Lecture 21

Default Methods

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

Motivation

Interface vs. Implementation

- Java: Interfaces are a contract
- A class implementing an interface must implement (or inherit) an implementation for each method

Issue: Evolving an Interface

- Adding a method to an interface: now all implementations must add this method, too
- E.g., Java 8 added `sort` to `List`
- Potentially impacts dozens (if not hundreds) of libraries!

New Interface Features in Java 8

- static methods in interfaces
- default methods (implementations in an interface)

Example

`List.sort`

```
default void sort(Comparator<? super E> c){
    Collections.sort(this, c);
}
```

Application

```
List<Integer> numbers = Arrays.asList(3, 5, 1, 2, 6);
numbers.sort(Comparator.naturalOrder());
```

`Comparator.naturalOrder`

- New `static` method in the `Comparator` interface
- returns a `Comparator` object to sort elements in natural order

Another example

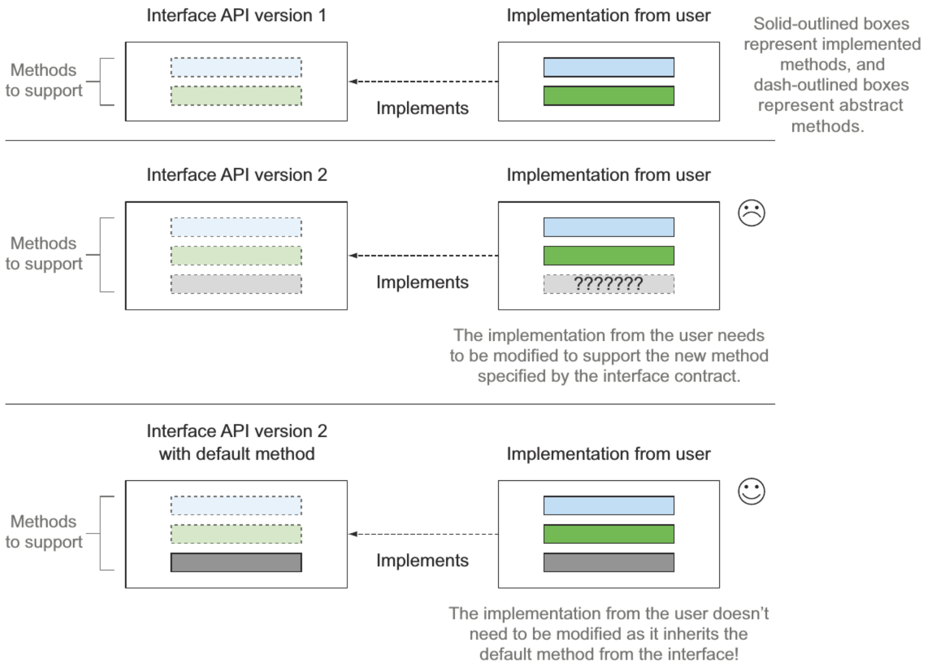
Turning a collection into a stream

```
menu.stream()...
```

stream method for Collection

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

Adding a method to an interface


Interface API version 1
Implementation from user
Methods to support
Implements
Solid-outlined boxes represent implemented methods, and dash-outlined boxes represent abstract methods.
Interface API version 2
Implementation from user
Methods to support
Implements
???????
The implementation from the user needs to be modified to support the new method specified by the interface contract.
Interface API version 2 with default method
Implementation from user
Methods to support
Implements
The implementation from the user doesn't need to be modified as it inherits the default method from the interface!

Java 8: Default Methods and Static Interface Methods

Default Methods

- Interfaces can now include implementations in form of default methods
- Makes it easier to evolve an API:
  - Add new method to interface
  - Also provide implementation in form of a default method
- Important new feature for library designers

Static methods

- Common Java design pattern: Interface class with corresponding utility companion class providing `static` helper methods
- For example, `Collection` interface and `Collections` companion class
- Now these static helper methods can go directly into interface

Outline

Design of a Java drawing library

Resizable interface

- Support methods like `getHeight()`, `setHeight()`
- Implementations for some shapes, like `Square` or `Rectangle`

First version

```
public interface Resizable extends Drawable {
  int getWidth();
  int getHeight();
  void setWidth(int width);
  void setHeight(int height);
  void setAbsoluteSize(int width, int height);
}
```

Users of our interface create additional classes implementing these methods

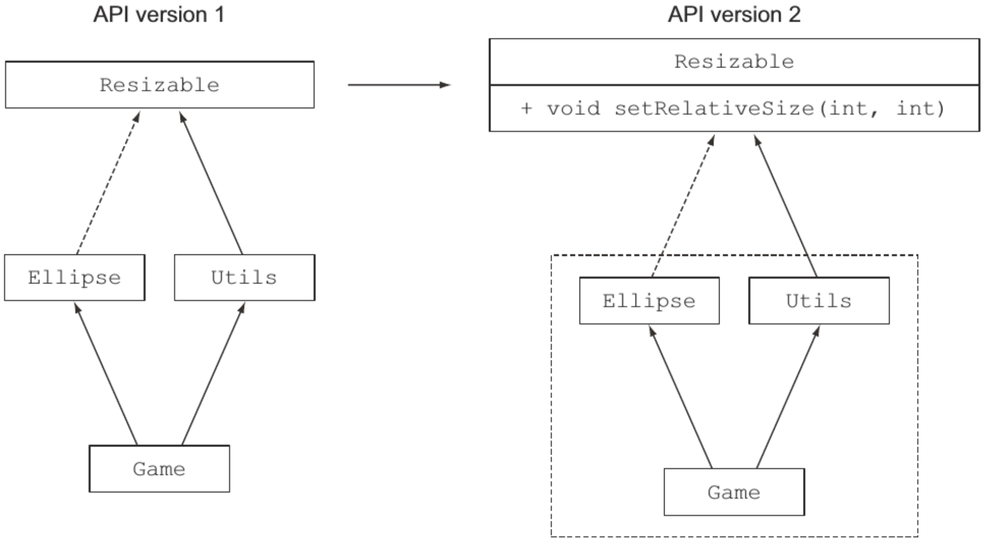
```
public class Ellipse implements Resizable {
  ...
}
```

Using the interface

```
public class Game {
  public static void main(String...args) {
    List<Resizable> resizableShapes =
      Arrays.asList(new Square(), new Rectangle(), new Ellipse());
    Utils.paint(resizableShapes);
  }
}

public class Utils {
  public static void paint(List<Resizable> l) {
    l.forEach(r -> {
      r.setAbsoluteSize(42, 42);
      r.draw();
    });
  }
}
```

API version 2

Adding new `setRelativeSize` method



Updated Interface

```
public interface Resizable {
  int getWidth();
  int getHeight();
  void setWidth(int width);
  void setHeight(int height);
  void setAbsoluteSize(int width, int height);
  void setRelativeSize(int wFactor, int hFactor);
}
```

Issues

- Class implementing `Resizable` now must implement `setRelativeSize`
- Code for the `Ellipse` class will no longer compile

```
Ellipse.java:6: error: Ellipse is not abstract and does
    not override abstract method setRelativeSize(int,int)
    in Resizable
```

- Our API is not backwards compatible

Compatibility in Java

Different types of compatibilities

Binary compatibility: existing binaries continue to link without error (e.g., adding a method to an interface)

Source compatibility: existing programs will still compile after the change

Behavioral compatibility: running a program after a change results in same behavior

See https://blogs.oracle.com/darcy/kinds-of-compatibility:-source,-binary,-and-behavioral

Default methods to the rescue

Java 8

- Evolve API by adding new method to interface
- and providing a default implementation

Example

```
default void setRelativeSize(int wFactor, int hFactor){
  setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
}
```

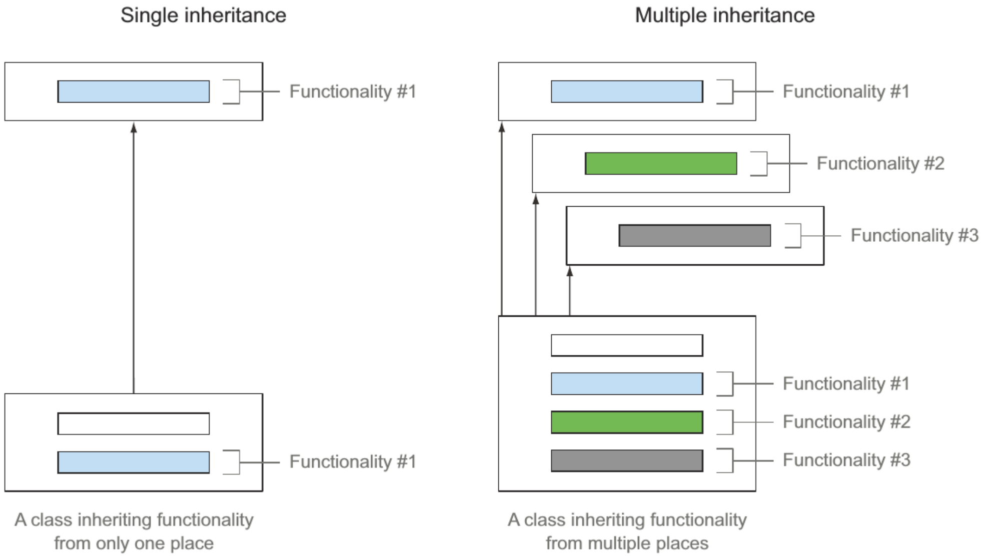
Outline

1. Introduction
2. Evolving APIs
3. **Usage Patterns**
   - Optional methods
   - Multiple inheritance
   - Composing Interfaces
4. Resolution Rules
5. Summary
6. Notes and Further Reading

Optional methods

```
interface Iterator<T> {
  boolean hasNext();
  T next();
  default void remove() {
    throw new UnsupportedOperationException();
  }
}
```

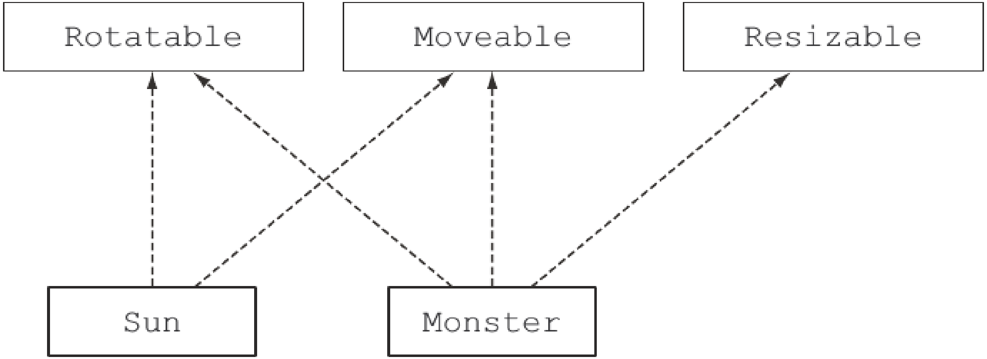
Multiple inheritance of behavior



Implementing multiple interfaces

```java
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable,
               Serializable, Iterable<E>, Collection<E> {
}
```

Composing Interfaces



```
public class Monster implements Rotatable, Moveable, Resizable {
   ...
}
```

Composing Interfaces (II)

```
public interface Rotatable {
  void setRotationAngle(int angleInDegrees);
  int getRotationAngle();
  default void rotateBy(int angleInDegrees){
    setRotationAngle((getRotationAngle () + angle) % 360);
  }
}

public interface Moveable {
  int getX();
  int getY();
  void setX(int x);
  void setY(int y);
  default void moveHorizontally(int distance){
    setX(getX() + distance);
  }
  default void moveVertically(int distance){
    setY(getY() + distance);
  }
}
```

Composing Interfaces (III)

```
public interface Resizable {
  int getWidth();
  int getHeight();
  void setWidth(int width);
  void setHeight(int height);
  void setAbsoluteSize(int width, int height);
  default void setRelativeSize(int wFactor, int hFactor){
    setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
  }
}
```

Composing Interfaces (IV)

```
Monster m = new Monster();
m.rotateBy(180);            // from Rotatable
m.moveVertically(10);       // from Moveable
```

Outline

Resolution Rules

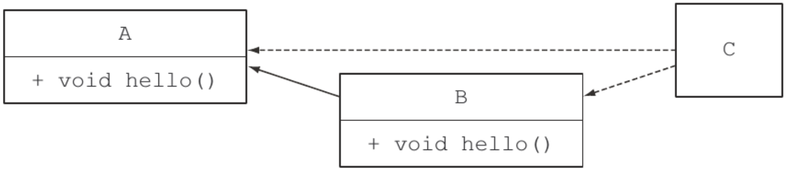
```
public interface A {
  default void hello() {
    System.out.println("Hello␣from␣A");
  }
}

public interface B extends A {
  default void hello() {
    System.out.println("Hello␣from␣B");
  }
}

public class C implements B, A {
  public static void main(String... args) {
    new C().hello();
  }
}
```

Three resolution rules to know

Rules to follow when a class inherits a method from multiple places

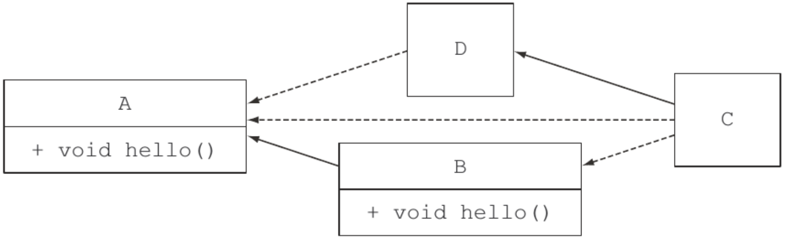
1. Classes always win. A method declaration in the class or a superclass takes priority over any default method declaration.
2. Otherwise, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected. (If B extends A, B is more specific than A).
3. Finally, if the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly.

Applying the Rules



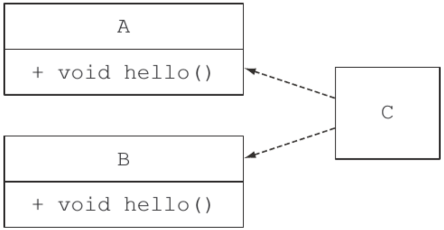
Applying the Rules (II)

```
public class D implements A{ }

public class C extends D implements B, A {
  public static void main(String... args) {
    new C().hello();
  }
}
```



Conflicts and explicit disambiguation

```
public interface A {
  void hello() {
    System.out.println("Hello␣from␣A");
  }
}

public interface B {
  void hello() {
    System.out.println("Hello␣from␣B");
  }
}

public class C implements B, A { }
```



Error

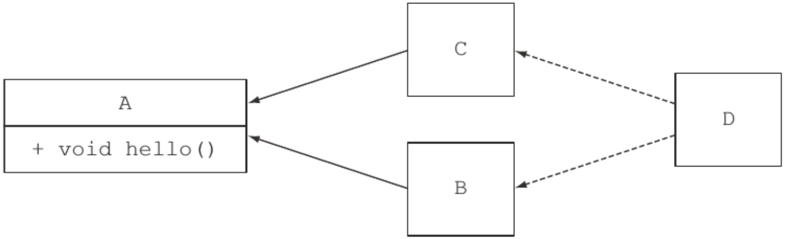
```
class C inherits unrelated defaults for hello() from types B and A.
```

Resolving the Conflict

```
public class C implements B, A {
  void hello(){
    B.super.hello();
  }
}
```

The Diamond Problem

```
public interface A{
  default void hello(){
    System.out.println("Hello␣from␣A");
  }
}

public interface B extends A { }

public interface C extends A { }

public class D implements B, C {
  public static void main(String... args) {
    new D().hello();
  }
}
```



Summary

Default Methods

- Interfaces in Java 8 can have implementation code through default methods and static methods.
- Default methods start with a `default` keyword and contain a body like class methods do.
- Adding an abstract method to a published interface is a source incompatibility.
- Default methods help library designers evolve APIs in a backward-compatible way.
- Default methods can be used for creating optional methods and multiple inheritance of behavior.
- There are resolution rules to resolve conflicts when a class inherits from several default methods with the same signature.
- A method declaration in the class or a superclass takes priority over any default method declaration. Otherwise, the method with the same signature in the most specific default-providing interface is selected.
- When two methods are equally specific, a class must explicitly override a method and select which one to call.

Outline

Reading Material

Required

- [UFM14, Chapter 9] (Default methods)

Supplemental

- [War14, Chapter 4] (Libraries: Default Methods)

References

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.

[War14] Richard Warburton.
Java 8 Lambdas.
O'Reilly, 2014.