CS 213 – Software Methodology
Spring 2017

Sesh Venugopal

Lecture 16/17 – Mar 21/23

Default Methods in Interfaces

Default Methods in Interfaces (Java 8)

- Starting with Java 8, interfaces may have *default* methods – a default method is fully implemented. Why the need for default methods?

Default Methods in Interfaces (Java 8) Why?

Library designer ships this interface:

```
public interface Stack<T> {
   void push(T item);
   T pop() throws
       NoSuchElementException;
   boolean isEmpty();
   int size();
   void clear();
}
```

Default Methods in Interfaces (Java 8) Why?

Application builds Stack implementation off this interface:

```
public class MyStack<T>
implements Stack<T> {
    ...
    public void push(T item) {...}
    public T pop() throws
        NoSuchElementException {...}
    public boolean isEmpty() {...}
    public int size() {...}
    public void clear() {...}
}
```

Default Methods in Interfaces (Java 8) Why?

Updated Interface

Interface designer decides to add a peek function:

```
public interface Stack<T> {
    ...
    T peek() throws
        NoSuchElementException;
}
```

Implementer

Implementer installs a new version of library that comes with updated Stack interface (implementer is unaware) – what happens?

The MyStack implementation no longer compiles because the peek method is not implemented

Default Methods in Interfaces (Java 8) Why?

Scenario: Library updates an interface with new functionality. Old code that implements this interface will no longer compile

Application has two choices:

1. Get the updated library binaries and run original implementation without recompiling (binary compatibility)

   Too restrictive, ultimately impractical

2. If other code in application changes, recompiling may be necessary, in which case implement peek, even if it is not needed (source incompatibility)

   Forces application to do unnecessary code rewrite

Default Methods in Interfaces (Java 8) Why?

Solution: Library updates an interface with new functionality.
Old code that implements this interface will no longer compile,
UNLESS interface can provide a default implementation

```
public interface Stack<T> {
    void push(T item);
    T pop() throws NoSuchElementException;
    boolean isEmpty();
    int size();
    void clear();

    default T peek() throws NoSuchElementException {
        T temp = pop();
        push(temp);
        return temp;
    }
}
```

Default Method in Java 8 Library: Example

Prior to Java 8, the way to sort a `List` was to call static method sort in the `java.util.Collections` class, with optionally a `Comparator`

```
List<MyType> list = ...
Comparator<MyType> myComparator = ...
Collections.sort(list, myComparator);
```

In Java 8, the `List` interface has been updated to include a `sort` method so applications can sort a `List` by invoking it directly:

```
list.sort(myComparator);
```

The sort method is declared default (with full implementation) so that legacy code can still compile and run with previous List implementations

Default Methods and Multiple Inheritance

Since interfaces can now implement default methods, what happens if a class implements multiple interfaces that share default methods with the same signature?

```
public interface Lion {
   default void roar() {
      System.out.println
          ("Lion: roar");
   }
}
```

```
public interface Tiger {
   default void roar() {
      System.out.println
          ("Tiger: roar");
   }
}
```

Default Methods and Multiple Inheritance

```
public class Liger implements Lion, Tiger {
   public static void main(String[] args) {
       new Liger().roar();
   }
}
```

Will this code compile?

NO

Default Methods and Multiple Inheritance

```
public interface Lion {
   default void roar() {
      System.out.println
          ("Lion: roar");
   }
}
```

```
public interface Tiger {
   default void roar() {
      System.out.println
          ("Tiger: roar");
   }
}
```

FIX: In `Liger`, override the common method, and have it explicitly call one of the default methods:

```
public class Liger implements Lion, Tiger {
   public void roar() {
       Lion.super.roar();
   }
   public static void main(String[] args) {
       new Liger().roar();
   }
}
```

Default Methods and Multiple Inheritance General Resolution Rules

Rules in order of highest to lowest priority:

1. Classes come first: A method declaration in a class takes priority over a default method declaration in an interface

```
public class Lion {
   public void roar() {
      System.out.println
          ("Lion: roar");
   }
}
```

```
public interface Tiger {
   default void roar() {
      System.out.println
          ("Tiger: roar");
   }
}
```

```
public class Liger extends Lion implements Tiger {
   public static void main(String[] args) {
      new Liger().roar();
   }
}
```

What is printed? `Lion: roar`

Default Methods and Multiple Inheritance General Resolution Rules

2. If there are only interface implementations (no subclassing), then the conflicting default method in the most specific sub-interface is used.

```
public interface Piece {
   default void move() {
      System.out.println
          ("Piece: move");
   }
}
```

```
public interface FlexiblePiece
  extends Piece {
   default void move() {
      System.out.println
          ("FlexiblePiece: move");
   }
}
```

```
public class SlowFlexiblePiece implements FlexiblePiece {
   public static void main(String[] args) {
       new SlowFlexiblePiece().move();
   }
}
```

What is printed? `FlexiblePiece: move`

Default Methods and Multiple Inheritance General Resolution Rules

3. If neither of the previous rules can be applied, then the class implementing the interfaces with the conflicting default methods has to explicitly pick which default method to use by:
   - overriding it
   - calling the desired method (as in the earlier example with `Lion.super.roar()`)