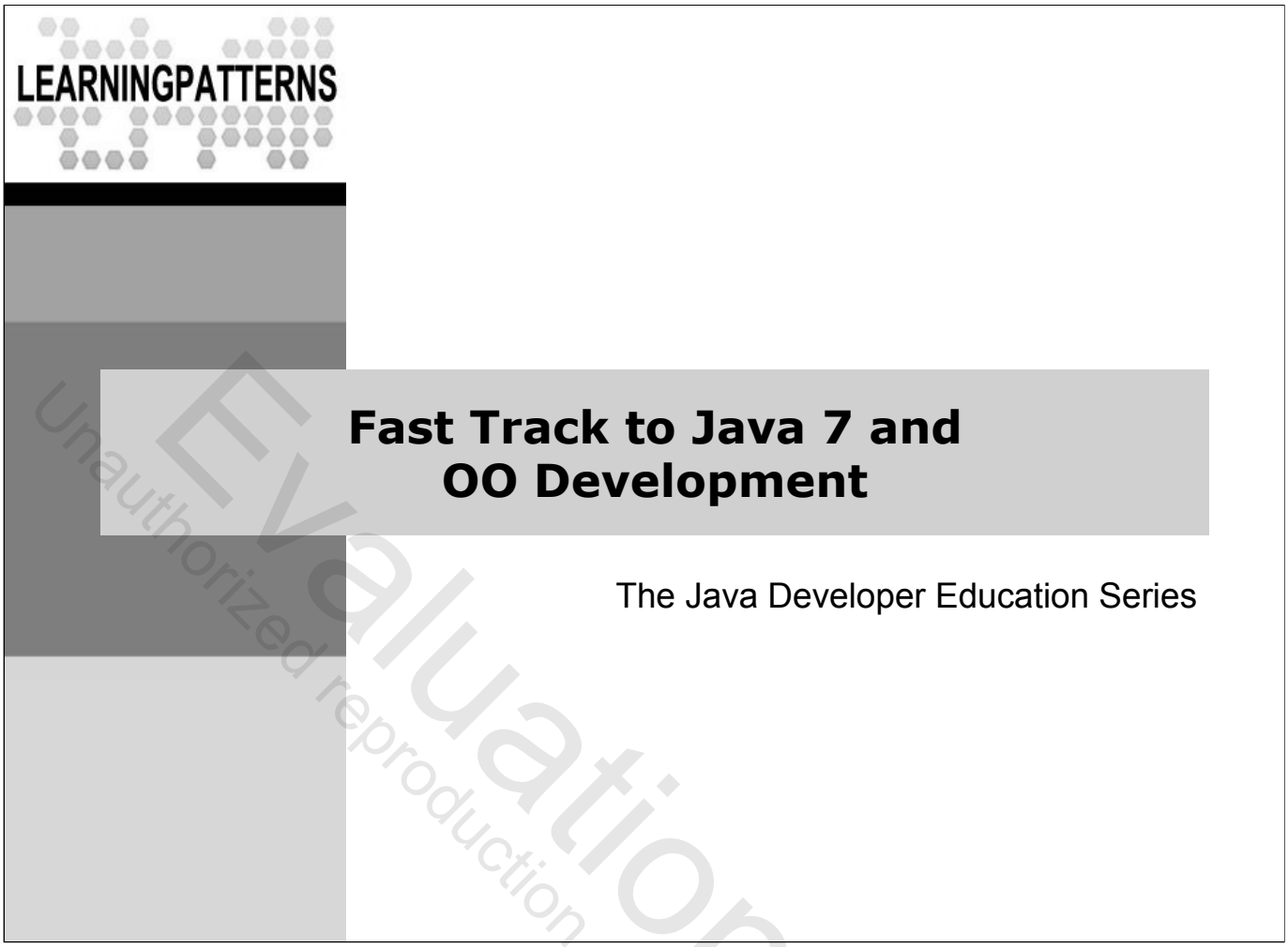
Table of Contents – Fast Track to Java 7



Notes:

Course Overview

- An introductory Java course that starts with basic principles
  - Provides a solid foundation in the concepts and practices for writing good object-oriented systems in Java
  - Provides knowledge needed to productively use core Java technology for programming
  - Including database access with JDBC/JPA
- Be prepared to work hard and learn a great deal!
- The course contains numerous hands-on labs
  - They exercise all the important concepts discussed
  - The lab solutions for the course are provided to you
- The course covers all core features of Java
  - It supports all recent versions of Java

Notes:

- The Java platform has evolved rapidly
  - However, many people are still using older versions
- The labs are designed to support those using Java 5 and any later versions

- Java 6 and 7 primarily introduced advanced features for running Java programs, and did not chance the API, though it did add to it
  - However, those features are mostly beyond the scope of this course
  - Accordingly, this course can be used in any of these environments

Course Objectives

- Learn Java's architecture and uses
- Understand Java language basics
- Compile and execute Java programs with development tools such as `javac` and `java`
- Learn object-oriented (OO) programming and the object model
  - Understand the differences between traditional programming and object-oriented programming
  - Understand important OO principles such as composition inheritance and polymorphism
- Use Java packages to organize code
- Understand interfaces, their importance, and their uses

Notes:

Course Objectives

- Learn (and practice!) Java naming conventions and good Java coding style
- Create well-structured Java programs
- Use core Java API class libraries
- Understand how exceptions are used for error handling
- Understand the basics of database access with JDBC and JPA
- Learn the basics of the Collections Framework
- See some of the new/advanced Java language features
- Understand and use basic I/O streams (optional)

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 student 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
- Lab slides have an icon like in the upper right corner of this slide
  - The end of a lab is marked with a stop like this one:

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*

  – We sometimes denote more info in the notes with a **star ***

  – Longer code examples appear in a separate code box - e.g.

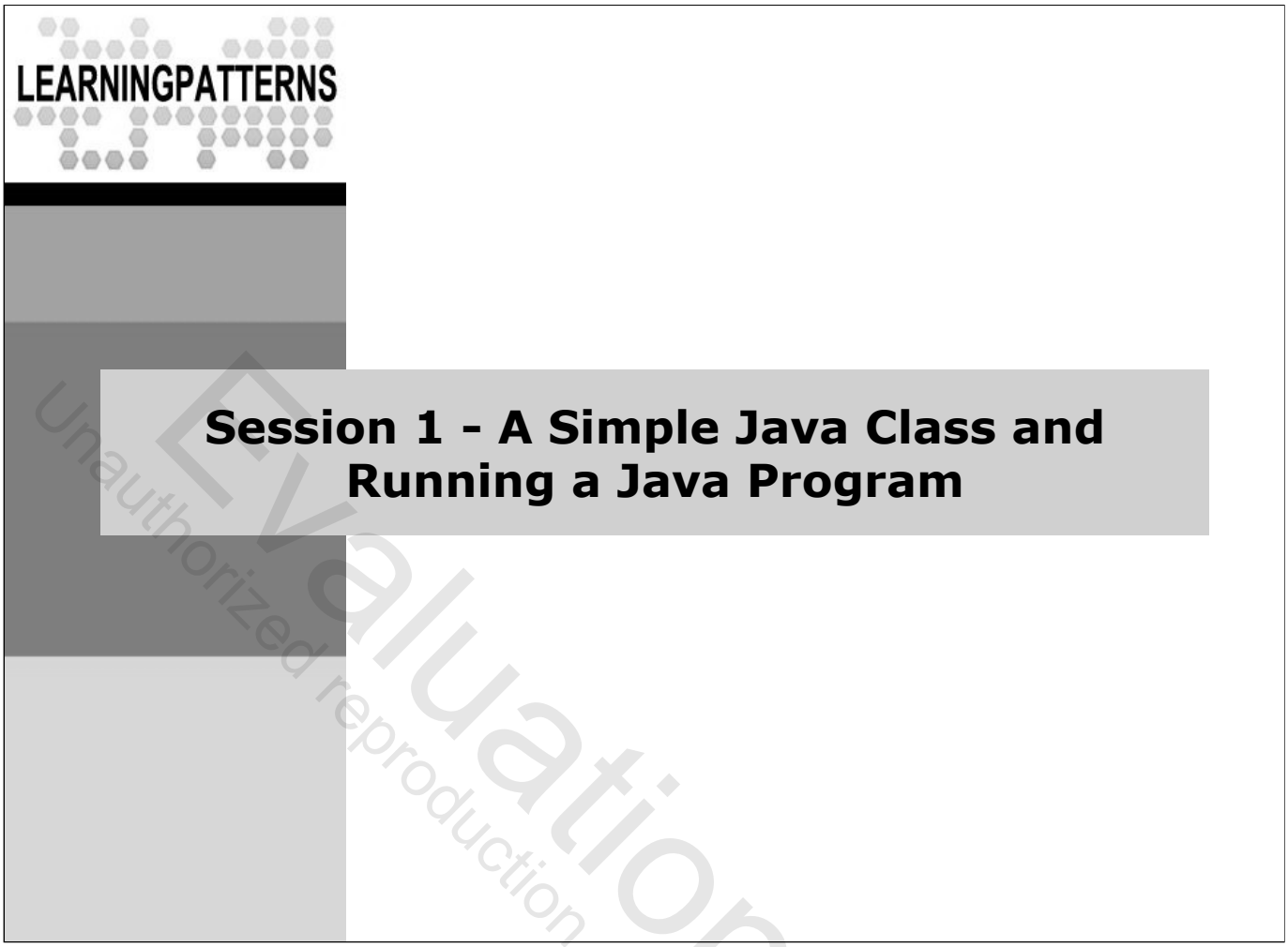
```
package com.javatunes.teach;
public class JavaInstructor implements Teacher {
   public void teach() {
      System.out.println("Java is way cool");
   }
}
```

Notes:

Course Outline

- Session 1: A Simple Java Program and the JDK
- Session 2: Java Overview
- Session 3: Class and Object Basics
- Session 4: More on Classes and Objects
- Session 5: Flow of Control
- Session 6: Strings and Arrays
- Session 7: Packages and Access Protection
- 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 simple Java program, and how it is compiled and run
- Set your computer up for Java development
  - Setting paths, environment variables, etc.
- Use a text editor to type in a simple Java class
- Use the Java Development Kit to compile and run the code

Notes:

A Simple Application Class

- To get started, we'll look at a very simple Java application, that displays the string "Hello World"
  - This is a simple non-graphic standalone application that runs on its own
- All programs in Java are just a **class** with a **`main`** method in it
  - The HelloWorld class needs to appear in a file **HelloWorld.java**
- The **`main`** method is a special method that is the starting point for every Java application
  - It must be declared as shown, and has to appear in a class
  - `System.out.println` how you print things to the console

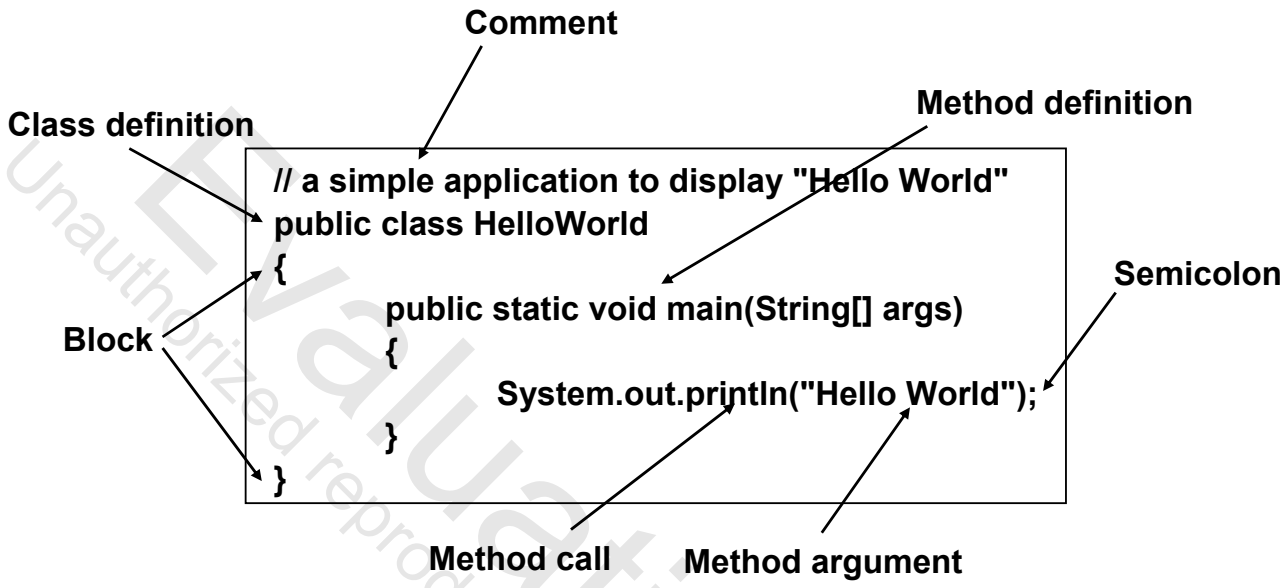
```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World");
   }
}
```

Notes:

- Java classes can be deployed into several runtime execution environments, including:
  - Standalone applications that are run explicitly, like our Hello World application.
  - Servlets, which are run in a Web server.
  - Applets, which are downloaded from a Web server and run in a Web browser.
- We will concentrate on applications in this course.
- `main` is the special method which is the starting point for every standalone Java application.
  - Applets and servlets work a little differently
- The `main` method has to appear inside a class.
  - `main` takes an 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.

The HelloWorld Program Broken Down

- We'll cover these details in more depth later



Notes:

Compiling HelloWorld

- To use a Java class, you have to **compile** the Java code

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

  - After compiling, a file called ***HelloWorld.class*** (containing Java **bytecode**) is created

- You use the Java Virtual Machine (**JVM**), to run a Java program
  - The JVM comes with Java as the **`java`** executable

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

  - This would produce the output "Hello World" printed in your command prompt window

Notes:

- HelloWorld.class is the bytecode for the program produced by the compiler
- We will talk more about all these details later in the course
- For now, we're giving you enough information to just run the program

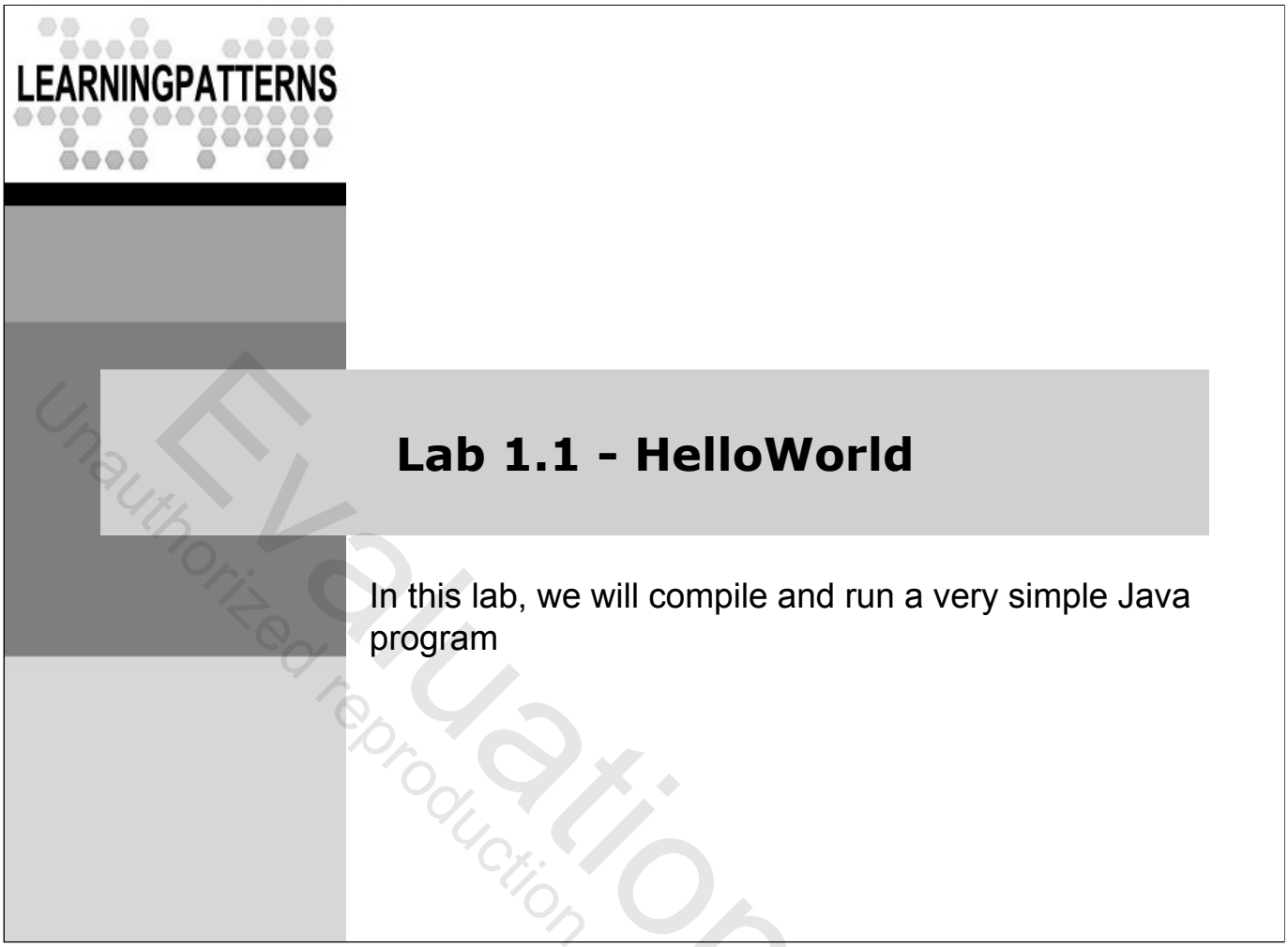
Note on Comments

- Java has 3 kinds of comments
  - Single line comment: Start with **//** …
  - Multiple lines using **/* … */**
  - Javadoc comments (see note) using **/** … */**

```
/**
 * 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");
    // we often comment out a line code this way
    // System.out.println("Bye");
  }
}
/* Note that this class still needs a lot of work.  We
   need to add all sorts of interesting things, and show
   how cool Java has become.  */
```


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
- A tool called **javadoc** reads these comments
  - Along with your source code
  - `javadoc` also understands a number of parameters that you can embed in these comments to enhance the generated documentation
  - We cover this tool in an appendix



Notes:

Session Review

1. Why must you set your path include the *<java>\bin* directory?

2. What is the purpose of the `main` method?

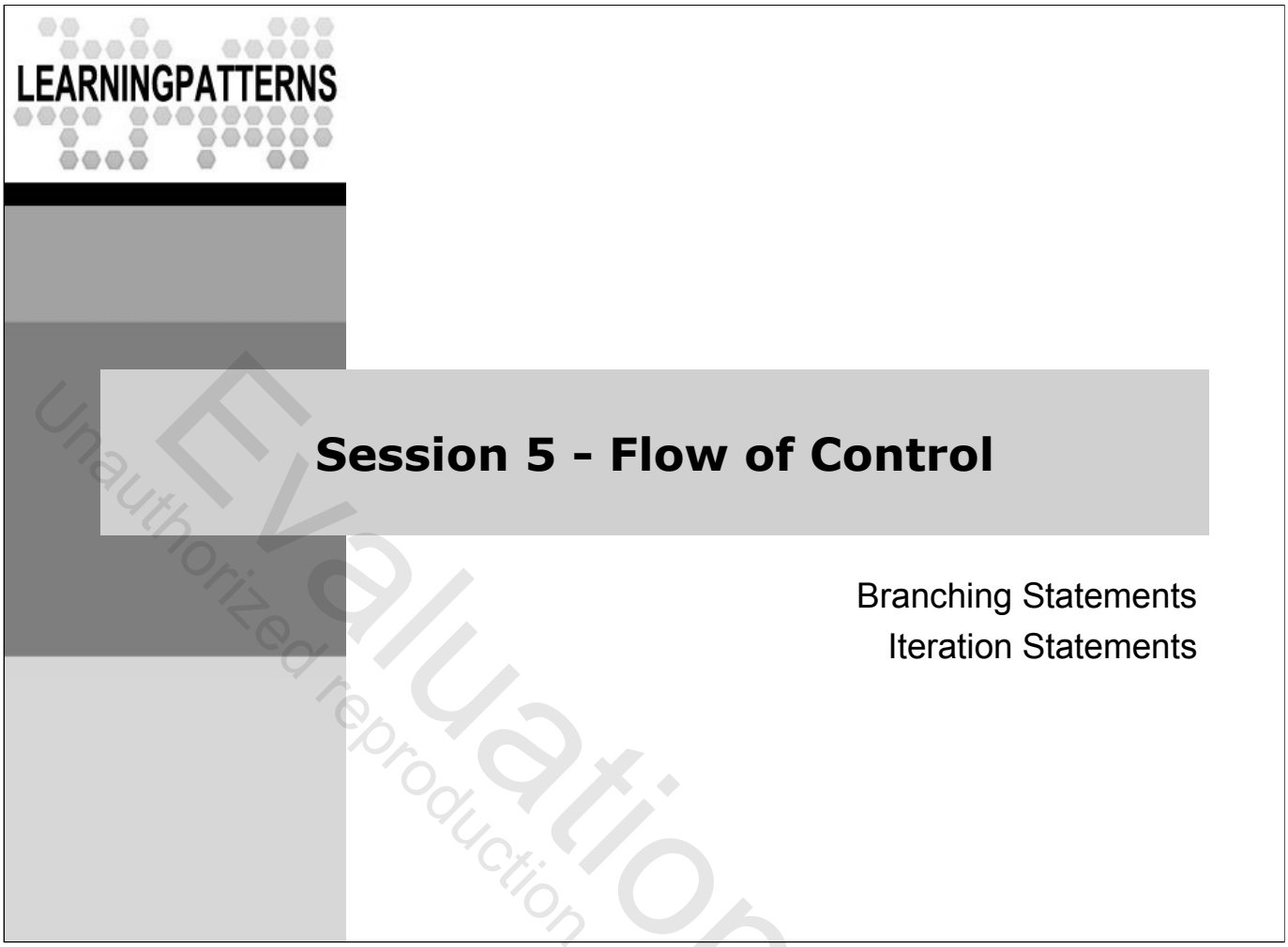
3. What is the signature of the `main` method?

4. How do you print something to standard output (the console)?

5. What tool do you use to compile Java source code?

6. What tool do you use to run compiled Java code?

Notes:

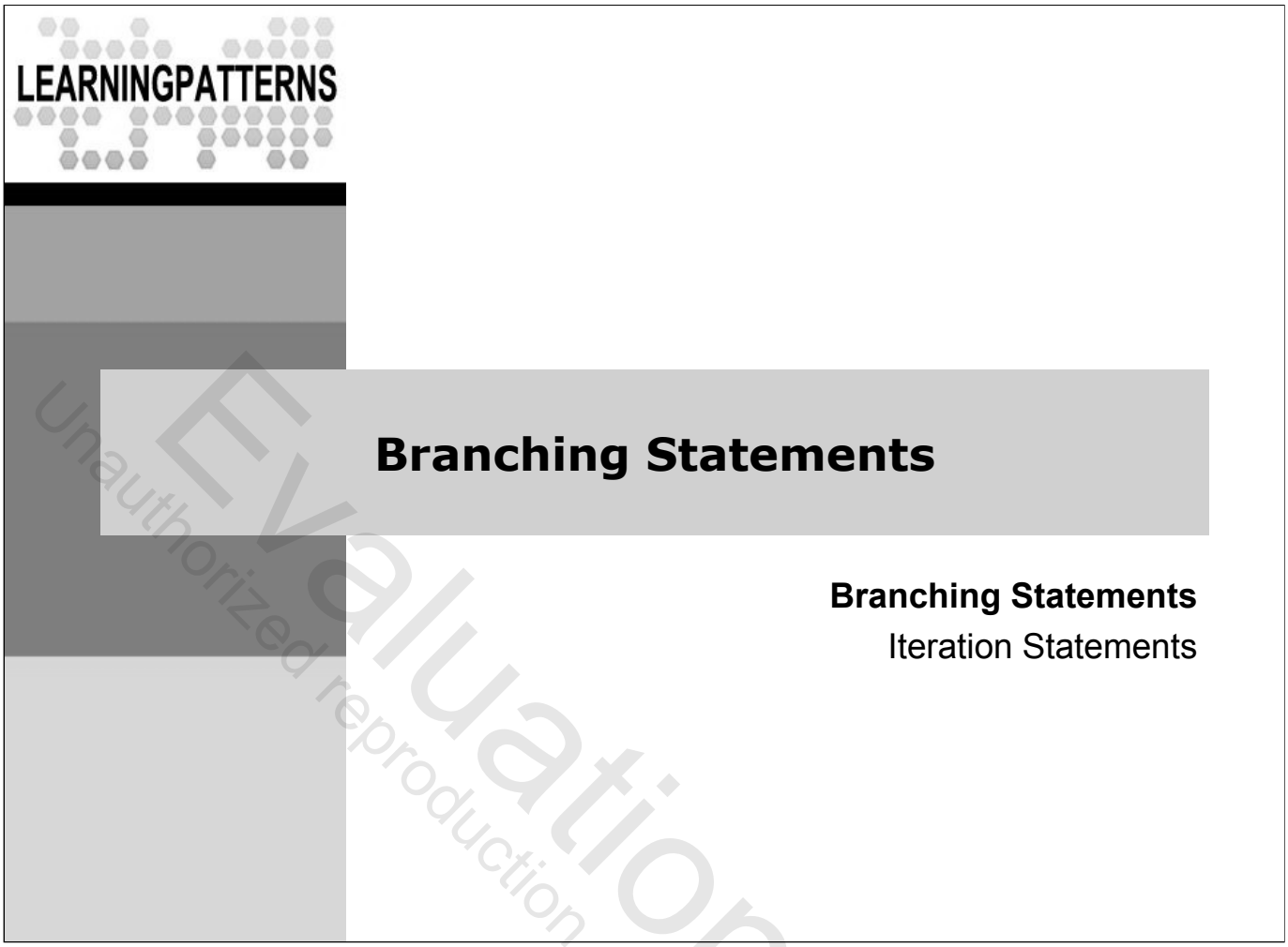


Notes:

Session Objectives

- Outline the comparison and boolean operators in Java
- Discuss branching statements and the operators used with them
  - `if`, `if-else`, `switch`
- Discuss iteration (looping) statements
  - `while`, `do-while`, `for`
  - `break`, `continue`
- Use flow of control logic to perform data validation in an object

Notes:



Notes:

Program Execution Sequence in Java

- Unless directed otherwise, statements in Java are executed in sequence
- Java has a number of statements to change the flow of control
  - Branching/selection statements choose one of several flows of control:
    `if`, **`if-else`**, and **`switch`**
  - Iteration statements specify looping
    `while`, **`do-while`**, and **`for`**
  - Jump statements transfer control unconditionally
    `break`, **`continue`**, **`return`**

Notes:

- All of these statements are modeled after their equivalent in the C programming language

The Comparison Operators

- Selection and iteration statements are based on the results of comparisons that return a `true` or `false` result
- The comparison operators compare numerical values and produce `boolean` results

Operator		Example
==	equal	`3 == 5 (== false)`
!=	not equal	`3 != 5 (== true)`
<	less than	`3 < 5 (== true)`
>	greater than	`3 > 5 (== false)`
<=	less than or equal	`5 <= 5 (== true)`
>=	greater than or equal	`5 >= 5 (== true)`

Notes:

The Logical Operators

- The logical operators compare `boolean` values and produce `boolean` results.
- In the table below, assume we have:

```
boolean t = true;
boolean f = false;
```

Operator	Meaning	Example
&	AND	`f & t` (== `false`, `t` evaluated)
&&	conditional AND	`f && t` (== `false`, `t` not evaluated)
|	OR	`t | f` (== `true`, `f` evaluated)
||	conditional OR	`t || f` (== `true`, `f` not evaluated)
^	exclusive OR	`t ^ t` (== `false`) `t ^ f` or `f ^ t` (== `true`)
!	NOT	`!t` (== `false`)
==	equal	`f == f` (== `true`)
!=	not equal	`t != t` (== `false`)

Notes:

if-else Statement

- The **if** / **if-else** statements control whether or not a statement is executed, based on the value of an expression
  - The else is optional
    - It's executed only if the expression evaluates to false

if (*Expression* **)** *Statement* **else** *Statement*

```
int i = 1;

if (i == 0) {
  System.out.println("i equals 0");
} else {
  System.out.println("i is not 0");
}
```

Notes:

- The parentheses are required around the expression
  - The expression must have a boolean type.
- The statement may be a block of statements, enclosed in { }.
- The expression is evaluated, and if it is true the statement is executed.
- RECOMMENDATION: always use the block form to avoid potential maintenance errors.
- You can chain them.

```
int k = 0;

if (k == 0)
{
  System.out.println("k equals 0");
}
else if (k == 1)
{
  System.out.println("k equals 1");
}
else
{
  System.out.println("k is neither 0 nor 1");
}
```

switch Statement

- The **switch** statement extends the `if` statement, to allow testing for more than one value
  - Can only switch on `byte`, `short`, `int`, `char`, and `enum`
  - Java 7+ allows switch on string values
  - `default` case is optional and gets control when no case matches
- Note: If the break statement is **not present**, **execution continues on** to the code in the next case

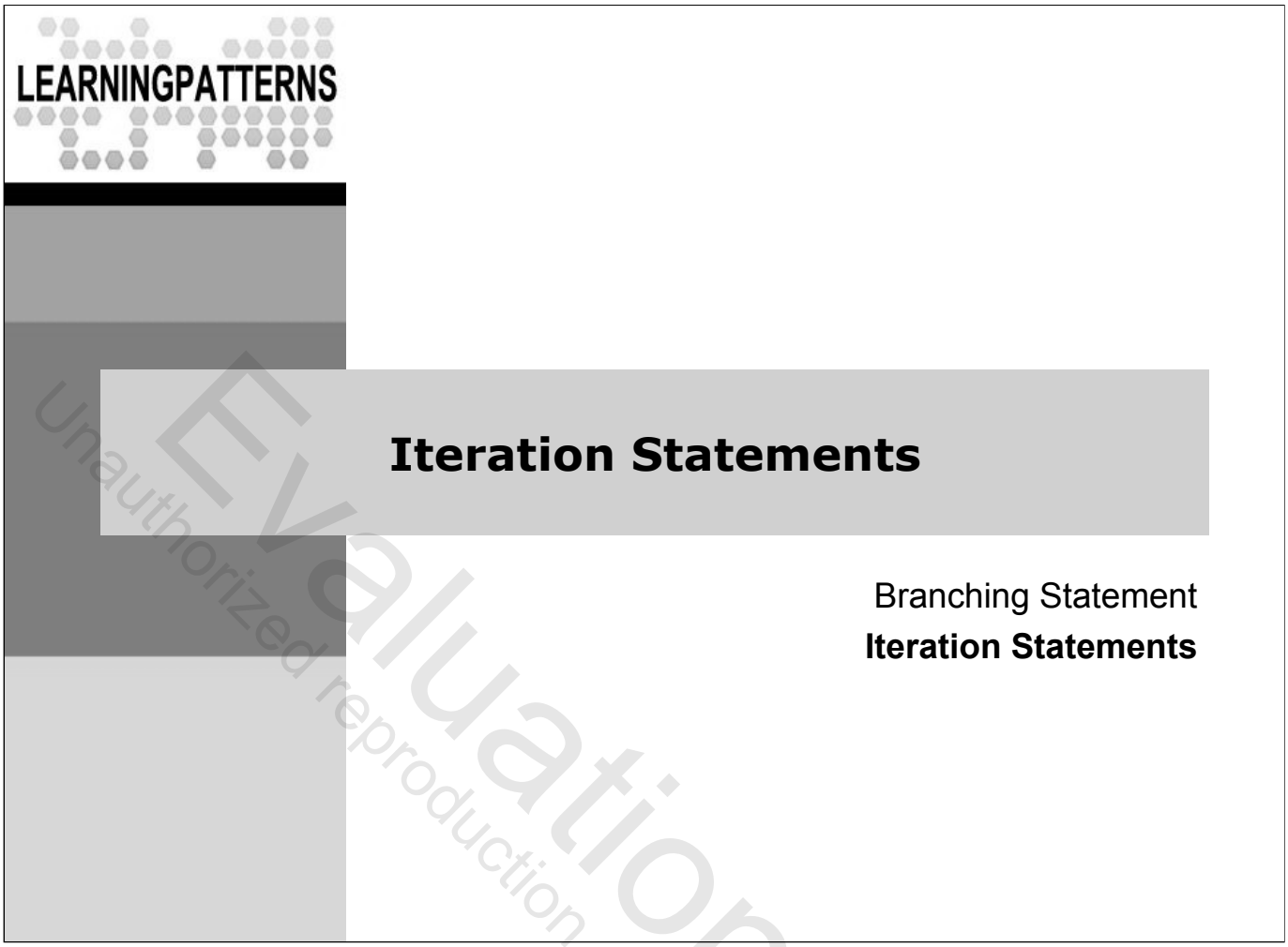
```
int i = 1; // This would usually be initialized elsewhere
switch ( i ) { // Execute a case based on value of i
  case 1:
    System.out.println("i is 1");
    break;
  case 2:
    System.out.println("i is 2");
    break;
  default:
    System.out.println("i is large");
    break;
}
```

Notes:

- The general form of the switch is:

```
switch ( variable ) {
  case value1:
    statement(s);
    break;
  case value2:
    statement(s);
    break;
  default:
    statement(s);
}
```

- A `switch` statement may execute one or more of several alternatives. When executed:
  - The expression in the `switch` is evaluated and compared to the `case` constants.
  - If one of the `case` constants is equal to the value of the expression, execution continues at that statement (i.e., a jump to that point in the code occurs).
  - If none of the `case` constants match, the statements after the `default` label, if present, are executed.



Notes:

while Statement

- The **while** statement creates a loop – it has the form:

while (*Expression* **)** *Statement*

```
int index = 10;

while (index > 0) // stop looping when index reaches 0
{
  // do some work
  index = index - 1;
}
```

```
ResultSet rs = ...; // This is a JDBC ResultSet object
while (rs.next())  // rs.next() eventually returns false
{
  // process next row in result set
}
```

Notes:

- The expression must have a `boolean` type.
- The expression is evaluated repeatedly, and as long as it evaluates to `true`, the statement is executed.
- When the expression is no longer `true`, execution stops.
- The expression is evaluated **before** each execution of the statement. Thus the statement may be executed zero times.

do-while Statement

- The **do-while** statement creates a loop – it has the form:

do *Statement* **while (** *Expression* **);**

```
int index = 10;

do
{
  // do some work
  index = index - 1;
}
while (index > 0); // stop looping when index reaches 0
```

- What is the difference between while and do-while?
  - What happens if the index is initially set to 0, before the do-while is executed?

Notes:

- The expression is evaluated **after** each execution of the statement.
  - Thus the statement in a do-while is executed **at least once.**
  - The statement in a while may never be executed
- The expression must have a boolean type.
- The statement is executed until the expression becomes false.

for Statement

- The **for** statement creates a loop – it has the form:

  for (*Initialization*$_{opt}$; *Expression*$_{opt}$; *Increment*$_{opt}$) *Statement*

```
// Print values from 0 to 3
for (int i = 0; i <= 3; i++)
{
  System.out.print(i + " ");
}
```

- Java 5 introduced another version of the for loop – called `for-each`
  - We'll look at this later

Notes:

- The initialization is executed first and specifies initialization of the loop.
- If the expression is `true`, the statement is executed.
- The loop is exited when the expression becomes `false`.
- The increment expression is executed after each iteration, i.e., after the statement is executed.
- The `for` loop is often used to iterate through an array. We will discuss arrays later.

break Statement

- The **break** statement transfers control to the end of the enclosing loop (`for`, `while`, `do-while`) or `switch` statement
- It has the form:
  break;

```
// this method scans values looking for a specific one
void findValue(int value)
{
  for (int i = 0; ; i++) // No terminate expression
  {
    if (i == value)
    {
      System.out.println("got the value");
      break;  // stop looping, found the value
    }
  }
  // control is here after the break
}
```


Notes:

- `break` statements can carry a label
- The form of the labeled `break` is:
  break *label*;

- Here is an example

```
outer:  // 'outer' is not a reserved word, just a label
for (int i = 0; i < 100; i++) {
   for (int k = 0; k < 100; k++) {
      if (/* some condition */) {
         break outer;  // exit outer loop
      }
   }
}
// control is here after the break
```

continue Statement

- The **continue** statement exits the **current iteration** of the loop (for, while, do-while)
  - It continues with the next iteration
- It has the form:
  continue;

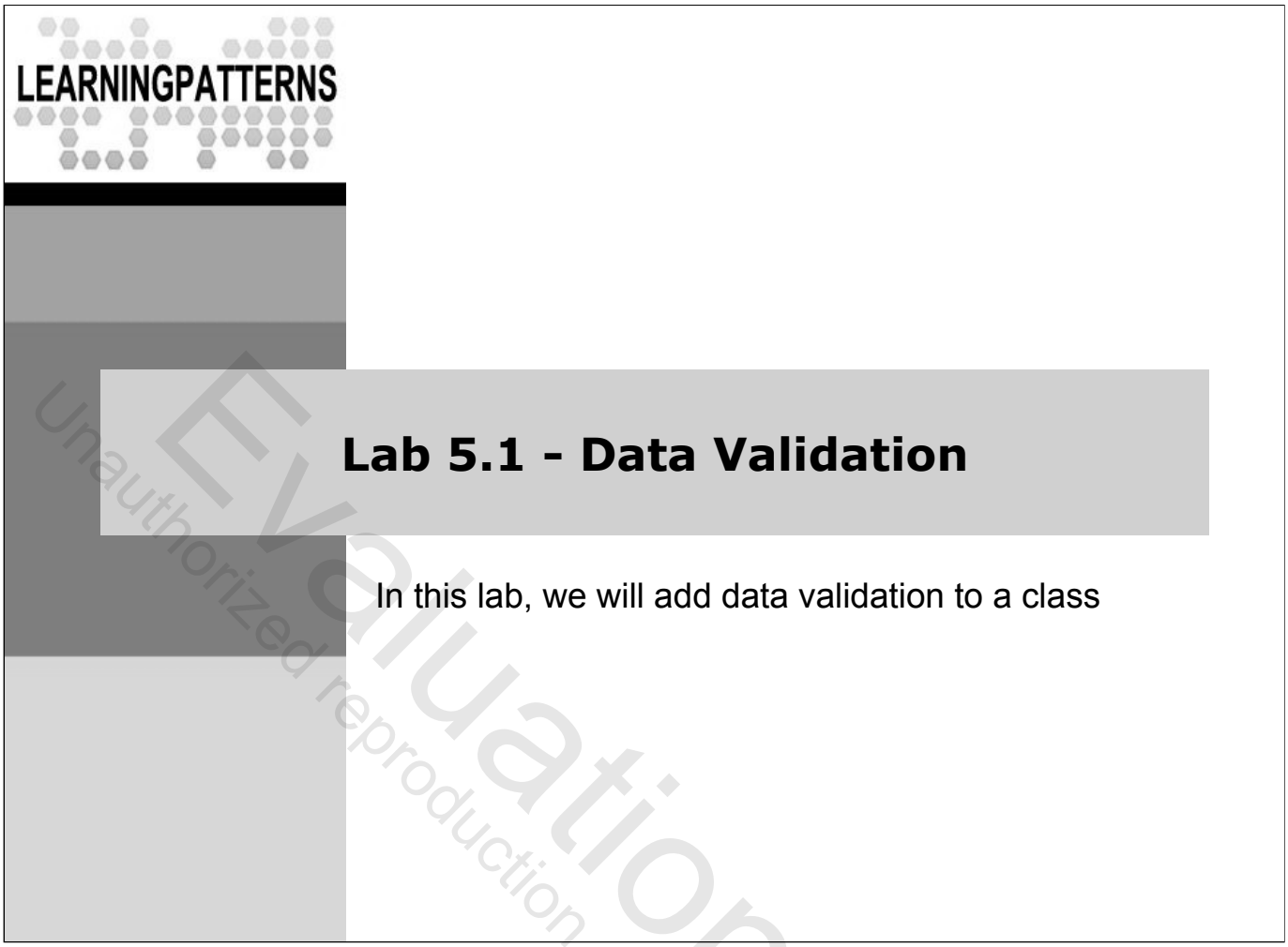
```
// here we use continue to print out even numbers only
for (int i = 1; i <= 10; i++)
{
  if ((i % 2) != 0)  // if not divisible by 2 (not even)
    continue;        // exit this iteration of the loop

  System.out.println(i);
}
```

Notes:

- Unlike break, continue does **not** exit the loop entirely; it just exits the **current iteration** of the loop, and continues with the next iteration.

- continue statements can carry a label, and the form of the labeled continue is:
  continue *label*;

```
outer:
  for (int i = 0; i < 100; i++) {
     for (int k = 0; k < 100; k++) {
        if (k == 17) {
           // exit this iteration of outer loop
           continue outer;
        }
    System.out.println(k);  // Otherwise normal looping goes on
  }
}
```

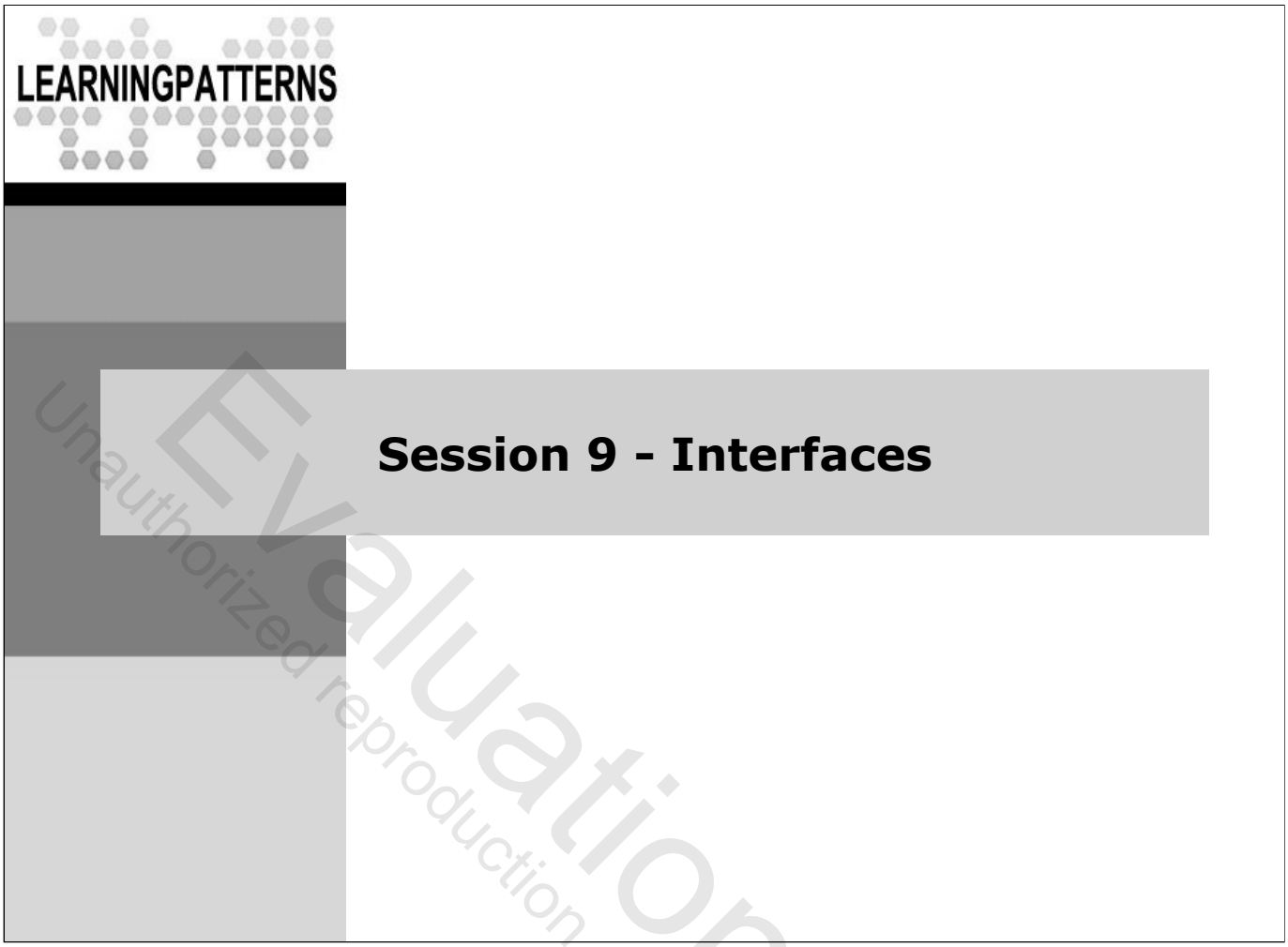



Notes:

Session Review

1. Name the two AND operators in Java. What is the difference between them?
2. True or false: `if` and `if-else` statements require the use of blocks
3. What data types can you use to control a `switch` statement?
4. What is the difference between `while` and `do-while`?
5. What is the difference between `break` and `continue`?

Notes:



Notes:

Session Objectives

- Understand the similarities between interface types and class types
- Use interface types the same way that class types are used
- Explain the role that interfaces play in "programming by contract"
- Define and implement an interface

Notes:

What if All You Have to Share is an Idea

- It often happens that you know what a type will do
  - You know **what** it's behavior (methods) are
  - But you don't know **how** that type will do it
- Or there may be many related types that will implement some behavior **differently**
  - And you want to be able to treat all those varieties the same
- For example, you know a Timepiece needs to display the time
  - But there may be many variations of Timepiece that display time completely differently, and have no shared implementation
  - A clock, a sundial
  - What about a cell phone?
  - One strategy is to make Timepiece an interface



Notes:

Interface Types

- Java interfaces allow you to specify a type that is totally separate from any implementation
  - It is an abstract type that can specify behavior
  - It embodies the idea of what a type is, but not its implementation
  - Interfaces are often used to define ***roles*** played by objects
  - Whereas a class can define how a type fulfills the role (via instance data and method implementation)
- An ***interface*** defines a type that is similar to a class
  - It can have method definitions, but **all methods are `abstract`**
  - Interfaces cannot be instantiated with the `new` keyword
  - It can also have properties, but **all properties are `static final` constants**

Notes:

- A `Person` can play roles such as `Programmer` or `Instructor`.
  - If `Programmer` is an interface with `code` and `test` methods, and
  - `Instructor` is an interface with `teach` and `debugLabs` methods,
  - then `Person` can play the role of `Programmer` if it implements the `code` and `test` methods, and `Person` can also play the role of `Instructor` if it implements the `teach` and `debugLabs` methods.
  - This means that you can use a `Programmer` or `Instructor` reference with this `Person` object, i.e., you can treat a `Person` as simply a `Programmer` or `Instructor`.
- The use of interfaces is often called *Programming by Contract*

Interface Definitions

- An interface definition uses the **`interface`** keyword, in a way similar to the `class` keyword
  - We'll show examples of this using types for a shipping company

```
// definition of interface Moveable
package com.mycompany.shipping;
public interface Moveable {
  // ...
}
```

- Interface methods are declared without a body
  - They are implicitly abstract, and have no implementation

```
package com.mycompany.shipping;
public interface Moveable {
  public void moveTo(String dest);
}
```

Notes:

The `implements` Keyword

- Interfaces are used via the **`implements`** keyword in class definitions
- When you implement an interface you **must** provide implementations for every method of the interface
  - If you don't, the class will not compile

```
import com.mycompany.shipping.Moveable;
public class PosterTube implements Moveable
{
  // provides an implemented moveTo(String) method
  public void moveTo(String dest)
  {
    // PosterTubes's implementation
  }
}
```

Notes:

- **NOTE**: when a class implements an interface, it can include concrete implementations or abstract implementations of the interface methods.
  - If some of those methods are abstract, then the class is abstract and cannot be instantiated.
  - **Once a subclass provides concrete implementations of all the abstract methods it can be instantiated**.

- An "empty" or no-op method **is** an implementation.

```
public void moveTo(String dest)
{

}
```

Example of Using Interface Types

```
class MovingCompany {

  // Moveable is an interface type
  Moveable[] goods = null;

  MovingCompany(Moveable[] goodsIn) {
    goods = goodsIn;
  }

  void deliverAllGoods(String location) {
    for (Moveable m : goods) { // Iterate with for-each
      m.moveTo(location);
    }
  }
}
```

Notes:

Interface Types - Revisited

- Keep in mind that an interface type is very similar to a class type
  - A instance of a class that implements an interface might only be "viewed" by other objects as that **interface type**, and **not** by its actual class type
  - For example, a moving company might not care about what **exactly** it is moving, just that the items are `Moveable`.
  - `Moveable` is a type, but it is an **interface types**, not class type

Notes:

Interface Types - Revisited

- Interface types can be used as **reference variable types**, but cannot be instantiated or used as object types

```
Moveable m = new PosterTube();
```

- Interface types can be used as **parameters to methods**

```
public void moveObject(Moveable m) { ... }
```

- Interface types can be used as **return types** from methods

```
public Moveable getMovedObject()
{
  return m;
}
```


Notes:

- You will often not know (or care) about what kind of type you are working with (class type or interface type).
  - You work with both of them the same way – basically, you call methods on them.

Extending Interfaces

- You can declare ***subinterfaces*** that extend other interfaces, using the **`extends`** keyword
  - This is the **IS-A** relationship again, this time with interfaces
  - Unlike class inheritance, you can extend multiple interfaces
  - A class that implements a subinterface must implement all the interfaces that the subinterface extends (**IS-A**)

```
// A generic movable
public interface Moveable {
  public void moveTo(String dest);
}
```

```
// A moveable that will go onto a truck
public interface Carton extends Moveable {
  public float getSize();
}
```

Notes:

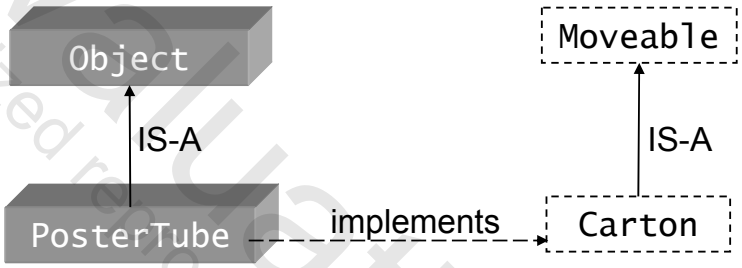
Implementing Extended Interfaces

```
// PosterTube implements Carton and thus Moveable, also
public class PosterTube implements Carton
{
  // from interface Moveable
  public void moveTo(String dest) {
    // ...
  }

  // from interface Carton
  public float getSize() {
    // ...
  }
}
```

Notes:

Example of Using Interface Types

```
interface Moveable { /* ... */ }
interface Carton  extends Moveable    { /* ... */ }

class Car         implements Moveable { /* ... */ }
class PosterTube  implements Carton   { /* ... */ }
class ShippingBox implements Carton   { /* ... */ }
class WardrobeBox extends ShippingBox { /* ... */ }
```



Notes:

- This shows the relationships between the class and interface types used in this example.

- In the example, a `PosterTube` object can be referenced by variables of type:
  - `PosterTube`.
  - `Object`.
  - `Moveable`.
  - `Carton`.
  - That is, a `PosterTube` can be treated as a `PosterTube`, and of course it can be treated as a plain old `Object`. A `PosterTube` can also be treated as a `Moveable` or a `Carton`.

Example of Using Interface Types

```
class MovingCompany {

  // Carton and Moveable are interface types
  Carton[] cartons = null;
  Moveable[] goods = null;

  MovingCompany(Carton[] cartonsIn, Moveable[] goodsIn) {
    cartons = cartonsIn;
    goods = goodsIn;
  }
  void deliverAllGoods(String location) {
    float totalSize = 0.0F;
    for (int i = 0; i < cartons.length; i++) {
      totalSize += cartons[i].getSize();
      cartons[i].moveTo(location);
    }
    for (Moveable m : goods) {
       m.moveTo(location);
    }
  }
}
```

Notes:

- Notice in the example above that all the types involved are interface types.
  - That is, the `MovingCompany` doesn't really know (or care) what **exactly** it is moving, just that the objects are `Carton`s and `Moveable`s.
  - This simplifies the point of view of the `MovingCompany` to just what's needed and no more. Furthermore, the `MovingCompany` class is flexible, in that the exact types that it's dealing with are not hardcoded.

Example of Using Interface Types

```
class GetMoving
{
  public static void main(String[] args)
  {
    Carton[] boxes = { new PosterTube(), new ShippingBox(),
                       new WardrobeBox() };

    Moveable[] bigStuff = { new Car() };

    MovingCompany acme = new MovingCompany(boxes, bigStuff);

    acme.deliverAllGoods("San Francisco");
  }
}
```

Notes:

Interfaces are Abstract

- Interfaces are implicitly `abstract`
  - You can also declare this explicitly, though you generally don't
  - The definition below is equivalent to one without using abstract

```
// 'abstract' legal, but generally not included
public abstract interface Moveable {
  // ...
}
```

  - `abstract` also legal on methods but generally not used
  - The definition below is equivalent to one without using abstract

```
public interface Moveable {
  // 'abstract' legal, but generally not included
  public abstract void moveTo(String dest);
}
```

Notes:

Data Members in Interfaces

- Interfaces can't declare instance data
  - If you need instance data, you need to use a class
- Interface data members are **implicitly** both **`static`** and **`final`**, and usually are declared as such, for clarity
  - It is legal to leave out final and static, but the compiler will just add them in for you

```
package com.mycompany.shipping;
public interface Moveable
{
  // static and final are usually included
  // note the ALL_CAPS convention for class constants

  public static final String HOME_LOCATION = "HQ Office";
  public void moveTo(String dest);
}
```

Notes:

- Another example is the `WindowConstants` interface.
  - It contains constants for the possible actions that can be taken when a user requests a window be closed.
  - These constants are used with the `setDefaultCLoseOperation` method when designing (GUIs) with Swing.

```
package javax.swing;
public interface WindowConstants
{
  // static and final are usually included
  // note the ALL_CAPS convention for constants
  public static final int DISPOSE_ON_CLOSE = 2;
  public static final int DO_NOTHING_ON_CLOSE = 0;
  public static final int HIDE_ON_CLOSE = 1;
}
```

Implementing Multiple Interfaces

- A class can implement more than one interface

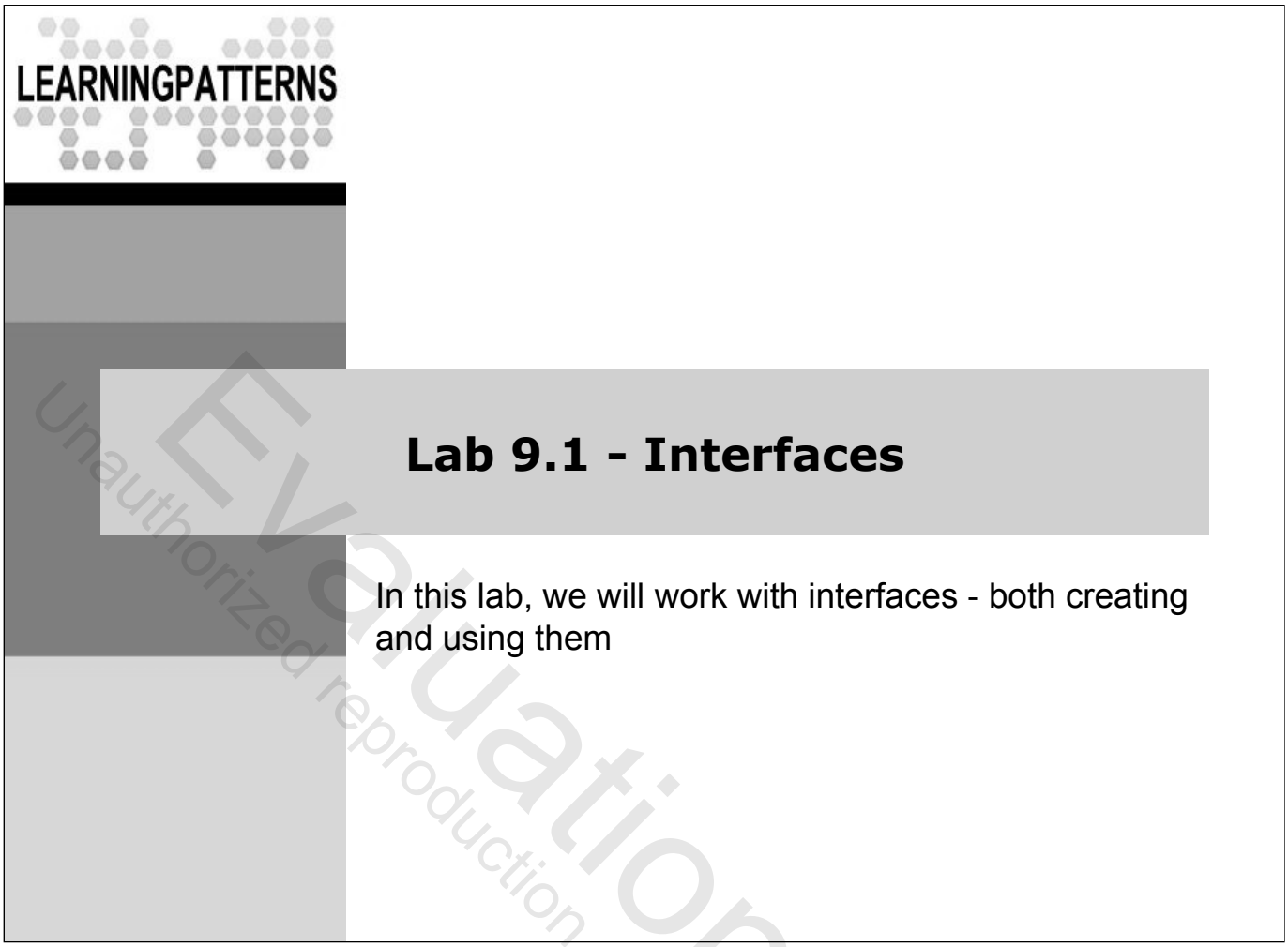
```
public class Car implements Moveable, Serviceable
{
  // from interface Moveable
  public void moveTo(String dest) {
    // ...
  }
  // from interface Serviceable
  public void serviceEngine() {
    // every 30,000 miles turn on "Check Engine" light
  }
}
```

- This is a useful property of interfaces
  - Has many of the advantages of multiple inheritance, which Java doesn't support
  - Is much less complex because the interfaces just declare a protocol, not an implementation

Notes:

- Interfaces are used in place of multiple inheritance. Consider the "Pegasus problem," in which we wish to define a flying horse.
  - If multiple inheritance was available, you might subclass both `Bird` and `Horse`. However, if there were any variables or methods defined in both `Bird` and `Horse`, you would have a duplicate/conflict situation and the compiler would have to resolve this somehow. Already it's getting a bit complicated.
  - If only single inheritance was available, would you extend `Bird` or `Horse`? Isn't Pegasus really a horse that has some of the characteristics of birds? Do all birds fly?
  - Using interfaces, we can define a role (a set of abstract methods) named `FlyingAnimal`, that can be used with **any** creature that can fly -- we can use it for Pegasus too.

```
interface FlyingAnimal

class Pegasus
extends Horse
implements FlyingAnimal
```



Notes:

Session Review

1. How does "programming by contract" apply to interfaces?
2. What keyword is used for a class to "sign an interface contract?"
3. True or false: interfaces can be placed in packages.
4. Explain the difference between interfaces and abstract classes.
5. True or false: interfaces can exhibit inheritance characteristics similar to classes.
6. True or false: interfaces can only have a default or no-argument constructor.

Notes: