circum;  // explicit "downcast"
```

- Automatic "upcasts" may occur during operations
  - Integers converted to floating points
  - Smaller size types converted to larger size types

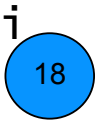
```
int   + float  -> float       // int converted to float
float + double -> double      // float converted to double
```


Notes:

Primitive Types are Value Types

- Primitive types are **value** types
  - While object references refer to something holding the data
  - Variables of a primitive type, **hold the value itself**
  - The variable name is an alias for the chunk of memory holding the data
  - For example, the below: looks like this in memory:

```
int i = 18;
```



Notes:

jshell: Read-Eval-Print-Loop (Java 9+)

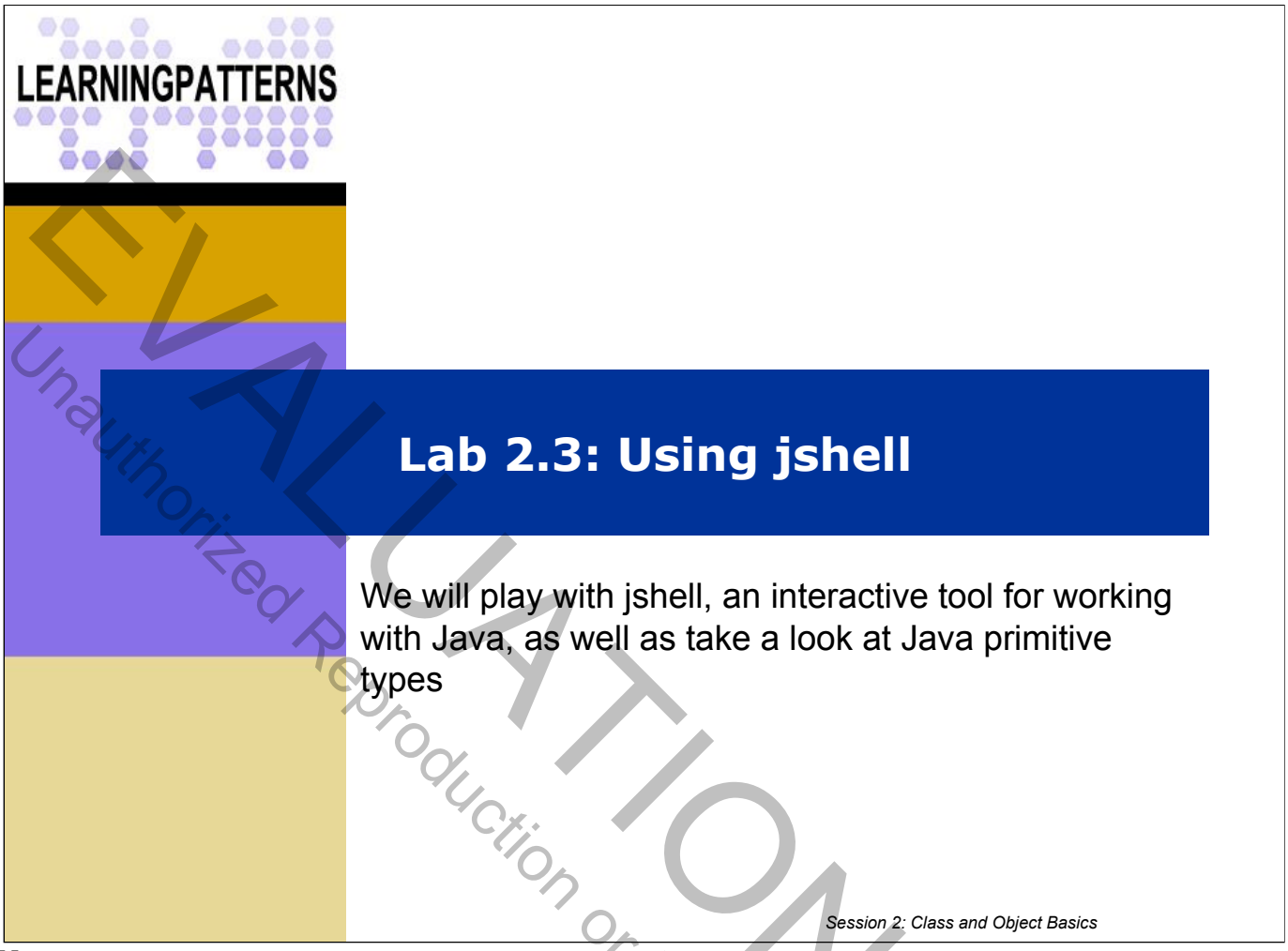
- An interactive tool to work with Java
  - Evaluates expressions, statements, and declarations
  - Automatically includes (imports) common Java libraries
  - Great for learning about Java
- Just start it up, and play, as shown below
  - We'll do that now in a lab

```
>jshell
|  Welcome to JShell -- Version 11.0.1
|  For an introduction type: /help intro

jshell> int i = 5+3;
i ==> 8

jshell>
```


Notes:



Notes:

Review Questions

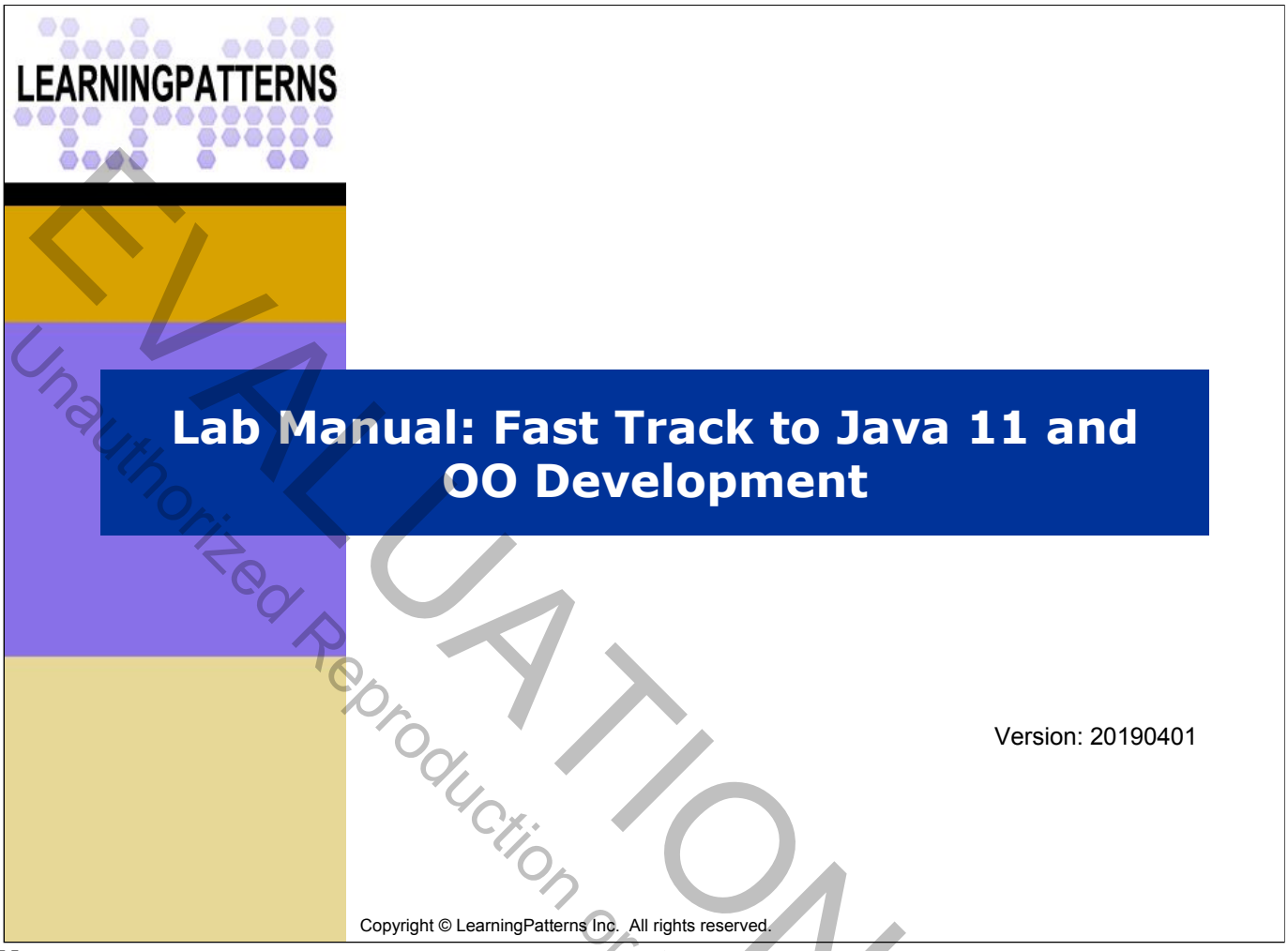
- What's a Java class, and how is it different from an object instance?
- What's the difference between an object instance and its references?
- What are some characteristics of an object instance?
- What are the default values for fields (instance variables) that are not explicitly initialized?

Notes:

Lesson Summary

- A **class** (or type) is a **blueprint** for objects – but not the object
  - Declared in a class definition defining fields and methods
  - Fields describe the **data** an instance holds, methods its **behavior**
- An **instance** is an **actual object** of some type
  - An instance can be **referred to** with a **reference**
  - This is a **handle** on the object
- Every object instance has a type
  - Each instance has its **own data**, and a **unique identity**
- Fields can be initialized with values
  - They have **default values** if not initialized
  - **Zero** for numerics, **false** for booleans, and **null** for references

Notes:



Notes:

- Version: 20190401

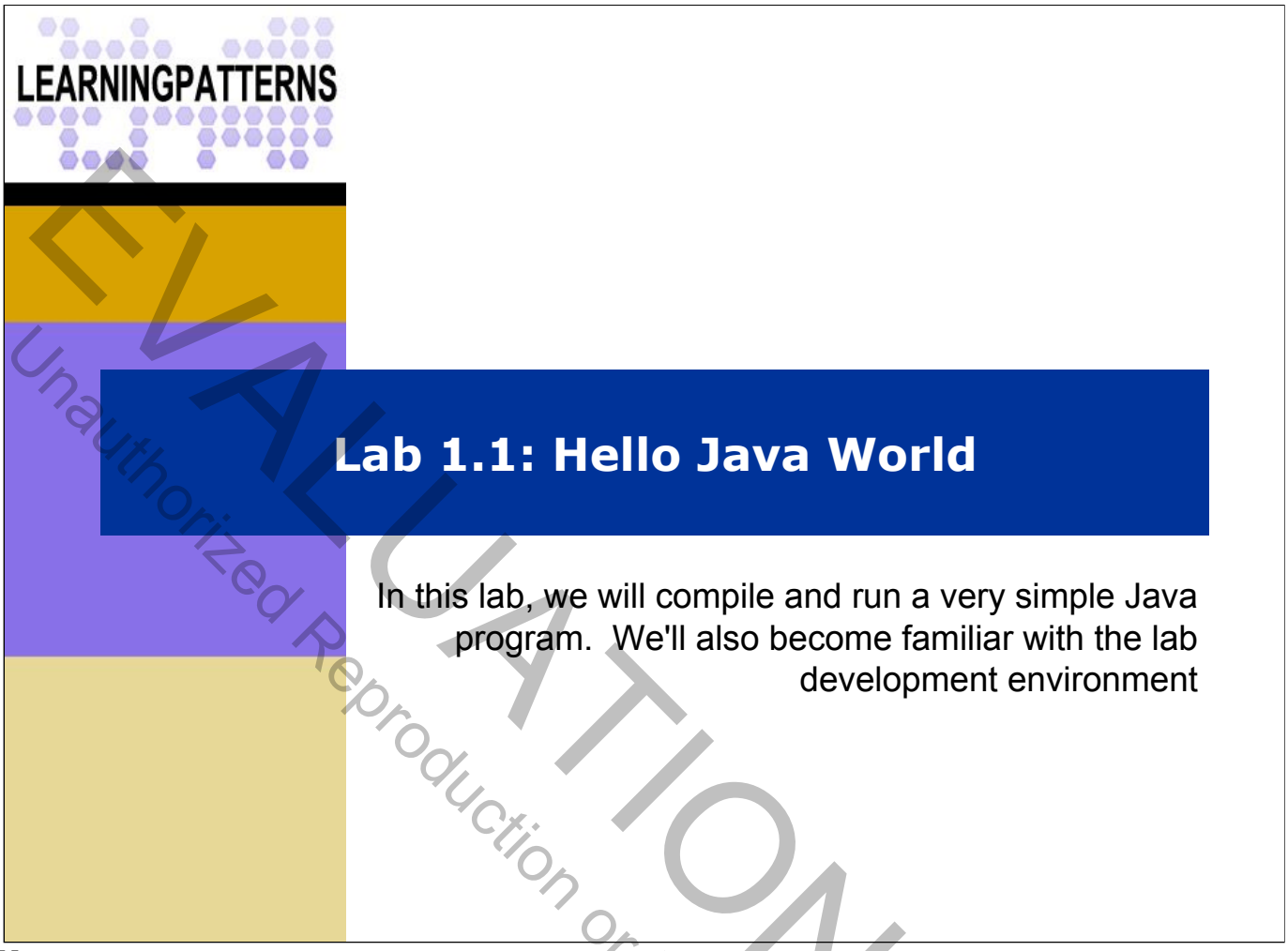
Release Level

Lab

- This manual contains instructions for creating and running the Fast Track to Java labs using the following Java platforms:
  - **Java 11**
    - Most labs can be done with Java 9 if you want
    - Certain labs/features require Java 10 or 11
  - **Eclipse Java EE** *
    - Eclipse 2018-12 (first release to fully support Java 11)
- Instructions are given for the Windows OS
  - However, if using Linux or Mac OS, you should easily be able to follow the instructions in your environment

Notes:

- The instructions for the labs are geared for the Eclipse Java EE version.
  - It can work with most versions of Eclipse, as long as they support Java 11.



Notes:

Lab Synopsis

- **Overview**: In this lab, we will set up our lab environment and compile and run a very simple Java program
  - We'll also set up and use the Eclipse development environment
    - Start it up, set up a workspace and project, and create/run a program
  - We will also work from the command line
    - Using **`javac`** (compiler) and **`java`** (JVM) directly to see how they work
  - Most later labs will only use the **Eclipse** IDE
- **Builds on previous labs**: None
- **Approximate Time**: 40-50 minutes

Lab 1.1: Hello Java World 4

Notes:

- Eclipse gives you a lot of help in writing and running Java programs.
  - So we use it :-)
  - It's also a good idea to know that Java programs don't require Eclipse to run.
  - So we do it without Eclipse also.

Information Content and Task Content

Lab

- In labs, information only content is presented in the normal way
  - Like these bullets at the top of the page
- Tasks students need to do are in a box like that below
  - So you can see them clearly

Tasks to Perform

- Note the different look of this box as compared to that above
  - All labs will use this format
- Make sure that you have **Java installed**
  - Open a command prompt, and execute the below

```
> java -version
> java version "11.0.1" 2018-10-16 LTS
```

  - You'll need at least Java 9, and preferably Java 11
  - If the command is not found, see the notes

Lab 1.1: Hello Java World 5

Notes:

- If the java command is not found, then it's likely that Java is not installed, or your path is set up incorrectly.
  - Check both of these with your instructor, as needed.
  - In general, on Windows machines Java is installed in a folder like "C:\Program Files\Java\jdk-11.0.1"
    - Depending on the Java version you have, this may change.
  - On Linux, the install folder would be different - check with your instructor.
  - Likewise, on Mac OS, there is a totally different setup, and if Java is installed it should work.

- If Java is not installed, you'll need to install it - It can be downloaded from:
  - www.oracle.com/technetwork/java/javase/downloads/index.html

Extract the Lab Setup Zip File

Lab

- You need the course setup zip file for the labs *
  - It has a name like: ***LabSetup_FTJ11_20190401.zip***
- The base lab folder is: ***C:\StudentWork\IntroJava***
  - Created when you extract the Setup zip - referred to as ***IntroJava***
  - It includes a directory structure and lab starter files
- Lab folders are under: *C:\StudentWork\IntroJava**workspace***
  - We refer to this as ***workspace*** or ***<workspace>***
- Instructions assume that this zip file is extracted to C:\
  - If you extracted elsewhere, then adjust accordingly

Tasks to Perform

- Unzip the lab setup file to **C:**
  - This creates the directory structure
  - Populates lab folders with any starter files that are needed

Lab 1.1: Hello Java World 6

Notes:

- The setup files may be on your workstation.
  - They may also be given by your instructor.

The Eclipse Platform

Lab

- **Eclipse** (**www.eclipse.org**): Open source platform for building integrated development environments (IDEs)
  - Used mainly for **Java development** - can be extended via plugins and is used in other areas (e.g. C# programming)
- Originally developed by IBM
  - Released into open source
  - IBM's RAD environment is built on top of Eclipse
- Eclipse products have two fundamental layers
  - The **Workspace** – files, packages, projects, resource connections, configuration properties
  - The **Workbench** – editors, views, and perspectives
- We will set up the workspace and workbench, then describe it in more detail at the end of the lab

Notes:

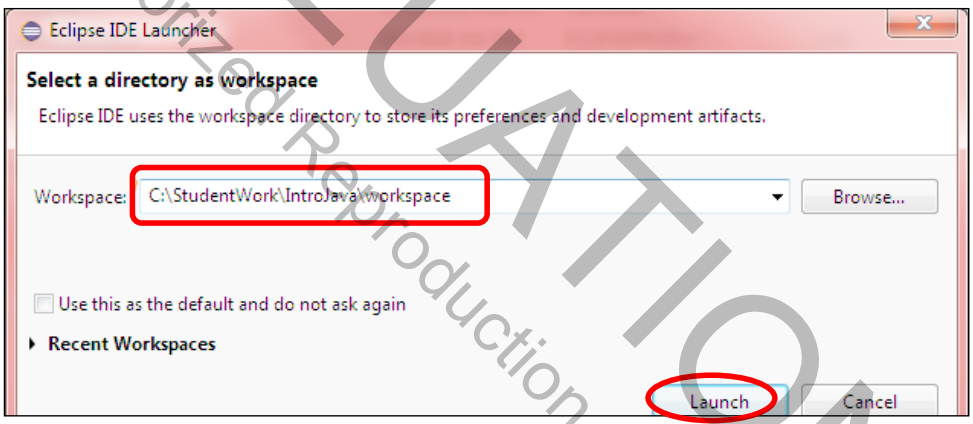
- The Workbench sits on top of the Workspace and provides visual artifacts that allow you to access and manipulate various aspects of the underlying Workspace resources.

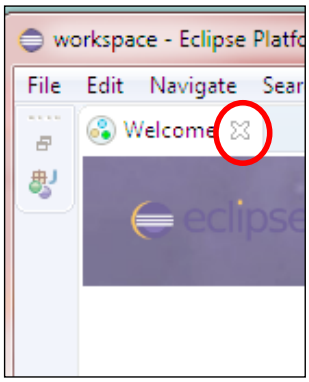
Getting Started With Eclipse

Lab

Tasks to Perform

- Make sure **Eclipse is installed** - likely in C:\eclipse (Windows)
- **Launch Eclipse**: Go to **c:\eclipse** and run **eclipse.exe**
  - A dialog box should appear prompting for a workspace location
  - Set Workspace to **C:\StudentWork\IntroJava\workspace**
  - Click **Launch**
  - **Close** the Welcome screen: Click the X on its tab (below right)





Lab 1.1: Hello Java World 8

Notes:

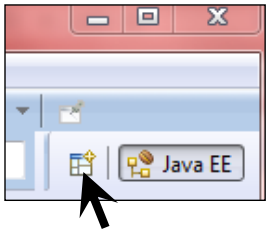
- If Eclipse was installed elsewhere, adjust the paths to the Eclipse executable accordingly.
- You can also put a shortcut on your desktop to start Eclipse.
- If you need to download Eclipse, go to **http://www.eclipse.org/downloads.**
  - Make sure you download a recent enough edition to support the Java version you're using.
    - For Java 11, that is Eclipse version 2018-12
  - In the **Package Solutions** section, click on the link for the **Eclipse IDE for Java EE Developers.**
  - Save the zip file to your computer, and unzip it (easiest location to unzip it to is C:\, but another location is fine as long as you can get to it to run the eclipse.exe executable).

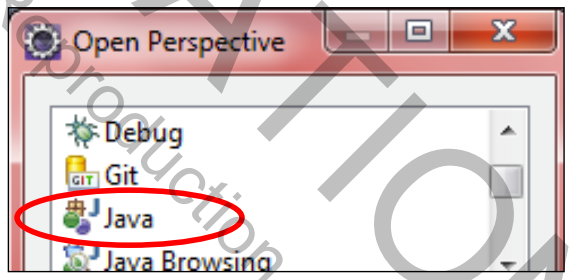
Workbench and Java Perspective

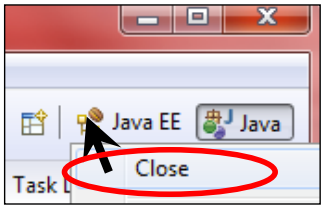
Lab

Tasks to Perform

- **Open a Java Perspective**: If needed - you may already be there (1)
  - Click the Perspective icon at the top right of the Workbench
  - Select Java (as shown below)
    - Most versions open in the Java EE or Resource Perspective
  - See notes on Perspective Icons and text labels
- Close the other perspective (Java EE illustrated below) by right clicking its icon, and selecting close (as shown below right)







Lab 1.1: Hello Java World 9

Notes:

(1) Different versions of the Eclipse platform have opened in different perspectives when they start up.

- Depending on what version you have, you may be in a Java perspective, a Java EE perspective, Resource perspective, or some other perspective.
- We want to be in the Java perspective, so if you are not in it, switch to it as described in the slide.

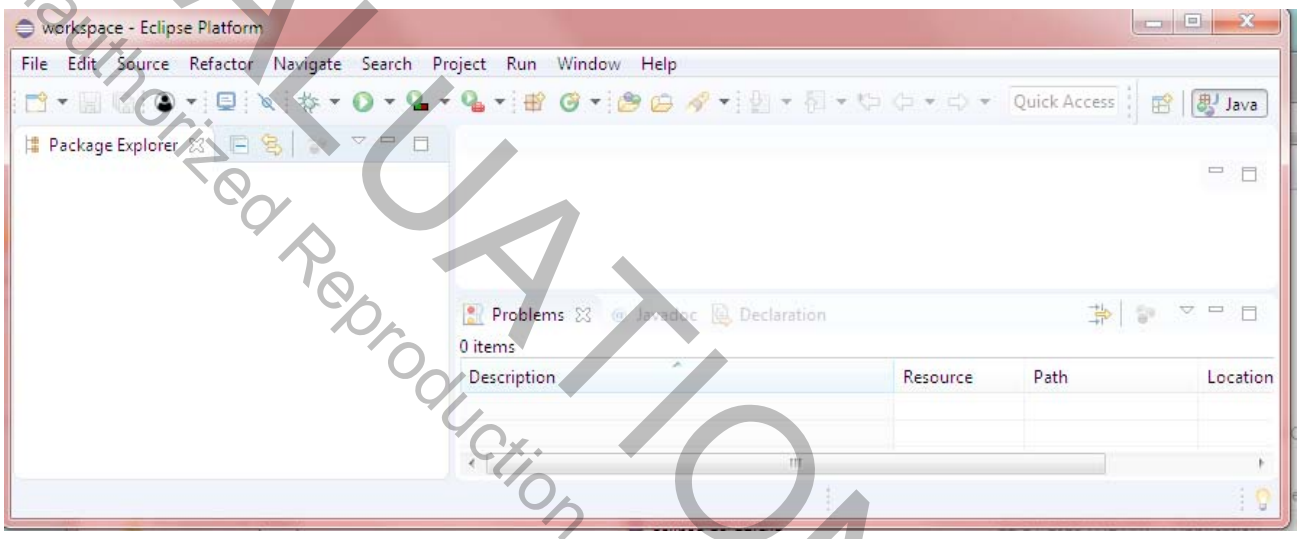
- Current Eclipse releases do not default to showing the perspective text in the perspective switcher.
  - If you want these, right click on the perspective icons, select **Show Text** - see examples below.

Unclutter the Workbench

Lab

Tasks to Perform

- Let's unclutter the Java Perspective by closing some views
  - Close the **Task List** and **Outline** views (click on the X)
  - Open the Navigator View (**Window | Show View | Navigator**)
  - You can save this as the default if you want (see note)



Lab 1.1: Hello Java World 10

Notes:

- To save the perspective as the default Java perspective go to the following.
  - **Window | Save Perspective As | Java**

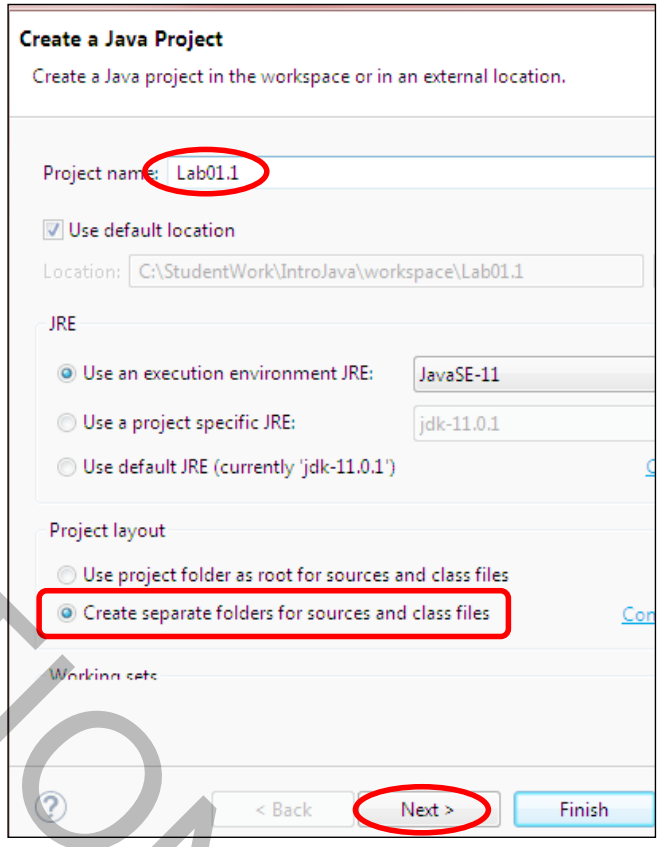
- You can reset the perspective to it's defaults via the following.
  - **Window | Reset Perspective**

Java Project

Lab

Tasks to Perform

- Create a new Java project
  - **File | New | Java Project**
- Fill in the name as **Lab01.1**
  - Make sure "**Create separate folders for source and class files**" is selected (it's the default)
  - Click **Next**
- There are many ways to create a project (see notes)
  - When you call the project Lab01.1, by default it will be stored in ***workspace\Lab01.1***



Lab 1.1: Hello Java World 11

Notes:

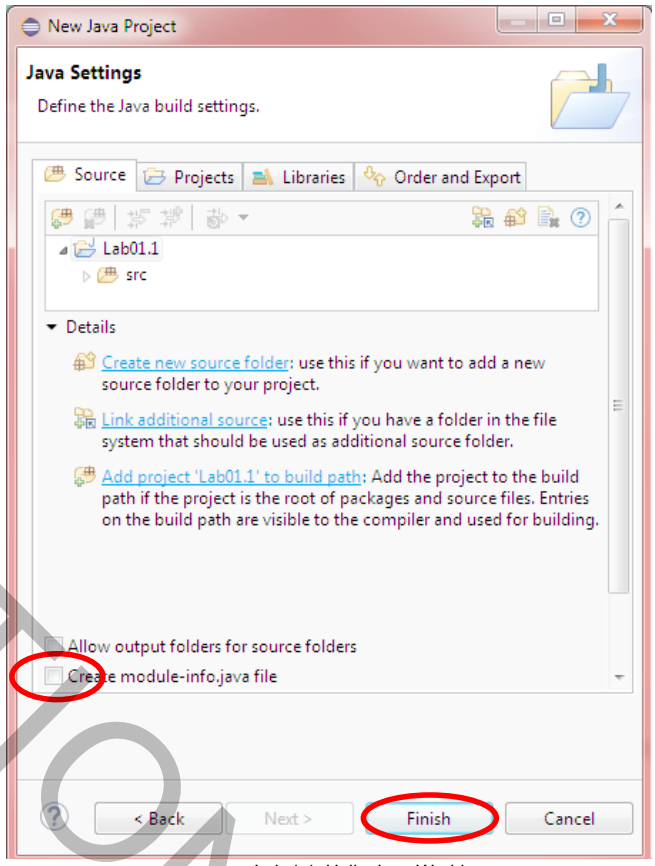
- There are multiple ways to create a new project:
  - **File | New | Project | Java | Java Project**
  - Click on the "**New**" wizard icon in left side of the toolbar.
  - Right click in the Package Explorer View, select **New ...**
- For almost all projects, it's usually better to create separate folders for source and class files.
  - We've done that here, and Eclipse puts the .java files in the Lab01.1\src directory, and the .class files in the Lab01.1\bin directory when we do this.
  - This is the default structure for Java projects in Eclipse.

Finish Java Project

Lab

Tasks to Perform

- In the next dialog
  - **Uncheck** *Create module-info.java file* (more on this later)
  - The remaining defaults are fine
  - We'll talk more about some of these later
  - Click **Finish**



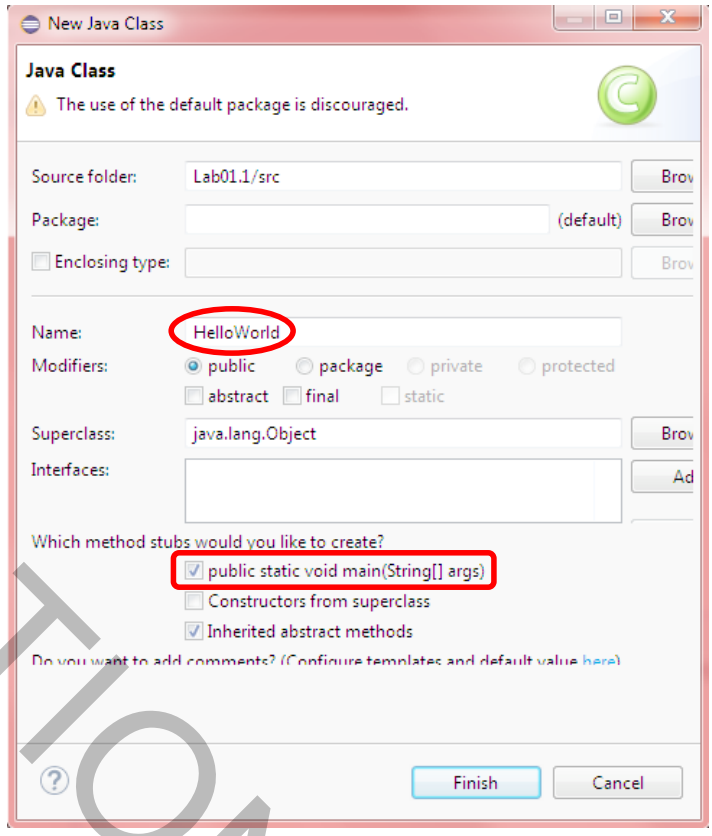
Notes:

Your First Application

Lab

Tasks to Perform

- Create a new Java class within the project *
  - **File | New | Class**
- Call the class **HelloWorld**
  - Generate a main method by checking the box shown
  - Ignore the warning about the default package (more later)
- Click **Finish**
  - *HelloWorld.java* will open in the editor
  - Add Hello World code as seen in the session slides
    - See notes for example



Lab 1.1: Hello Java World 13

Notes:

- There are multiple ways to get to the new class wizard, for example:
  - **File | New ...**
  - Right Click in a View → New ...
  - New Toolbar Icon → New …
- Don't worry about the warning that "The use of the default package is discouraged."
  - We'll learn about and use packages later.
  - The default package will work fine for our labs until then.
- The `HelloWorld` code is given below.
  - Note that the class will be placed in a file *HelloWorld.java*. This is required.
  - Write the code exactly as shown below. Remember that JAVA IS CASE SENSITIVE!!!

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World");
   }
}
```

Compilation and the Problems View

Lab

Tasks to Perform

- The file should compile automatically when you save it
  - Resolve any warnings or errors the compiler adds to the Problems view (see next slide)
- Add an error in your code and save the file
  - You should see the error in the Problems view
- The **Problems view** gets populated by problems Eclipse detects in your code
  - A compiler may add errors and warnings that need resolution
  - Eclipse may add it's own warnings (1)
  - **Double click** on a problem in Problems view to jump to the code associated with that problem

Lab 1.1: Hello Java World 14

Notes:

- By default classes get compiled when you save the source file.
  - This setting can be changed in the Workbench Preferences window: **Window | Preferences | General | Workspace** by Checking/Unchecking the **Build Automatically** selection.
  - It's usually best to leave the setting to build automatically.

(1) Eclipse does a more thorough checking of code than the Java compiler.

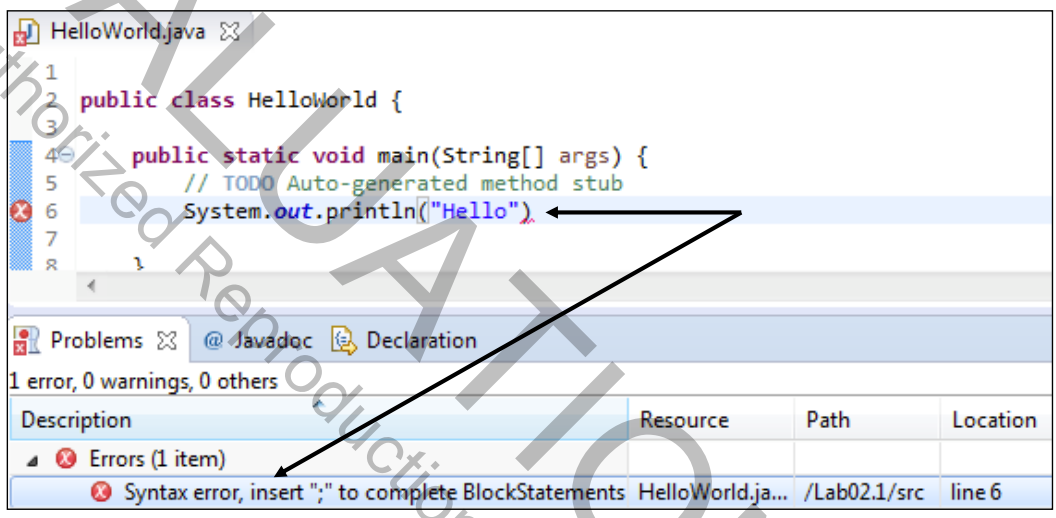
- Based on the experience of where common errors may arise.
- You can customize these warnings via **Window | Preferences | Java | Compiler | Errors/Warnings.**

Seeing a Problem

Lab

Tasks to Perform

- Note the missing semicolon at the end of line 6
  - Note the error message in the problem view
  - Note: Line numbers can be turned on/off via the menu selection: ***Window | Preferences | General | Editors | Text Editors***



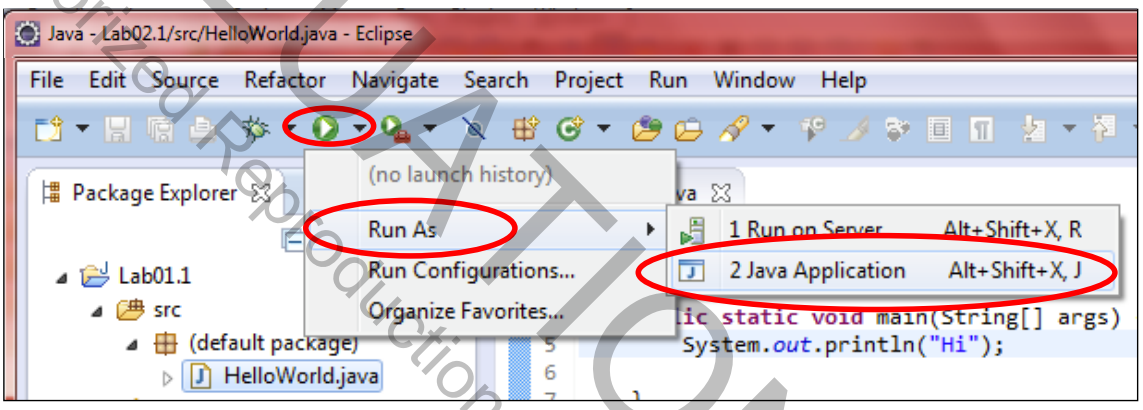
Notes:

Testing your Application

Lab

Tasks to Perform

- After a clean build (error-free, but not necessarily warning-free), test the application as follows
  - Select *HelloWorld.java* in the **Package Explorer** view
  - Click the run button arrow on the task bar *
  - Choose **Run As | Java Application** from the menu that appears



Lab 1.1: Hello Java World 16

Notes:

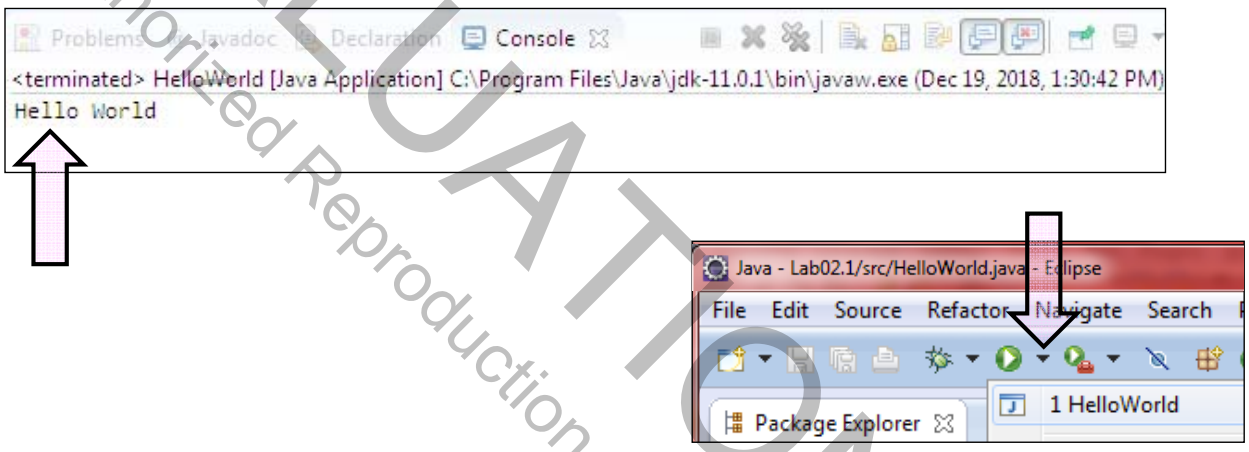
- To run the application you can also select **Run** from the menu.
- You can also right click on the *HelloWorld.java* file and select **Run As.**

Viewing Results

Lab

Tasks to Perform

- You'll see results in the console view as shown below
  - If necessary, open the Console (**Window | Show View | Console**)
- To run again, you can press the **Run Icon arrow** as shown at bottom
  - This brings up a list of previously run programs that you can pick from
  - Just select the `HelloWorld` program



Notes:

- When you select the Run icon, it shows a list of the existing launch configurations.
  - The configurations are named, and will appear in the run drop downs when you click on it.
  - See the next slide for information on launch configurations.

Using the JDK Tools Directly

Lab

Tasks to Perform

- We'll use **javac** (compiler) and **java** (Virtual Machine) directly
  - To understand them better, and see how to work without an IDE
  - The environment should already be set up to use these tools
  - If any of the commands aren't found, you'll need to check your path (1)
- Open a command prompt in the Lab01.1 directory
  - Delete the existing *.class* file (see at bottom)
  - Invoke the compiler to compile *HelloWorld.java*
    - javac expects files files for compiling to include the *.java* extension
    - **-d** specifies the output folder. When compiling finishes, you should have a file *bin\HelloWorld.class*
  - We illustrate below using Windows OS commands/paths
    - If using *nix / Mac OS, then adjust accordingly (in all labs)

```
> del bin\*.class
> javac -d bin src\*.java
```

Lab 1.1: Hello Java World 18

Notes:

(1) The path should already be set up on your system to use Java from the command line. If it's not, then you need to adjust your path accordingly, and we give you some guidelines below.

- On Windows, this would be done through the Control Panel by setting an environment variable JAVA_HOME to the appropriate location (e.g. "C:\Program Files\Java\jdk-11.0.1")
  - You would then set your Path environment variable to include %JAVA_HOME%\bin; at the start.
- On Linux, you'd likely set this in your .bashrc file, or something similar.
- On Mac OS, you'll likely use .bashrc also.
  - But you may not need to do this. Your system may already have java and javac on the path (generally from executables or links in /usr/bin)

- If you have compiler errors, check the following:
  - The word `String` is capitalized, since `String` is a class in the API.
  - The word `System` is capitalized.
  - The semicolon is in the proper place at the end of any statement.
  - The braces are paired properly.
  - The main method description is coded exactly as shown in the slides.

Running HelloWorld

Lab

Tasks to Perform

- You'll use the **java** executable (the JVM / Java Virtual Machine)
  - Run from the same command prompt where you compiled
  - You'll execute **HelloWorld**'s `main()` method
- Change to the bin folder, then run by typing the below:

```
> cd bin
> java HelloWorld
Hello World
```

  - Note: The JVM does **not** expect you to include the .class extension when passing the name of the program to run
- You should see the output "Hello World" in your command prompt
  - Congrats! Your first Java program done in an IDE and with the JDK

Notes:

Summary

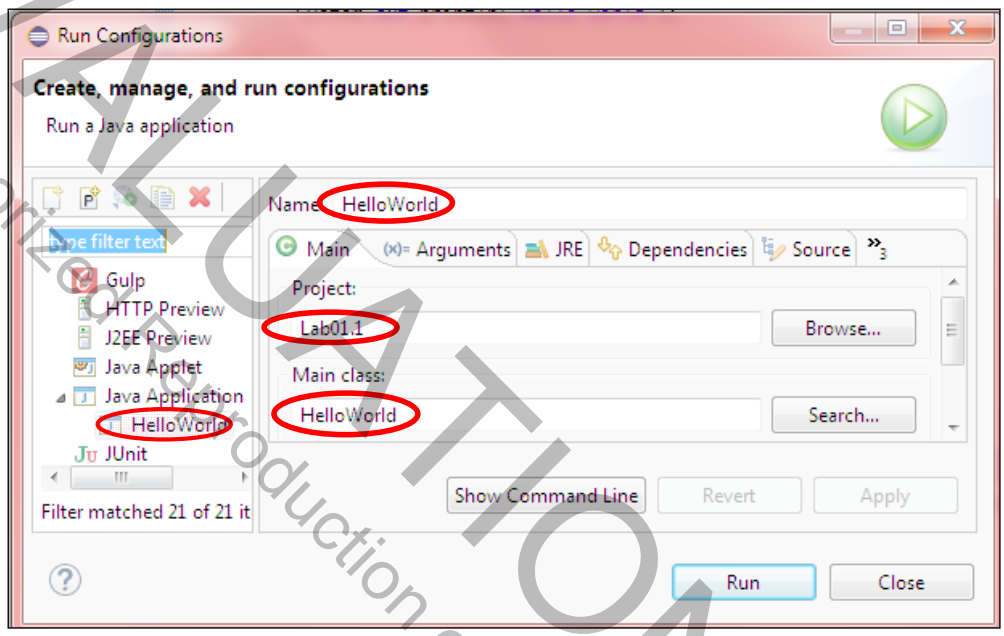
- You've written a (very small) Java program
  - You've used Eclipse to compile and run it
    - Easy!
  - You've also used the JDK directly
    - Informative and useful at times
- The rest of this lab provides useful general information about Eclipse
  - If not familiar with Eclipse, then review the slides briefly (5 minutes)
  - We'll use some of the capabilities mentioned in later labs

Notes:

Launch Configurations

Lab

- Eclipse creates a **Launch configuration** to run a program
- Lets you customize the execution of an application
  - Review it by going to the **Run icon**, selecting **Run Configurations …**, and selecting **HelloWorld** from the next dialog



Notes:

- As you see, the launch configuration has all the information necessary to run the program.
  - Including the project, and the class that has the main method in it.

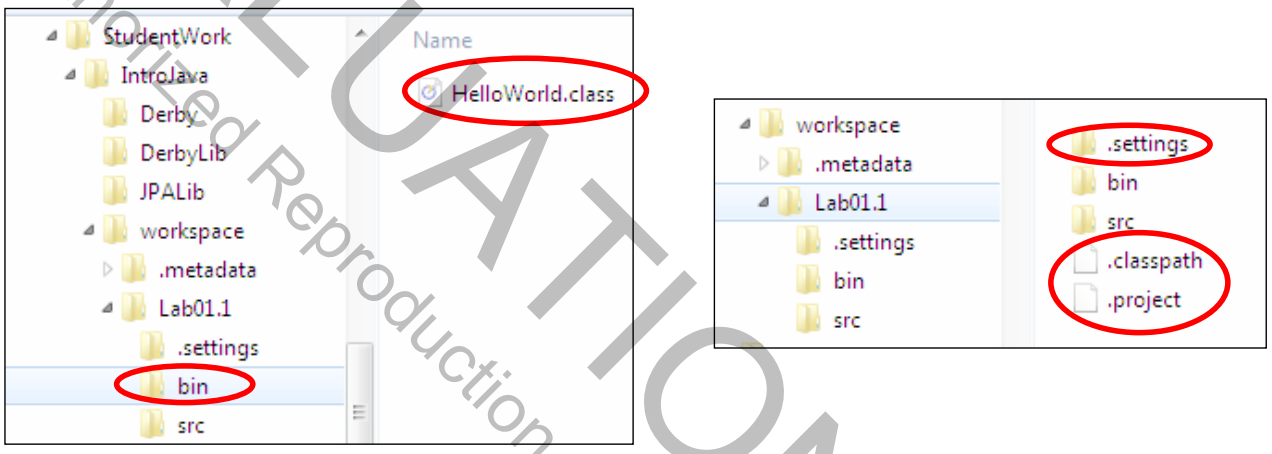
- We'll need to work with the launch configuration in the future when we use program arguments that are passed to the main method.

The Project Directory

Lab

Tasks to Perform

- Using your standard environments file browser, view the *workspace\Lab01.1* - src and bin folders
  - You'll see *HelloWorld.java* and *HelloWorld.class* files (in src and bin)
  - In the project root, there is a.*settings* folder, *.project* and *.classpath* files
    - These may not be displayed, depending on your system's settings
  - These are used by Eclipse to maintain the project



Lab 1.1: Hello Java World 22

Notes:

- In general, many environments do not display files whose name begins with a period.
  - These are called hidden files.
  - If you want, you can turn on the viewing of these files through your file viewer or OS preferences.
  - You can also open up the **Navigator** view within Eclipse and see these files.

Important Notes for Using Eclipse

Lab

- Any lab that has a **new lab directory** will require you to create a **new Eclipse project**
  - Sometimes several labs are done one directory, in which case you will use the same project for all of them
- If you copy/paste files, paste them **within Eclipse**
  - Generally you'll be copying files from the one of the *LabSetup* subfolders
  - Then pasting them into the project you are working on
  - We'll give detailed instructions the first time we do this
- For anyone not familiar with Eclipse, the next few slides give a (very) brief overview of how Eclipse is structured
  - There is **nothing you need to do** in those slides – they are for information purposes only

Notes:

- For any lab that has you copy files from the setup to your working directory you should paste them directly into the Eclipse IDE.
  - Otherwise, if you paste using your file system tools, Eclipse will require you to refresh the project to detect the new files.

The Eclipse Paradigm

Lab

- Eclipse products have two fundamental layers
  - The **Workspace** – files, packages, projects, resource connections, configuration properties
  - The **Workbench** – editors, views, and perspectives
- The Workbench sits on top of the Workspace
  - Provides views to access/manipulate resources
  - **Editor** – A component that allows a developer to interact with and modify the contents of a file.
  - **View** – A component that exposes meta-data about the currently selected resource.
  - **Perspective** – A grouping of related editors and views that are relevant to a particular task and/or role.
- You can have multiple perspectives open to provide access to different aspects of the underlying resources

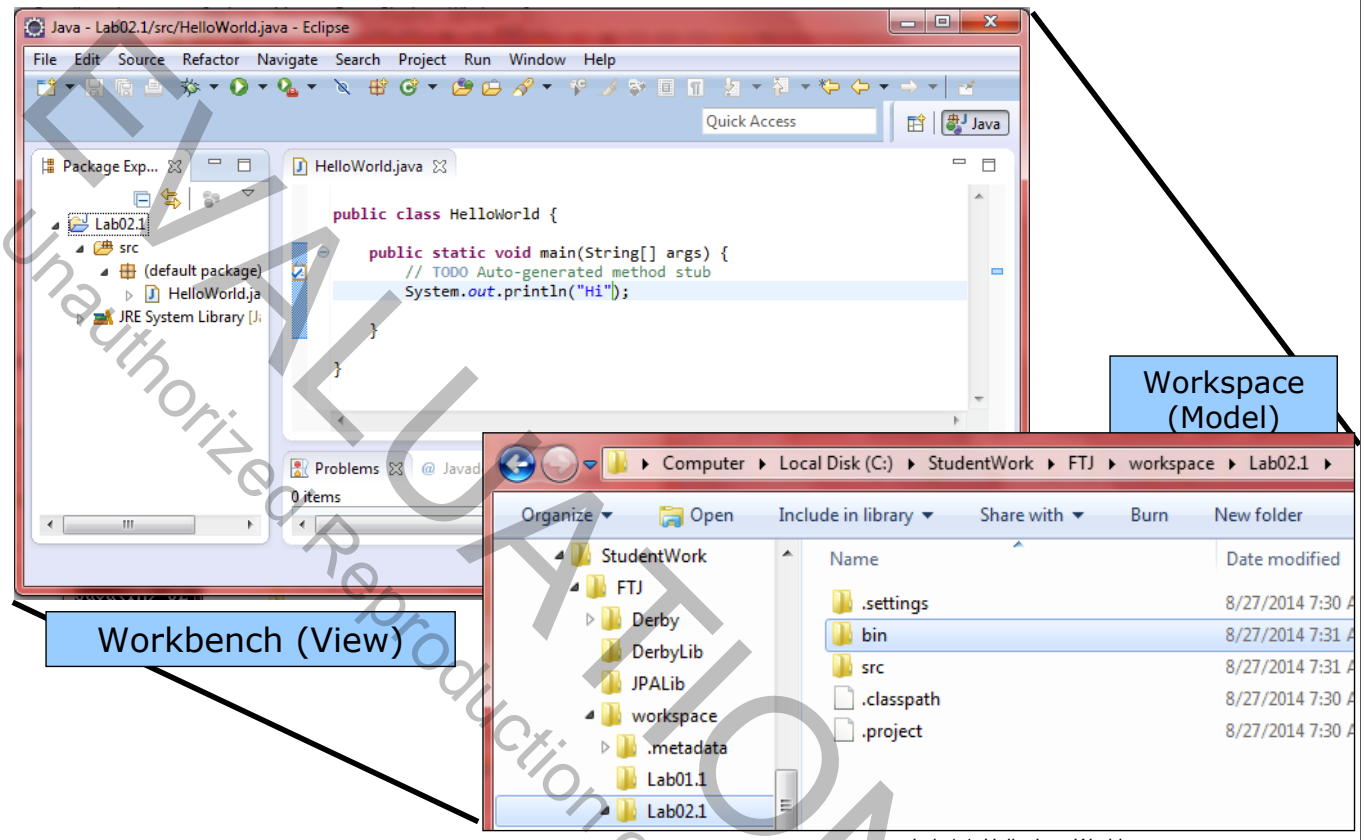
Lab 1.1: Hello Java World 24

Notes:

- The physical directory structure for the Workspace can be found in the "workspace" folder.
  - The default location is under the Eclipse home directory.
  - We specified a different workspace location when we started Eclipse.
- It is even possible to set up multiple workspaces (with corresponding Workbenchs). Simply create a folder to house the additional workspace, and write a script that uses the Eclipse executable file and supplies the 'data' argument with the location of the workspace directory to load:
  - eclipse.exe -data other_workspace_folder
- A Perspective is basically a collection of views that are focused on a given task.
  - They provide different tools to work with the resources.
  - For example, the debugging perspective has views open for debugging, such as: Active Threads, Variables, Breakpoints, etc.
  - There are perspectives for Java development (Java Perspective), and so on.
  - What perspectives are available depends on what version of Eclipse you have, and what plugins you have installed.

Workbench and Workspace

Lab



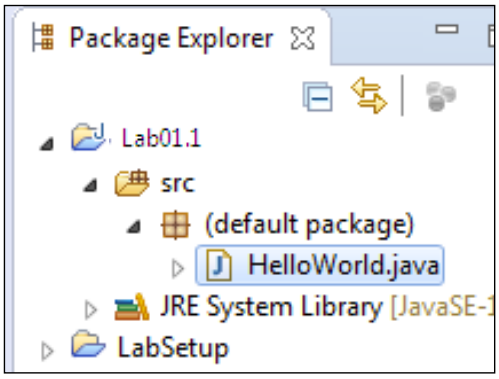
Notes:

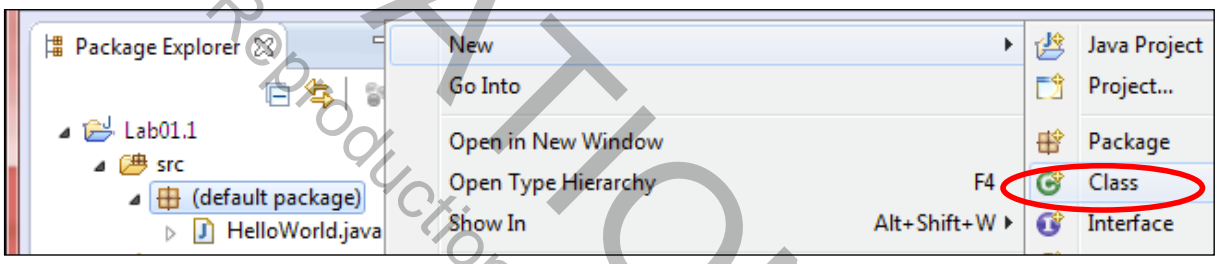
- We use the terms Model and View here in the same sense as when talking about Model-View-Controller (MVC).
  - The Model is the actual data (the files).
  - The View is the Eclipse Workbench.

Package Explorer View

Lab

- Shown by default in the Java perspective
  - A Java-specific view of resources
  - Shows Java element hierarchy of projects
  - Easy to use with Java resources
  - e.g., to create a new class, right click on the package [1] you want and select **New | Class**





Notes:

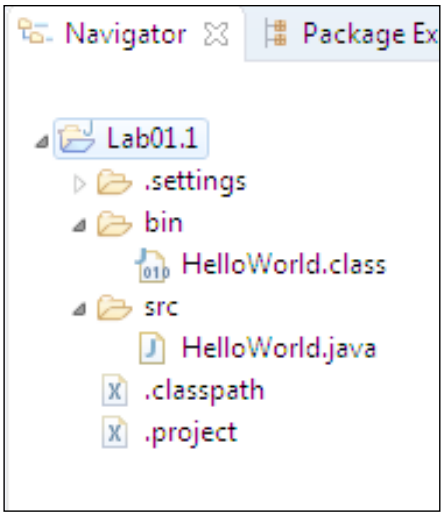
[1] We will talk about packages later in the course.

- For now, just right click on the "default package" icon if you want to test this.

Navigator View

Lab

- Similar to file system view
  - There are three kinds of resources described below
- **Files**
  - Correspond to files on the file system
- **Folders**
  - Like folders on the file system
- **Projects**
  - Used to organize all your resources and for version control.
  - Creating a new project assigns a physical location for it on the file system.
  - A third-party SCM (Source Control Manager) may be used to properly share project files amongst developers.



Lab 1.1: Hello Java World 27

Notes:

- The Navigator view is useful if you want to see what files actually exist on the file system.
  - It provides a file-based organization of the various resources. There are other views that give application-based, and project-based organization of data.

Editors

Lab

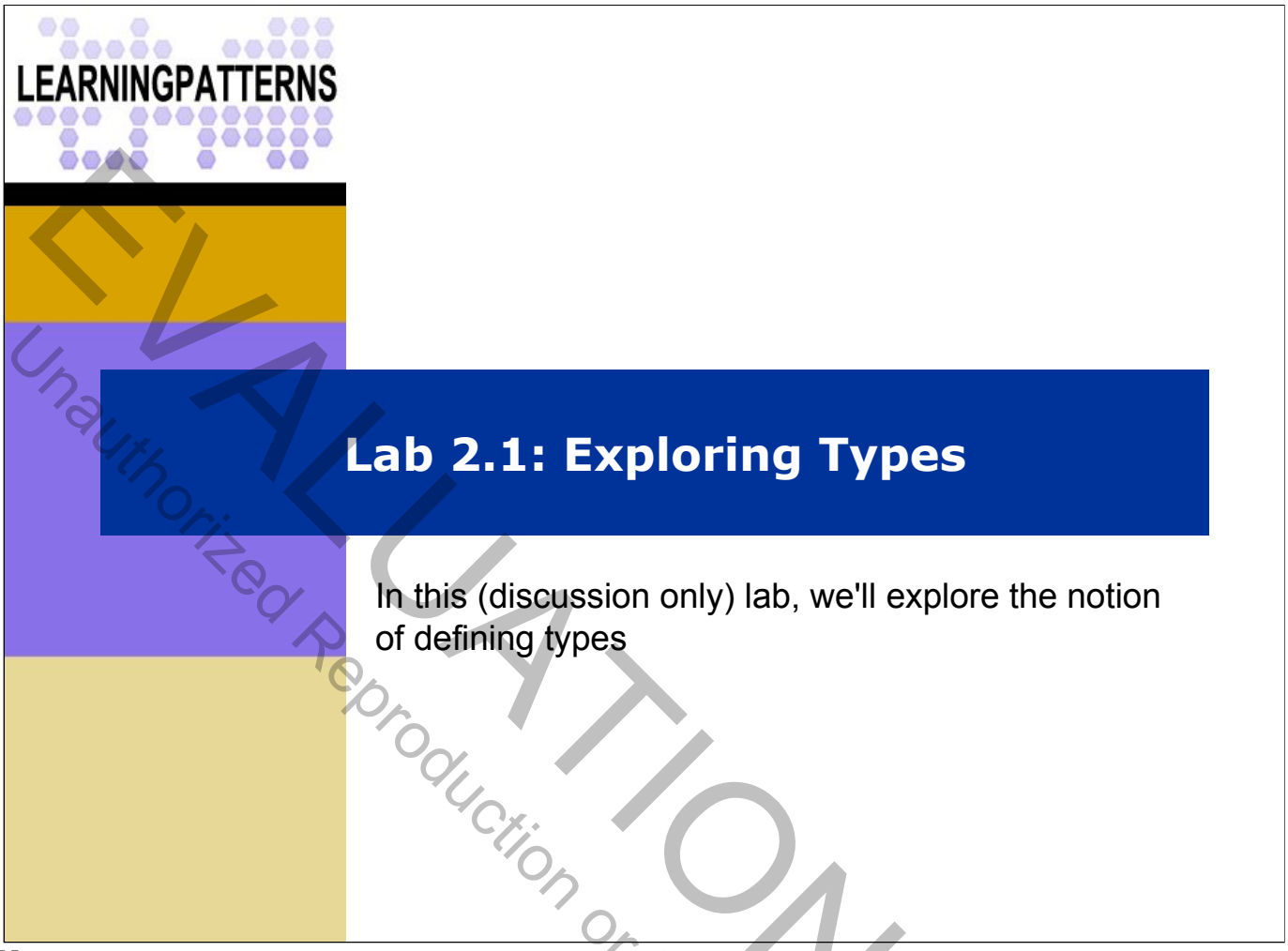
- There is a customized source editor (like this one for a .java file) for all character files. (.java, .jsp, .html, etc.)

HelloWorld.java

```
1
2  public class HelloWorld {
3
4      public static void main(String[] args) {
5          // TODO Auto-generated method stub
6          System.out.println("Hello");
7
8      }
9
10 }
11
```

Lab 1.1: Hello Java World 28

Notes:



Notes:

Lab Synopsis

- **Overview**: In this (discussion only) lab, we'll explore the notion of defining types much like the ones you might define in a program
  - We'll define a type's characteristics (its name, properties, and behavior)
- **Builds on previous labs**: None
- **Approximate Time**: 15-20 minutes

Lab 2.1: Exploring Types 30

Notes:

Create Your Own Types

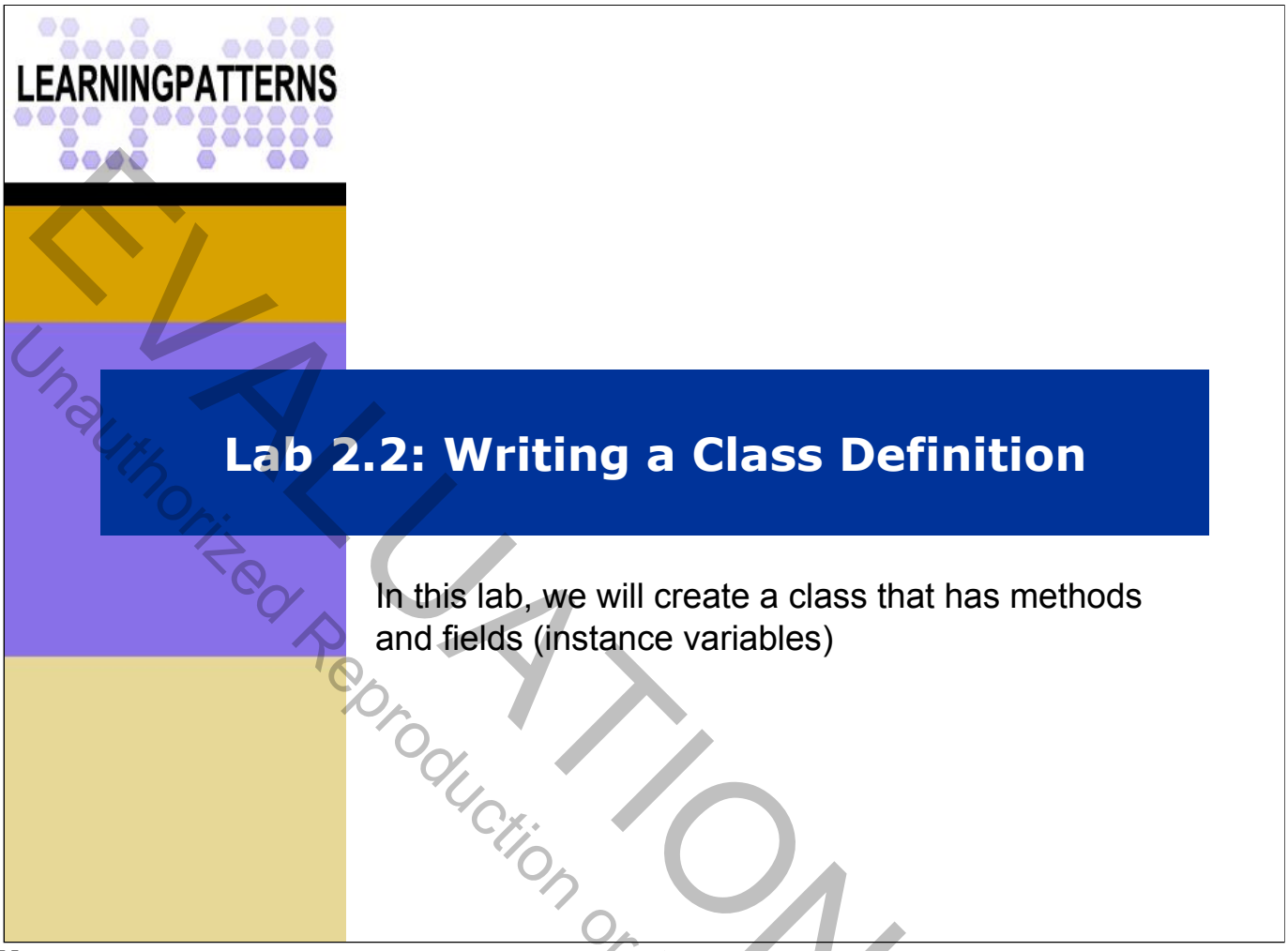
Lab

Tasks to Perform

- Split up into pairs or small groups, and work together on creating several types, including their properties and behavior
  - You can use types from projects that you are working on in your company, or make up a new problem domain
  - Discuss each type a little – does everyone agree that this is a type? Discuss its properties and behavior
  - Discuss the names of your types – names are important
    - They convey information to users of the type
  - Have one member from each of a few different teams go to the front of the room and present their types

Lab 2.1: Exploring Types 31

Notes:



Notes:

Lab Synopsis

- **Overview**: In this lab, we will create a class that has methods and fields (instance variables)
  - We will also write and run a test program that creates instances of the class and works with those instances
  - NOTE: we will continue to modify and add to this class as we learn more of Java's capabilities
- **Builds on previous labs**: None
  - The new lab folder and project is **Lab02.2**
- **Approximate Time**: 25-35 minutes

Notes:

Lab Preparation

Lab

- The root lab folder is **workspace\Lab02.2**
  - This is a new lab directory
- We'll create a **`Television`** class
  - It will have brand and volume fields to work with the concepts we just learned (but **no `main()` method**)
- We'll also create the **`TelevisionTest`** class
  - It **will** have a `main()` method where you create `Television` instances and call business methods on them
  - Note "property" is often used as a name for the data fields of an object

Tasks to Perform

- Close all files and projects you have open (1)
- Create a new Java project called **Lab02.2** in your workspace
  - Again, **uncheck** *Create module-info.java* file, click **Finish**
  - See the earlier lab's instructions if you need details

Lab 2.2: Writing a Class Definition 34

Notes:

- For Eclipse, and most likely for any other project based environments you might be using, you'll need to do the same configuration you did for the first project you created.
  - For example, adding in user libraries, or modifying the classpath by adding in jars or libraries.
  - For this simple lab, there aren't any library or classpath issues to deal with.

(1) To close a project, just right click on it, and select **Close Project.**

Creating a Class

Lab

Tasks to Perform

- Create a new class called `Television`
  - It will be saved in a file ***Television.java***
  - Use the Eclipse wizard similarly to the earlier labs
  - In your `Television` class, declare two fields:

```
String brand;    // brand of Television
int volume;      // current volume
```

- Write two business methods `turnOn()` and `turnOff()`
  - Note the following of Java naming conventions (1)
  - The methods should take no arguments, and return nothing
    - What will the return type be declared as?
  - Print a message to the console in the methods (`System.out.println()`)
    - The messages should include the value of at least one of the fields

Lab 2.2: Writing a Class Definition 35

Notes:

- **WARNING: Java is case-sensitive!**
  - If you create a class named `Television`, then it must be saved in a file named *Television.java*.
  - If you save it in a file named *television.java*, you will end up having problems later.
  - Eclipse does this automatically for you, but it's important to know this
- **NOTE**: We are creating classes in the default package right now.
  - This isn't recommended practice, and we'll move them into a package later.
- **NOTE**: we normally make all our data private, for design reasons. We will discuss that soon.
  - It's good practice to use a comment on each line above or next to your declaration of a variable, describing what the variable is.

(1) Method names should follow the same conventions as variables, i.e., first "word" in lower case, each subsequent "word" capitalized, e.g., `turnOff`, `turnOn`, etc.

- Your `System.out.println` can look something like this:

```
System.out.println("Turning on your TV with volume " + volume);
```

Writing a Test Class

Lab

Tasks to Perform

- Create a new class, **`TelevisionTest`**
  - Include a **`main()`** method in `TelevisionTest`
  - Remember - you can check off the box to create a main method in the Eclipse new class wizard
- In `TelevisionTest.main()` instantiate two Televisions
  - You'll want `Television` references also, of course

```
Television tv1 = (what goes here?)
```

  - Call **`turnOn()`** and **`turnOff()`** on each instance
  - This should result in your messages being output to the console

Lab 2.2: Writing a Class Definition 36

Notes:

- Again, `TelevisionTest` will be saved in a file named ***TelevisionTest.java.***

Running the Program

Lab

Tasks to Perform

- **Run the program** as you did in the earlier lab (see notes)
  - You have two classes now, `Television` and `TelevisionTest`
  - Which class should be your class to run the program?
  - **Hint**: Use the one with the `main()` method in it
  - Is the output what you expect?
- Next, add initializers in `Television` for brand and volume

```
String brand = "Toshiba";
int volume = 1;
```

  - Run the program again. What do you see now?
- The next slide shows some handy editor features for reference

Lab 2.2: Writing a Class Definition 37

Notes:

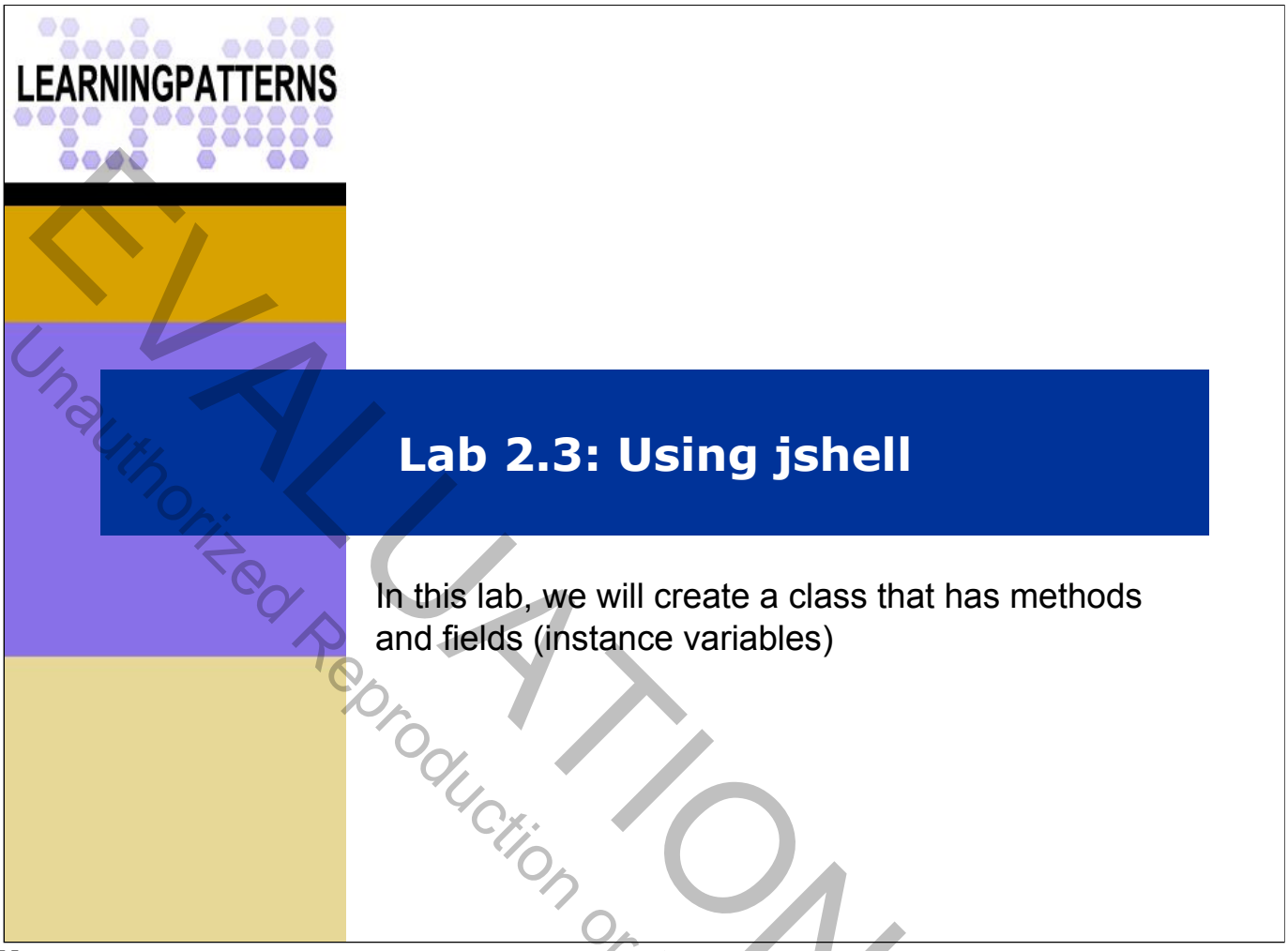
- Select your .java file in the Package Explorer view.
  - Click the run button arrow on the task bar.
  - Choose **Run As | Java Application** from the menu that appears.
- What happens if you try to run a class with no main method?
  - Try it and see.

Handy Editor Features

- Some of the more common features include
  - Undo (C-Z), Redo (C-Y), Cut (C-X), Copy (C-C), Paste (C-V), Select All (C-A]), etc.
  - Content Assistance (*C-spacebar*)
  - Code Formatting (*Select code*, *right-click, Source* | *Format*)
  - Jump to a Line (*C-L*)
  - Find/Replace (*C-F*)
  - Add bookmark (*Edit menu or right-click on source line*)
  - Add task or breakpoint (*right-click in left margin*)
  - Searching: (*C-H*)
  - Note that C- is an abbreviation for pressing the Ctrl key along with the associated key

Lab 2.2: Writing a Class Definition 38

Notes:



Notes:

Lab Synopsis

- **Overview**: In this lab, we will use `jshell`
  - jshell is Java's interactive tool for using Java - added in Java 9
  - Also called a Read-Eval-Print-Loop (or REPL)
    - Because it reads your input, evaluates it, prints the value, then loops to do it again
- **Builds on previous labs**: None
- **Approximate Time**: 10-20 minutes (Depending on optional part)

Notes:

Work with jshell

Lab

Tasks to Perform

- Open a command prompt, and start jshell
  - Once it starts up, review some of the help, e.g. the below
  - jshell>`/help intro`
  - jshell>`/help`
  - Execute some statements, e.g. the below
  - jshell>`int i = 5+3;`
  - jshell>`i`
  - jshell>`System.out.println("Hi");`
  - Try out some float arithmetic that is illegal - what do you see?
  - jshell>`float f = 55.5;`
  - Try anything else of interest for a few minutes (see next slide also)

Notes:

[Optional] Define Television in jshell

Lab

Tasks to Perform

- Optionally, define a `Television` class in jshell
  - Just give it a `String brand` as a field - ignore volume
  - Optionally, give it a method also (e.g. `turnOn()`)
  - Note that your class def can extend over multiple lines - jshell understands Java syntax
- Create instances of your `Television` class, as in your previous lab, e.g.

```
Television t1 = new Television();
```

  - Play with them - call methods on the instance
  - Try to access the brand value (e.g. `t1.brand`)
    - This should work - we'll talk more about this soon
- Exit jshell when done (`/exit`)

Lab 2.3: Using jshell 42

Notes:

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