

A large, two-story, light-colored building with a red-tiled roof and a central tower, surrounded by green grass and trees under a clear blue sky.

MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

CS401 Modern Programming
Practices (MPP)
Dr. Bright Gee Varghese

CS-401 MODERN PROGRAMMING PRACTICES						
	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
OOAD / RDBMS / OOP	AM: Lesson 1: The OO Paradigm for Building Software Solutions	AM: Lesson 2: Associations among Objects and Classes [Lab 1 due 10 AM]	AM: Lesson 3: Inheritance and Composition [Lab 2 due 10 AM]	AM: Lesson 4: Interaction Diagrams [Lab 3 due 10 AM]	AM: Lesson 5: Inheritance and Abstraction [Lab 4 due 10 AM]	AM: Lesson 6: Relational Model, View & Normalization [Lab 5 due 10 AM]
	PM: Lab 1	PM: Lab 1 solutions, Lab 2	PM: Lab 2 solutions, Lab 3	PM: Lab 3 solutions, Lab 4	PM: Lab 4 solutions, Lab 5	Lab 5 solutions
	AM: Lesson 7: SQL (DML&DDL)	AM: Lesson 8: Index, SQL Injection, JDBC application & Intro to Maven [Lab 6 due 10 AM] [Lab 7 due 5 PM]	AM: Review for Midterm exam Lab 7 & 8 solutions	MIDTERM EXAM	AM: Lesson 9: Interfaces in Java 8 and the Object Superclass	AM: Lesson 10: Functional Programming in Java [Lab 9 due 10 AM]
	PM: Lab 6, 7	PM: Lab 6 solutions, Lab 8	PM: Study for Midterm		PM: Lab 9	Lab 9 solutions
OOP	AM: Lesson 11: The Stream API	AM: Lesson 11: The Stream API [Lab 10 due 10 AM]	AM: Lesson 12 Best Programming Practices with Java 8 [Lab 11 due 10 AM]	AM: Lesson 13 Generic Programming	AM: Review for Final exam Lab 13 solutions	Final Exam
	PM: Lab 10	PM: Lab 10 solutions, Lab 11	PM: Lab 11 solutions, Lab 12	PM: Lab 12 solutions, Lab 13		
	AM: Java Swing PM: Course Project	Course Project	Course Project	AM: Project Presentation – Connect whole with parts		

Lecture 9: Interfaces in Java 24 and the Object Superclass

Wholeness Statement

Java supports inheritance between classes in support of the OO concepts of inherited types and polymorphism. Interfaces support encapsulation, play a role similar to abstract classes, and provide a safe alternative to multiple inheritance.

Likewise, relationships of any kind that are grounded on the deeper values at the source of the individuals involved result in fuller creativity of expression with fewer mistakes.

Outline

- ❑ Java 24 interfaces: Introduction
- ❑ Java 24 interfaces: Application of Default Method
- ❑ Java 24 interfaces and the Diamond Problem
- ❑ Java 24 Sealed Interface
- ❑ FPP Review: Overriding Methods in the Object Class (Optional)

Java 24 Features of Interfaces

Default Methods (Java 8)

- Methods with a default implementation using default keyword. Enables interface evolution.

Static Methods in Interfaces (Java 8)

- Interfaces can declare static methods.

Private Methods (Java 9)

- Interfaces can define private methods to share code among default methods.

Sealed Interfaces (Java 15 – preview, Java 17 – stable)

- Restrict which classes or interfaces can implement or extend them.


Java 24 Features of Interfaces

- Variables in Interfaces
 - All variables in an interface are implicitly public static final (constants).
 - You cannot declare:
 - private, protected, or package-private fields
 - non-final or non-static fields
 - instance variables (since interfaces don't have constructors)
- No constructors

See Demos in package

`lesson09.lecture.new_java.interface_example.syntax.Vehicle`

```
public interface Vehicle {  
    String TYPE = "Transport";  
    void start();  
    default void stop() {  
        log("Stopping...");  
        System.out.println("Vehicle Stopped");  
    }  
    private void log(String msg) {  
        System.out.println(msg);  
    }  
    static void honk() {  
        System.out.println("Beep...beep");  
    }  
}
```



```
bright~$javap -cp  
./out/production/mppJuneP  
lesson9.smart_vehicles.Vehicle  
Compiled from "Vehicle.java"  
public interface  
lesson9.smart_vehicles.Vehicle {  
    public static final java.lang.String TYPE;  
    public abstract void start();  
    public default void stop();  
    public static void honk();  
}  
bright~$
```


New Programming Style

Default Methods in an interface eliminate the need to create special classes that represent a default implementation of the interface.

- Examples from pre-Java 8 of default implementations of interfaces:
WindowListener / WindowAdapter (in the AWT),
List / AbstractList. [See JavaLibrary project in workspace]
- Now, in developing new code, it is possible in many cases to place these default implementations in the interface directly.

Static Methods in an interface eliminate the need to create special utility classes that naturally belong with the interface.

- Examples from pre-Java 8 of how interfaces sometimes have companion utility classes (consisting of static methods):
Collection / Collections [See JavaLibrary project]
Path / Paths.
- For new code, it is now possible to place this static companion code directly in the interface.

```

public class AuditUtils {
    public static String getCurrentUser() {
        // ...fetch user from security context...
        return "admin"; // Example
    }
}

```

```

public class Invoice {
    public void update() {
        String user =
        AuditUtils.getCurrentUser();
        // ...update invoice logic...
    }
}

```

```

public class Order {
    public void approve() {
        String user =
        AuditUtils.getCurrentUser();
        // ...approve order logic...
    }
}

```

Evolving Design: The Auditable Interface

```

public interface Auditable {
    void audit(String action);

    // Static method (moved from AuditUtils!)
    static String getCurrentUser() {
        // ...fetch user from security context...
        return "admin"; // Example
    }
}

```


Solution to Evolving API Problem

When you need to add new methods to an existing interface, provide them with default implementations using the new Java 8 default feature. Then

- legacy code will not be required to implement the new methods, so existing code will not be broken
- new functionality will be available for new client projects.

`lesson09.lecture.new_java.interface_example.default_methods`

Exercise 9.1 – Rewrite List Interface

Explore the package `exercise9_1` in the InClass Exercises project. You will see a class `MyStringList` along with an interface `StringList` that it implements. `StringList` contains several common list operations:

```
String[] strArray();  int size();  void setSize(int s);  
void add(String s);  String get(int i);
```

Show how to use Java 8 default methods to provide implementations of `add` and `get`. This considerably reduces the effort to implement `StringList` in `MyStringList` since most of the implementation work has been moved into the interface.

NOTE: Something like this could have been done in Java's `List` interface (moving most of the implementations from `AbstractList` into default methods of `List`), except that the `List` interface was created long before default interface methods had been introduced.

Exercise 9.1 – Rewrite List Interface

```
public interface StringList {
```

```
String[] strArray();
```

```
int size();
```

```
void incrementSize();
```

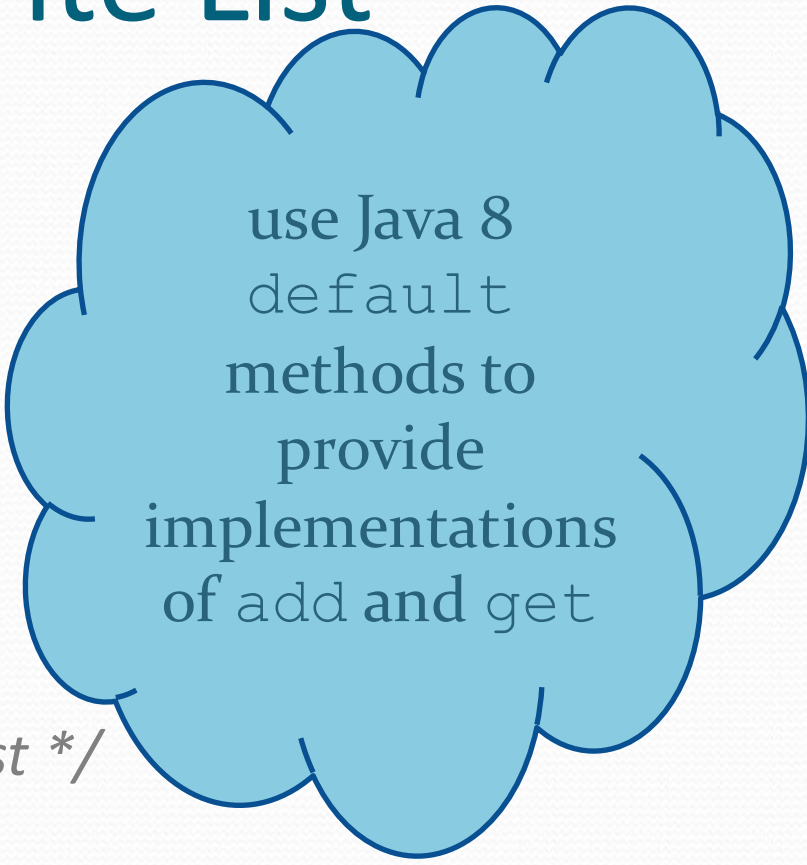
```
/** Adds a string to the end of the list */
```

```
public void add(String s);
```

```
/** Gets the i_th string in the list */
```

```
public String get(int i);
```

```
}
```



use Java 8
default
methods to
provide
implementations
of add and get

Exercise 9.1 – Solution

*/** Adds a string to the end of the list */*

```
default public void add(String s) {  
    strArray()[size()] = s;  
    incrementSize();  
}
```

*/** Gets the i_th string in the list */*

```
default public String get(int i) {  
    if (i < 0 || i >= size()) {  
        return null;  
    }  
    return strArray()[i];  
}
```

return i > 0 && i < strArray().length ? strArray()[i]: null;

Outline

- ❑ Java 24 interfaces: Introduction
- ❑ Java 24 interfaces: Application of Default Method
- ❑ Java 24 interfaces and the Diamond Problem
- ❑ Java 24 Sealed Interface
- ❑ FPP Review: Overriding Methods in the Object Class (Optional)

Application of Default Methods

Review of Enums

- An *enumerated type* is a Java class all of whose possible instances are explicitly enumerated during initialization.
- Example:

```
public enum Size { SMALL, MEDIUM, LARGE};
```

//usage:

```
if (requestedSize==Size.LARGE)  
    applyDiscount();
```

- The enum `Size` (which is a special kind of Java class) has been declared to have just three instances, named `SMALL`, `MEDIUM`, `LARGE`.

Review of Enums (cont)

Two important applications for enums:

1. Using enums as *constants* in an application

- *Weak Programming Practice:* Create a class (or interface) containing constants, stored as public static final values – most often arising when constants are ints or Strings
- *Problem.* No compiler control over usage of these constants when they occur as input arguments to methods (example on next slide)
- *Better Approach* Represent constants as instances of an enumerated type.

2. Optimal, threadsafe implementation of the Singleton Pattern

Example of Handling Constants in Java

In the java.awt package there is a class Label, used to represent a label in a UI (built using the old AWT). It makes use of constants to designate alignment properties: LEFT, CENTER, RIGHT. This use of constants is flawed, but it is a commonly used style

```
Label.java x
Since: 1.0
Author: Sami Shaio

59 public class Label extends Component implements Accessible {
60
61     static {
62         /* ensure that the necessary native libraries are loaded */
63         Toolkit.loadLibraries();
64         if (!GraphicsEnvironment.isHeadless()) {
65             initIDs();
66         }
67     }
68
69     Indicates that the label should be left justified.
72     public static final int LEFT = 0;
73
74     Indicates that the label should be centered.
77     public static final int CENTER = 1;
78
79     Indicates that the label should be right justified.
82     public static final int RIGHT = 2;
83 }
```

```
public class Label extends Component implements Accessible {
    public Label(String text, int alignment) throws HeadlessException {
        GraphicsEnvironment.checkHeadless();
        this.text = text;
        setAlignment(alignment);
    }

    public synchronized void setAlignment(int alignment) {
        switch (alignment) {
            case LEFT:
            case CENTER:
            case RIGHT:
                this.alignment = alignment;
                LabelPeer peer = (LabelPeer)this.peer;
                if (peer != null) {
                    peer.setAlignment(alignment);
                }
                return;
            }
            throw new IllegalArgumentException("improper alignment: " +
                alignment);
        }
    }
}
```

Problem: No compiler control over use of these constants.
Could make the following call:

```
Label label = new Label("Hello", 23);
```

You won't know till you run the code that “23” is meaningless. The compiler sees that a value of the correct type has been used, but at runtime, 23 will be recognized as an illegal value.

It is better to control the values passed in with the help of the compiler. This is accomplished using an enum to store constants, rather than collecting together a bunch of public static final integers.

Improved Label Using enums

```
public enum Alignment {  
    /**  
     * Indicates that the label should be left justified.  
     */  
    LEFT,  
  
    /**  
     * Indicates that the label should be centered.  
     */  
    CENTER,  
  
    /**  
     * Indicates that the label should be right justified.  
     * @since JDK1.0t.  
     */  
    RIGHT;  
}
```

```
//Better way, not currently implemented  
//in Java libraries  
public class Label {  
    private String text;  
    private Alignment alignment;  
    public Label(String text, Alignment alignment) {  
        this.text = text;  
        setAlignment(alignment);  
    }  
    public synchronized void setAlignment(Alignment alignment)  
    {  
        this.alignment = alignment;  
    }  
    public String getText() {  
        return text;  
    }  
    public Alignment getAlignment() {  
        return alignment;  
    }  
}
```

See the demo: `lesson09.lecture.enums.*`

Review of Best Practice for Using enums

From Bloch, *Effective Java* (2nd edition):

Use enums (in place of public static final variables) whenever you need a fixed set of constants all of whose values you know at compile time.

Best Practices, continued

- Question: What if you have constants that must be of specific types, like int, double or String (or another type)?

```
class DimConstants {  
    public static final double LENGTH = 1.0;  
    public static final double WIDTH = 2.0;  
}  
class Test {  
    public static void main(String[] args) {  
        System.out.println(DimConstants.LENGTH);  
    }  
}
```

- Solution: Use an enum constructor.

```
class Test {  
    public static void main(String[] args) {  
        System.out.println(Dim.LENGTH.val());  
    }  
}
```

```
public enum Dim {  
    LENGTH(1.0),  
    WIDTH(2.0);  
    double val;  
    Dim(double x) {  
        val = x;  
    }  
    public double val() {  
        return val;  
    }  
}
```


Exercise 9.2

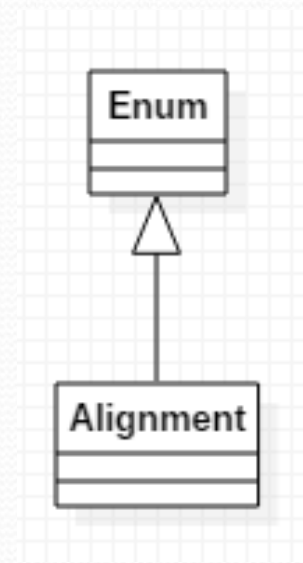
- Below is a `Constants` class consisting of `public final static` variables to provide constants for the rest of the application. Replace with an `enum Const` that provides the same functionality. Refactor the `main` method so that it uses the new `Const` type.
- You can find `Constants` and a test class in the `InClassExercises` project.

```
public class Constants {  
    public static final String COMPANY = "Microsoft";  
    public static final int SALES_TARGET = 200000000;  
}
```

```
public enum Constants {  
    COMPANY("Microsoft"), SALESTARGET(20000000);  
  
    private int target;  
    private String companyName;  
    Constants(String companyName) {  
        this.companyName = companyName;  
    }  
  
    Constants(int target) {  
        this.target = target;  
    }  
    public int getTarget() {  
        return target;  
    }  
  
    public String getCompanyName() {  
        return companyName;  
    }  
}
```


Review of enum Implementation in Java

- In the Label example (earlier slide), each of the instances declared within the `Alignment` enum has type `Alignment`, which is a subclass of `Enum`. Therefore
 - `Alignment` is itself a *class*
 - `Alignment` is not allowed to inherit from any other class (multiple inheritance not allowed).

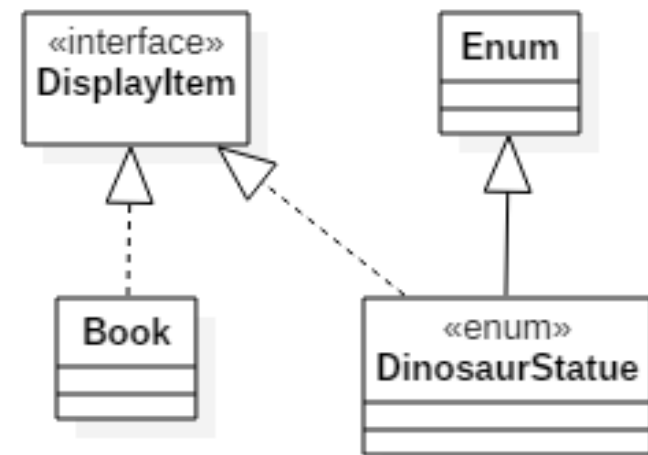


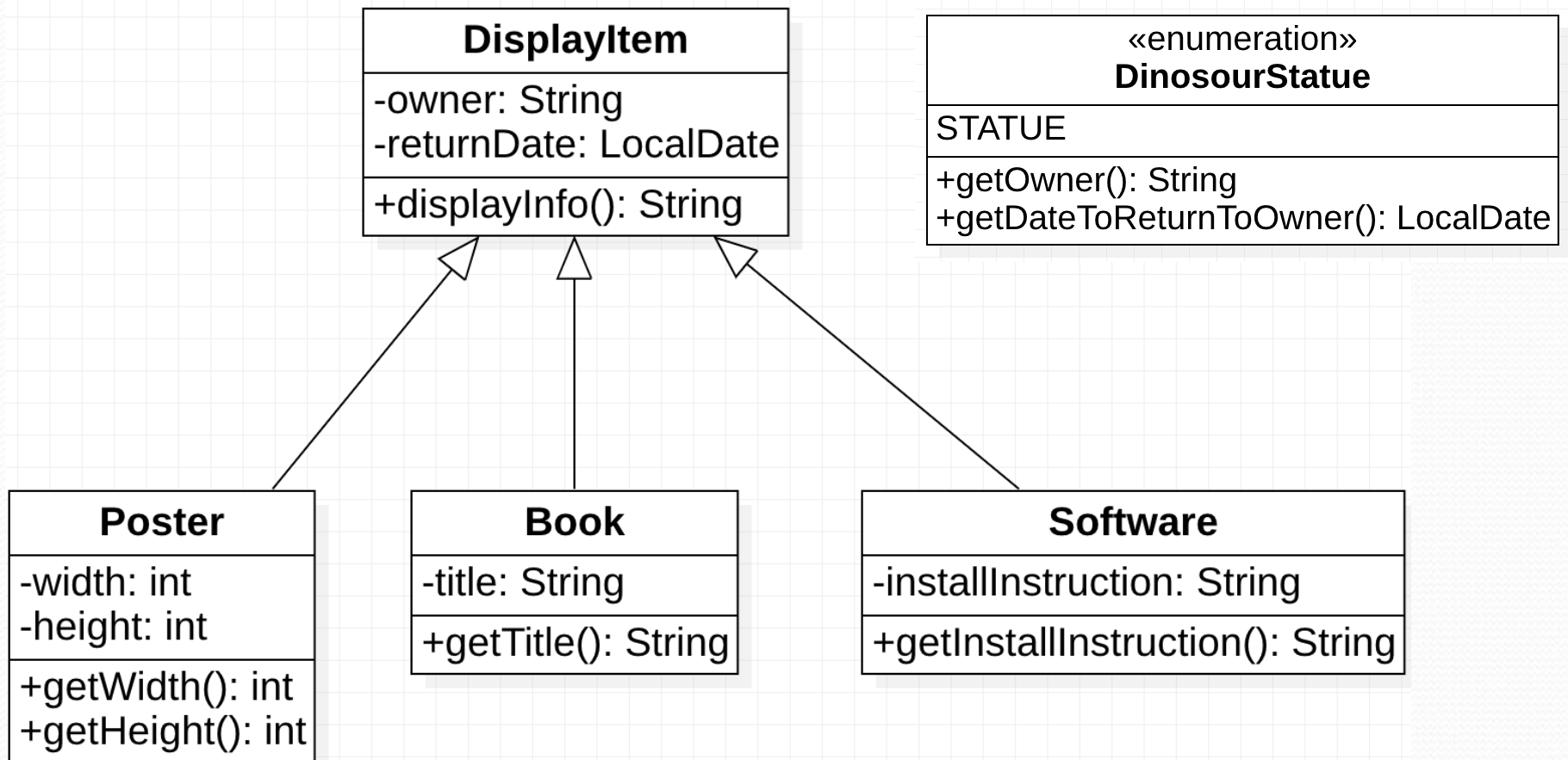
In Java 8, Enums Can “inherit”

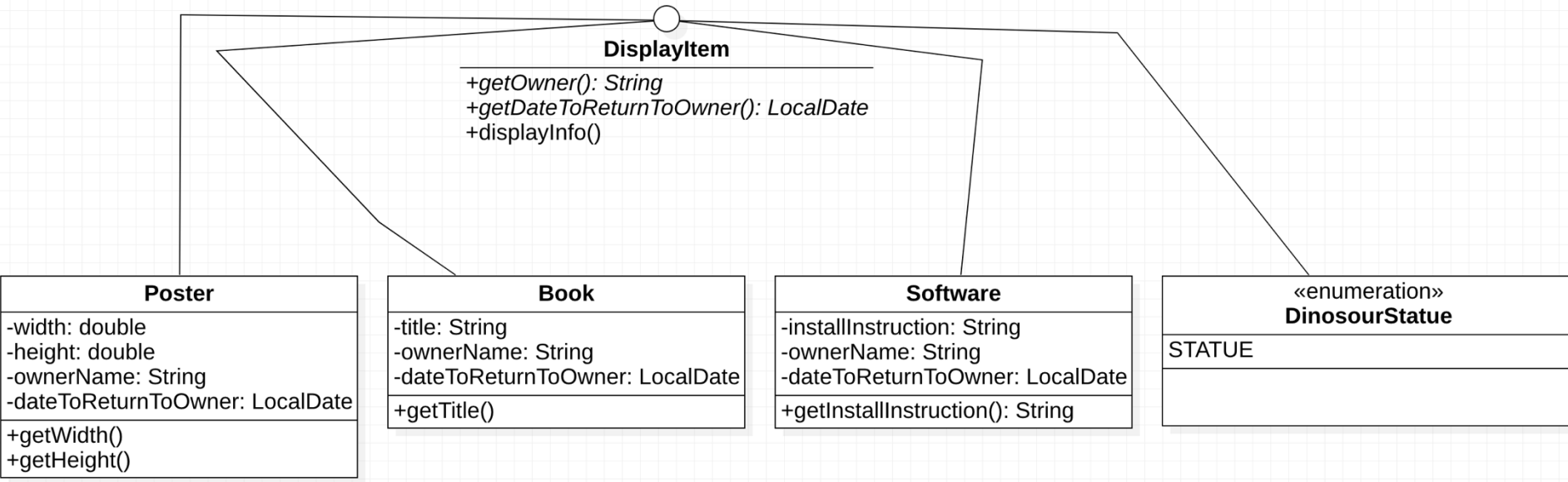
//from lesson09.lecture.enums3.java8

```
interface DisplayItem {  
    default String displayInfo() {  
        . . .  
    }  
}  
  
class Book implements DisplayItem {  
}  
  
enum DinosaurStatue implements DisplayItem {  
    INSTANCE;  
}
```

See lesson09.lecture.enums3.java7 and
lesson09.lecture.enums3.java8







```

public interface DisplayItem {
    public abstract String getOwner();
    public abstract LocalDate getReturnDate();
    default void displayInfo() {
        System.out.println(getOwner() + " " + getReturnDate());
    }
}

```



```

public enum DinasourStatue implements DisplayItem{
    DINASOUR_STATUE("Jack", LocalDate.of(2025, 7, 10));
    private final String owner;
    private final LocalDate returnDate;

    DinasourStatue(String owner, LocalDate returnDate) {
        this.owner = owner;
        this.returnDate = returnDate;
    }

    @Override
    public String getOwner() {
        return owner;
    }

    @Override
    public LocalDate getReturnDate() {
        return returnDate;
    }
}

```

Using enums to Create Singletons

- A *singleton* class is a class that can have at most one instance
- Easy implementation using an enum:

```
enum MySingleton {  
    INSTANCE;  
    public void behavior() {}  
}  
  
//access it like this:  
MySingleton.INSTANCE.behavior();
```

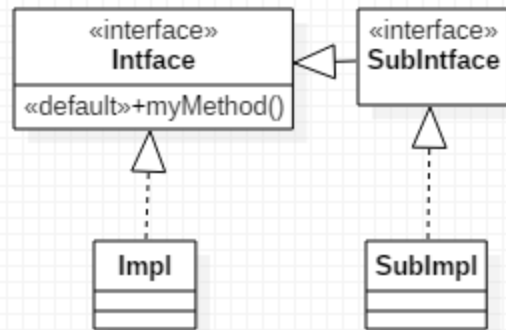
Demo: `lesson09.lecture.singletons`

Outline

- ❑ Java 24 interfaces: Introduction
- ❑ Java 24 interfaces: Application of Default Method
- ❑ Java 24 interfaces and the Diamond Problem
- ❑ Java 24 Sealed Interface
- ❑ FPP Review: Overriding Methods in the Object Class (Optional)

Rules for Default Methods in an Interface

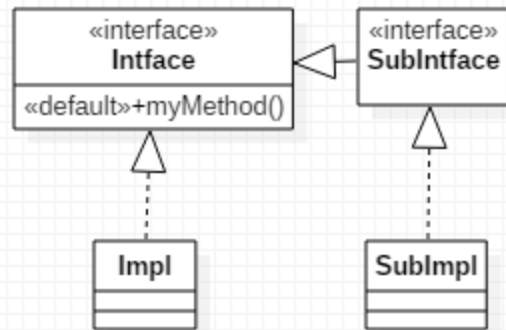
- If a class implements an interface with a default method, that class inherits the default method (or can override it).



- Potential clash if
 - two interfaces have the same method, or
 - one interface and a superclass have the same method

Rules for Default Methods in an Interface

- If a class implements an interface with a default method, that class inherits the default method (or can override it).



- Potential clash if
 - two interfaces have the same method, or
 - one interface and a superclass have the same method


```
interface I1 {  
    void fun();  
}  
interface I2 {  
    void fun();  
}
```

```
class MyClassImpl implements I1, I2 {  
  
    @Override  
    public void fun() {  
        //...  
    }  
}
```




```
interface I1 {  
    void fun();  
}  
interface I2 {  
    default void fun(){}  
}
```

```
abstract class A implements I1, I2 {  
    @Override  
    abstract public void fun();  
}
```

OR

```
class MyClassImpl implements I1, I2 {  
  
    @Override  
    public void fun() {  
        //...  
    }  
}
```



```
interface I1 {  
    void fun();  
}  
interface I2 {  
    void fun();  
}
```

```
interface I3 extends I1, I2 {  
  
}
```




```
interface I1 {  
    void fun();  
}  
interface I2 {  
    default void fun(){}  
}
```



```
interface I3 extends I1, I2 {  
    @Override  
    default void fun() {}  
}
```

OR

```
interface I3 extends I1, I2 {  
    @Override  
    void fun();  
}
```



```
interface I1 {  
    default void fun(){}  
}  
interface I2 {  
    default void fun(){}  
}
```



```
interface I3 extends I1, I2 {  
    @Override  
    default void fun() {  
        //I1.super.fun();  
    }  
}
```

OR

```
interface I3 extends I1, I2 {  
    @Override  
    void fun();  
}
```

```
interface I1 {  
    default void fun(){}  
}  
interface I2 {  
    default void fun(){}  
}
```

```
abstract class A implements I1, I2 {  
    @Override  
    abstract public void fun();  
}
```

```
class MyClassImpl implements I1, I2 {  
    @Override  
    public void fun() {  
        //...  
    }  
}
```




```
interface I1 {  
    default void fun(){  
        System.out.println("fun in I1");  
    }  
}
```

```
class SuperClass{  
    public void fun(){  
        System.out.println("fun in Super class");  
    }  
}
```

```
class MyClassImpl extends SuperClass implements I1{ }
```

```
class Main {  
    public static void main(String[] args) {  
        new MyClassImpl().fun();  
    }  
}
```



fun in Super class

Static & Private Methods

DO NOT Clash

- In Java, both `static` and `private` methods in interfaces are not inherited by implementing classes — so method clashes do not occur, even if multiple interfaces define methods with the same name.
- Static methods belong to the interface itself, not to the implementing class.
 - They are not inherited.
 - You must call them using the interface name:
`InterfaceName.method()`.

See demo:

`lesson09.lecture.new_java.interface_example.diamond.static_methods`

Static & Private Methods DO NOT Clash

- Private methods in interfaces are used only inside that interface.
 - They are like private helpers.
 - They are not visible or usable outside the interface (not even in the implementing class).

See demo:

```
lesson09.lecture.new_java.interface_example.diamond.private  
_methods
```


Exercise 9.4

Look at the code snippets on the PDF file in `lesson09.exercise_4` package of the `InClassExercises` project. Try to determine, without using a compiler, what happens when the code is compiled/run.

1. What happens when the following code is compiled/run from the main method?

<pre>public interface Iface1 { default int myMethod(int x) { return 2 * x; } } public interface Iface2 { default int myMethod(int x) { return 2 * x; } }</pre>	<pre>public class Impl implements Iface1, Iface2 { } public class Main { public static void main(String[] args) { Impl ob = new Impl(); System.out.println(ob.myMethod(2)); } }</pre>
--	--

- A. Compiler error
- B. Runtime error
- C. 4 is printed to the console

2. What happens when the following code is compiled/run from the main method?

<pre>public interface Iface1 { default int myMethod(int x) { return 2 * x; } } public interface Iface2 { public int myMethod(int x); }</pre>	<pre>public class Impl implements Iface1, Iface2 { } public class Main { public static void main(String[] args) { Impl ob = new Impl(); System.out.println(ob.myMethod(2)); } }</pre>
--	--

- A. Compiler error
- B. Runtime error
- C. 4 is printed to the console

3. What happens when the following code is compiled/run from the main method?

```
public interface Iface1 {  
    default int myMethod(int x) {  
        return 2 * x;  
    }  
}  
  
public interface Iface2 {  
    static int myMethod(int x) {  
        return x + 1;  
    }  
}
```

```
public class Impl implements Iface1, Iface2 {  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Impl ob = new Impl();  
        System.out.println(ob.myMethod(2));  
    }  
}
```

- A. Compiler error
- B. Runtime error
- C. 4 is printed to the console

Main Point 1

Interfaces are used in Java to specify publicly available services in the form of method declarations. A class that implements such an interface must make each of the methods operational. Interfaces may be used polymorphically, in the same way as a superclass in an inheritance hierarchy. Because many interfaces can be implemented by the same class, interfaces provide a safe alternative to multiple inheritance. Java8 now supports static and default methods in an interface, which make interfaces even more flexible.

The concept of an interface is analogous to the creation itself – the creation may be viewed as an “interface” to the undifferentiated field of pure consciousness; each object and avenue of activity in the creation serves as a reminder and embodiment of the ultimate reality.

Outline

- ❑ Java 24 interfaces: Introduction
- ❑ Java 24 interfaces: Application of Default Method
- ❑ Java 24 interfaces and the Diamond Problem
- ❑ Java 24 Sealed Interface
- ❑ FPP Review: Overriding Methods in the Object Class (Optional)

What is a Sealed Interface?

- A sealed interface restricts which classes or interfaces are allowed to implement or extend it.
- Each permitted class must also be:
 - final → cannot be extended
 - sealed → further restrict its subclasses
 - non-sealed → open it back up for extension

See Demo:

`lesson09.lecture.new_java.interface_example.sealed_demo`

Why Do We Need Sealed Interfaces in Java?

1. Controlled Extensibility (Design Safety)

- In traditional interfaces:
 - Any class can implement the interface.
 - You have no control over who implements it — even across packages and libraries.
- With sealed interfaces, you explicitly control which classes can implement an interface. This enforces a closed hierarchy — which makes the codebase:
 - More predictable
 - Safer from unintended implementations
 - Easier to maintain

Why Do We Need Sealed Interfaces in Java?

2. Exhaustive Pattern Matching (Better switch)
 - Sealed interfaces enable the compiler to verify that all possible cases are handled.
 - You get compile-time exhaustiveness checking, similar to `enum` but with object hierarchies.

See Demo:

`lesson09.lecture.new_java.interface_example.patternmatching`

sealed interface Command permits

LoginCommand, LogoutCommand, ShowProfileCommand, UserPluginCommand {

void execute();

}

final class LoginCommand implements Command{

@Override

public void execute() {}

}

final class LogoutCommand implements Command{

@Override

public void execute() {}

}

final class ShowProfileCommand implements Command{

@Override

public void execute() {}

}

non-sealed class UserPluginCommand implements Command {

// plugin-specific API

@Override

public void execute() {}

}

```

class MyTestClass {
    public static void main(String[] args) {
        Command command = new UserPluginCommand();//It can be any...
        switch (command) {
            case LoginCommand loginCommand -> {
                loginCommand.execute();
            }
            case LogoutCommand logoutCommand -> {
                logoutCommand.execute();
            }
            case ShowProfileCommand showProfileCommand -> {
                showProfileCommand.execute();
            }
            case UserPluginCommand userPluginCommand -> {
                userPluginCommand.execute();
            }
        }
    }
}

```

Ref:

1. <https://openjdk.org/jeps/360>
2. <https://stackoverflow.com/questions/63860110/what-is-the-point-of-extending-a-sealed-class-with-a-non-sealed-class/63887136#63887136>
3. <https://stackoverflow.com/questions/63860110/what-is-the-point-of-extending-a-sealed-class-with-a-non-sealed-class>


```
public sealed class Person permits Patient, HealthTracker { }
```

```
public final class Patient extends Person {  
    // Core patient logic  
}
```

```
// Allow 3rd-party developers to add new health trackers (plugins)  
public non-sealed class HealthTracker extends Person {  
    // Base class for fitness bands, smartwatches, glucose monitors, etc.  
}
```

```
public class SleepTracker extends HealthTracker { }  
public class HeartRateMonitor extends HealthTracker { }
```

Why is it useful?

- Compile-time safety: Java ensures that all subtypes are handled.
- Avoids runtime errors: No `ClassCastException` or missing cases.
- Makes code future-proof: If a new subtype is added, the compiler will flag all affected switch expressions as incomplete.

Outline

- ❑ Java 24 interfaces: Introduction
- ❑ Java 24 interfaces: Application of Default Method
- ❑ Java 24 interfaces and the Diamond Problem
- ❑ Java 24 Sealed Interface
- ❑ FPP Review: Overriding Methods in the Object Class (Optional)

FPP Review: Overriding Methods in the Object Class

The `Object` class is the superclass of all Java classes, and contains several useful methods -- in most cases, they are useful *only if* they are overridden.

- `toString`
- `equals`
- `hashCode`

Overriding toString()

The purpose of `toString()` is to provide a (readable) `String` representation (which can be logged or printed to the console) of the state of an object.

Example from FPP:

```
// toString for an Account object
public String toString(){
    String ret =
        "Account type: " + acctType +
        "\nCurrent bal:  " + balance;
    return ret;
}
```

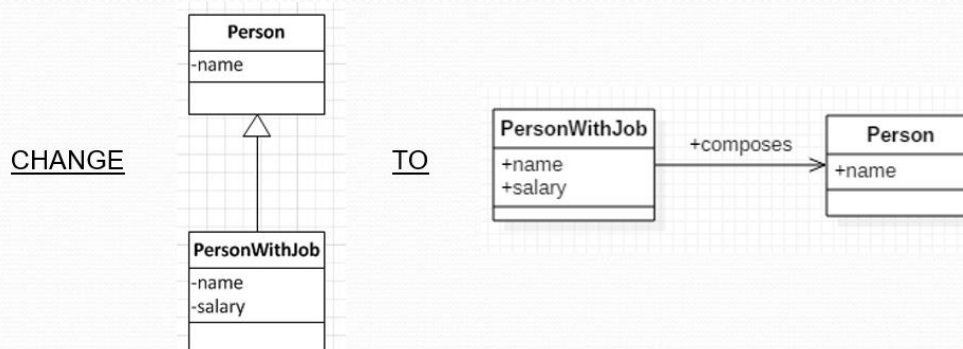
Best Practice. For every significant class you create, override the `toString` method.

Overriding equals()

Care is needed in overriding equals when one class inherits from another.

Best Practices Suppose B is a subclass of A.

1. If it is acceptable for B to use the same equals method as used in A, then the best strategy is the *instanceof strategy* and make equals final.
See `lesson09.lecture.overrideequals.instanceofstrategy3`.
2. If *two different equals methods* are required, two strategies are possible
 - A. Use the same classes strategy, but declare subclass B to be final
See `lesson09.lecture.overrideequals.sameclassesstrategy`
 - B. Use composition instead of inheritance – this will always work as long as the inheritance relationship between B and A is not needed (e.g. for polymorphism). See `lesson09.lecture.overrideequals.composition`



Overriding hashCode ()

There are two general rules for creating hash codes:

- I. (Primary Hashing Rule) Equal keys must be given the same hash code (otherwise, the same key will occupy different slots in the table)
If $k1.equals(k2)$ then $k1.hashCode() == k2.hashCode()$
- II. (Secondary Hashing Guideline) Different keys should be given different hash codes (if not, in the worst case, if every key is given the same hash code, then all keys are sent to the same slot in the table; in this case, hashtable performance degrades dramatically).

Best Practice: The hash codes should be distributed as evenly as possible (this means that one integer occurs as a hash code approximately just as frequently as any other)

Overriding hashCode ()

Best Practices:

- Whenever equals is overridden, hashCode should also be overridden
- The hashCode method should take into account the same fields as the equals method
- the class on which the object is based should be *immutable*

To define your own hashCode method, use the Objects.hash(...) method.

Example

To override hashCode, we make use of the Java library method `Objects.hash`, which takes any number of arguments; the method creates a hashcode based on the hashcodes of the instance variables of `Person`

```
public class Person {  
    private LocalDate hireDate;  
    private String name;  
    private int age;  
    @Override  
    public boolean equals(Object ob) {  
        if(ob==null) return false;  
        if(!(ob instanceof Person)) return false;  
        Person p = (Person)ob;  
        return hireDate.equals(p.hireDate)  
            && name.equals(p.name)  
            && age == p.age;  
    }  
    @Override  
    public int hashCode() {  
        return Objects.hash(hireDate, name, age);  
    }  
}
```

Review: Making Your Classes Immutable

1. A class is immutable if the data it stores cannot be modified once it is initialized. Java's String and number classes (such as Integer, Double, BigInteger) are immutable. Immutable classes provide good building blocks for creating more complex objects. **Java 8:** LocalDate, as we saw earlier, is also immutable.
2. Immutable classes tend to be smaller and focused (building blocks for more complex behavior). If many instances are needed, a “mutable companion” should also be created (for example, the mutable companion for String is StringBuilder) to handle the multiplicity without hindering performance.
3. Guidelines for creating an immutable class (from *Effective Java*, 2nd ed.)
 - **All fields should be *private* and *final*.** This keeps internals private and prevents data from changing once the object is created.
 - **Provide *getters* but no *setters* for all fields.** Not providing setters is essential for making the class immutable.
 - **Make the class *final*.** (This prevents users of the class from accessing the internals of the class in another way – to be discussed in Lesson 6.)
 - **Make sure that getters do not return mutable objects.**

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. Inheritance in Java makes it possible for a subclass to enjoy (and re-use) the features of a superclass.
2. All classes in Java – even user defined classes – automatically inherit from the class Object
3. ***Transcendental Consciousness*** is the field of pure awareness, beyond the active thinking level, that is the birthright and essential nature of everyone. Everyone “inherits” from pure consciousness
4. ***Wholeness moving within itself***: In Unity Consciousness, there is an even deeper realization: The only data and behavior that exist in the universe is that which is “inherited from” pure consciousness – everything in that state is seen as the play of one’s own consciousness.

