
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Paul Corazza

Lecture 7: Interfaces in Java 8 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 8 interfaces: Introduction
- Java 8 interfaces: Two Applications of Default Method
- Java 8 interfaces and the Diamond Problem
- FPP Review: Overriding Methods in the Object Class

Java 8 Features of Interfaces

- Before Java 8, none of the methods in an interface had a method body; all were unimplemented.

- In Java 8, two kinds of implemented methods are now allowed: *default methods* and *static methods*. Both can be added to legacy interfaces without breaking code.

- A default method is a fully implemented method within an interface, whose declaration begins with the keyword `default`
- A static method in an interface is a fully implemented static method having the same characteristics as any static method in a class.

See Demos in package lesson7.lecture.defaultmethods and lesson7.lecture.interfacestatic

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 Java Library project]
  Path / Paths.
- For new code, it is now possible to place this static companion code directly in the interface.

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.

Exercise 7.1 – Rewrite List Interface

Explore the package exercise7_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.

Interfaces in Java 7 and Java 8

Interview Question: What is the difference between an abstract class and an interface?

Answer from the perspective of Java 7 (and before)

- Abstract classes may have fully implemented methods, but interfaces may not
- Abstract classes may contain static methods while interfaces may not
- Abstract classes may have instance variables of any kind, whereas interfaces can have only public static final variables
- All methods in an interface are public, but abstract classes may have implemented methods of any visibility and abstract methods that have public, protected, or package-level visibility

Answer from the perspective of Java 8 (and after)

- Abstract classes may have fully implemented instance methods; interfaces may also have implemented instance methods, but they must bear the keyword “default”
- Abstract classes may have instance variables of any kind, whereas interfaces can have only public static final variables
- All methods in an interface are public, but abstract classes may have implemented methods of any visibility and may have abstract methods that have public, protected, or package-level visibility

Outline

- ✓ Java 8 interfaces: Introduction
- ❑ Java 8 interfaces: Two Applications of Default Methods
- ❑ Java 8 interfaces and the Diamond Problem
- ❑ Review: Overriding Methods in the Object Class

Two Applications of Default Methods

First Set of Examples:

`enum`s can now "inherit" from another type

Second Set of Examples:

`forEach` – default method in `Iterable`

First Set of Examples: 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

```java
public class AlignmentConstants {
    /**
     * Indicates that the label should be left justified.
     */
    public static final int LEFT        = 0;

    /**
     * Indicates that the label should be centered.
     */
    public static final int CENTER      = 1;

    /**
     * Indicates that the label should be right justified.
     * @since   JDK1.0t.
     */
    public static final int RIGHT       = 2;
}
```

```java
//extracted from java.awt.Label
//Java library does it the bad way
public class Label {
    private String text;
    private int alignment;
    public Label(String text, int alignment) {
        this.text = text;
        setAlignment(alignment);
    }
    public synchronized void setAlignment(int alignment) {
        switch (alignment) {
        case AlignmentConstants.LEFT:
        case AlignmentConstants.CENTER:
        case AlignmentConstants.RIGHT:
            this.alignment = alignment;
            return;
        }
        throw new IllegalArgumentException(
          "improper alignment: " + alignment);
    }
    public String getText() {
        return text;
    }
    public int getAlignment() {
        return 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

```java
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;
}
```

```java
//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: lesson7.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 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 7.2

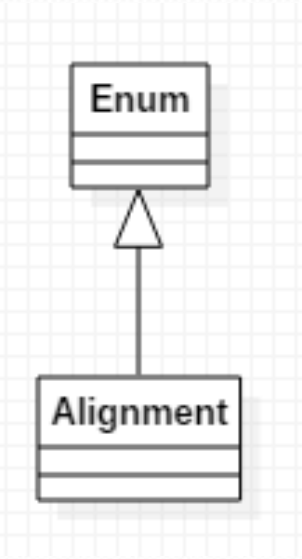
- Below is a `Constants` class consisting of `public final static` variables to provide constants for the rest of the application. Replace with an `enum Const` that provides the same functionality. Refactor the `main` method so that it uses the new `Const` type.

- 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 = 20000000;
}
```

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).



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();
```

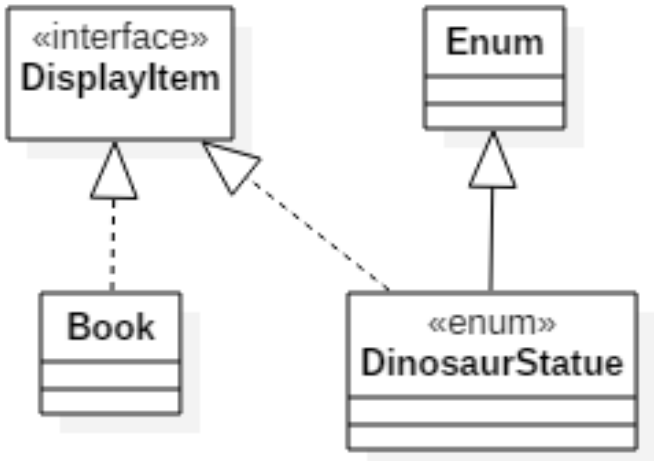
Demo: lesson7.lecture.singletons

In Java 8, Enums Can "inherit"

//from lesson7.lecture.enums3.java8

```
interface DisplayItem {
  default String displayInfo() {
                . . .
  }
}
class Book implements DisplayItem {
}

enum DinosaurStatue implements DisplayItem {
  INSTANCE;
}
```



See lesson7.lecture.enums3.java7 and lesson7.lecture.enums3.java8

Second Set of Examples: forEach

- The `Iterable` interface is part of the Collections API that is implemented by all collection classes, and supports iteration through a collection
- The only method in `Iterable` is `iterator()`, which returns an `Iterator`
- `Iterator` has two methods:
  - `hasNext()`
  - `next()`
- When a class (even user-defined) implements the `Iterable` interface, the "for each" construct can be used (and of course, an instance of `Iterator` is available).

See Demo: `lesson7.lecture.iterator`

New (Java 8) in the Iterable interface is a default method:

`forEach`

Sample usage:

```
Consumer<String> consumer = new Consumer<String>() {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
};
System.out.println("------using new forEach method------");
l.forEach(consumer);
```

Output:

```
------using new forEach method------
Bob
Steve
Susan
Mark
Dave
```

See Demos:
lesson7.lecture.iterator

1. The `forEach` method applies the `Consumer` method `accept` to each element of the list.

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

2. In this example, the `accept` method just prints the value to the console.

3. `Consumer` is an interface introduced in Java 8, with just one abstract method `accept`, which accepts a single argument and produces no return value.

```
interface Consumer<T> {
   void accept(T input);
}
```

Exercise 7.3

- You have a Java `ArrayList` containing multiple elements and an empty MyStringList (from In-Class Exercise 7.1). Use the new Java 8 `forEach` method on the Java list to copy all its elements into the instance of MyStringList.

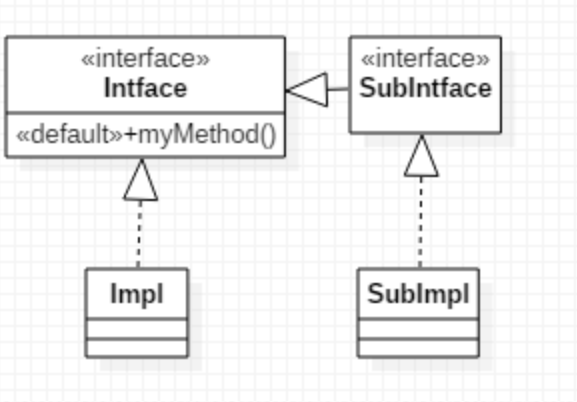
- Startup code is in the exercise_3 package in the InClassExercises project. Test your work by using the `main` method in the ListInfo class.

Outline

- ✓ Java 8 interfaces: Introduction
- ✓ Java 8 interfaces: Two Applications of Default Methods
- ❑ Java 8 interfaces and the Diamond Problem
- ❑ FPP FPP Review: Overriding Methods in the Object Class

Rules for Default Methods in an Interface

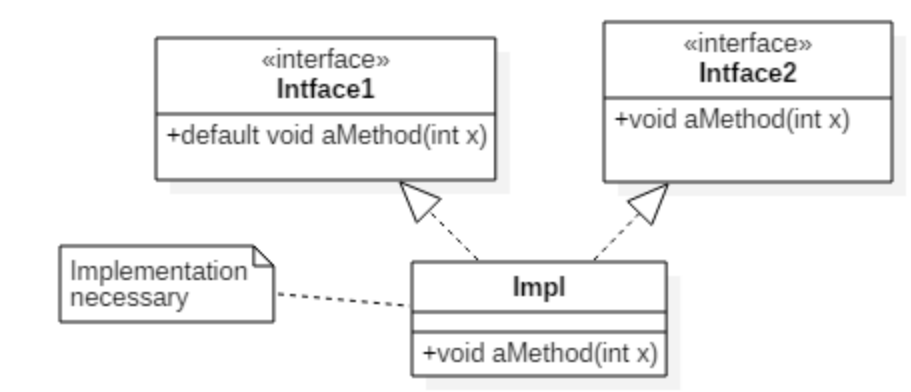
- If a class implements an interface with a default method, that class inherits the default method (or can override it).



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

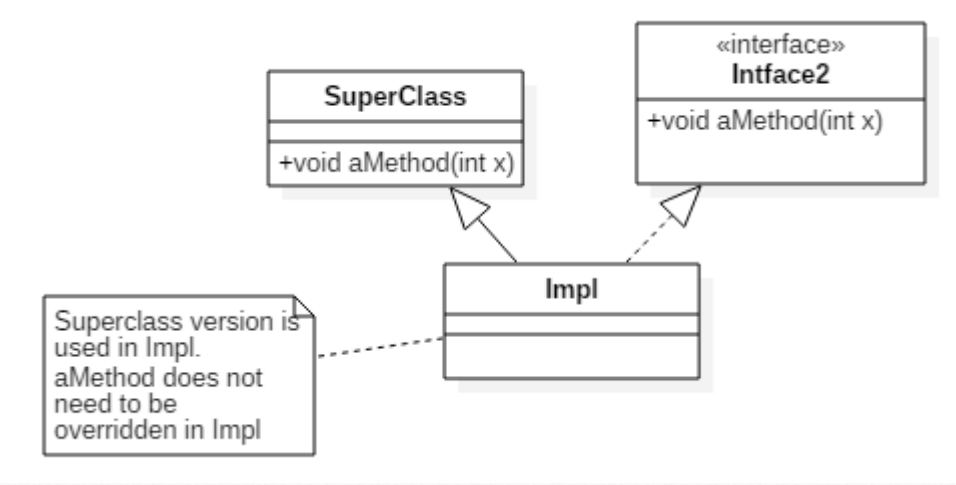
Interface vs Interface – clash! When two interfaces each have a method with the same signature:

- If one of these is a default method, any implementer of both interfaces *must* override the method (or declare it as an abstract method) – can't simply do nothing.
- If one of these is a default method, any *subinterface* of both interfaces must provide a default method (i.e. an implementation) of this method, or declare the method (even if unimplemented).
- Note: Even in Java 7, it is not possible to implement two interfaces each of which has a method with the same signature but different return types.



Superclass vs Interface – superclass wins! When a class extends a superclass and also implements an interface, and both super class and interface have a method with the same signature, the superclass implementation wins – this is the version that is inherited by the class. The subclass/ implementer is not required to override the shared method.

(See Demos in lesson7.lecture.defaultmethodrules)



Static Methods Do Not Clash

- Static methods defined in an interface are *not* inherited by implementers (this differs from the behavior for subclasses of a class)
- Therefore, if two interfaces implement static methods with the same signature, there is no clash to address when a class implements these interfaces.
- Static methods can always be accessed in a static way in such cases, but it is not related to inheritance.

  See demo lesson7.lecture.interfacestatic_clash

Exercise 7.4

Look at the code snippets on the PDF file in lesson7.exercise_4 package of the InClassExercises project. Try to determine, without using a compiler, what happens when the code is compiled/run.

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: For instance, enums can now "inherit" from other types and new public operations can be added to legacy interfaces without breaking code (as was done with the `forEach` method in the `Iterable` interface).

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 8 interfaces: Introduction
- ✓ Java 8 interfaces: Two Applications of Default Methods
- ✓ Java 8 interfaces and the Diamond Problem
- ❑ FPP Review: Overriding Methods in the Object Class

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.

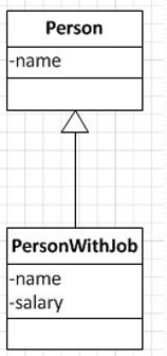
Best Practices Suppose B is a subclass of A.

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 lesson7.lecture.overrideequals.instanceofstrategy3.

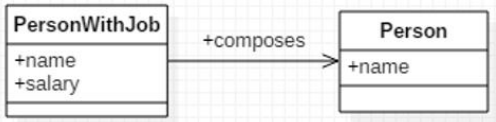
2. If *two different equals methods* are required, two strategies are possible
   A. Use the same classes strategy, but declare subclass B to be final
      See lesson7.lecture.overrideequals.sameclassesstrategy

   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 lesson7.lecture.overrideequals.composition

CHANGE



TO



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.

Refer: `lesson7.lecture.hashcode package`

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.