CS 213 – Software Methodology
Spring 2023

Sesh Venugopal

Mar 22

Default Methods in Interfaces

Default Methods in Interfaces

- 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: 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: 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() {...}
}
```

Modification of Interface – Problem

Interface designer decides to add a peek function:

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

Implementer installs a new version of library that has, among other new artifacts, the updated Stack interface, of which they are not aware– what happens?

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

Application choices to work with new interface

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)
Doable, but restrictive – how long can you avoid recompiling?

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 implementation in interface for backward compatibility

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;
    }
}
```

Other interface methods can be called because this code will run in a class that implements the interface

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, because which roar version to call?

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();  // note the syntax!!
   }
   public static void main(String[] args) {
       new Liger().roar();
   }
}
```

Conflict between inherited class and interface methods

```
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();
   }
}
```

Lion: roar

For `Liger` to compile, `roar` method in `Lion` MUST be `public`:
since `Liger` also implements `Tiger`, which has an (implicitly `public`) conflicting default method `roar`