Default and Static Methods on Interfaces

Richard Warburton
Raoul-Gabriel Urma

Module Outline

1. Motivation for default methods
2. Useful default methods and resolution rules
3. Patterns for default methods
4. Static methods in interfaces

Motivation for default methods

Evolution of APIs

- As code matures new features are added to libraries
- Often makes sense to add new methods to interfaces
- Classes implementing interfaces now all have to change
- Difficult when others use your API and implement their own classes

Collection Problem

- Several libraries use the `Collection` interface, e.g. Guava, Apache Commons
- Streams and Java 8 features modify the `Collection` interface
- Heavily break backwards compatibility at compilation time in Java 8

Quiz

- What options are there to solving the problem of evolving interfaces?
- What techniques have you applied in the past?
- Does this scale where other teams implement your interfaces?

Default Methods

- Introduced to resolve backwards compatibility in JDK
- Allows better API evolution for users of Java
- Interfaces can now define behaviour

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

Default Methods

- Identified by using the `default` keyword
- Useful for defining simple behaviour that can be overridden
- Eg `forEach` on `Iterable<T>`

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

What’s an interesting property about interfaces?

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>,
                   RandomAccess,
                   Cloneable,
                   java.io.Serializable

abstract class AbstractList<E> extends AbstractCollection<E>
         implements List<E>

abstract class AbstractCollection<E> implements Collection<E>

interface Collection<E> extends Iterable<E>
```

Resolution rules

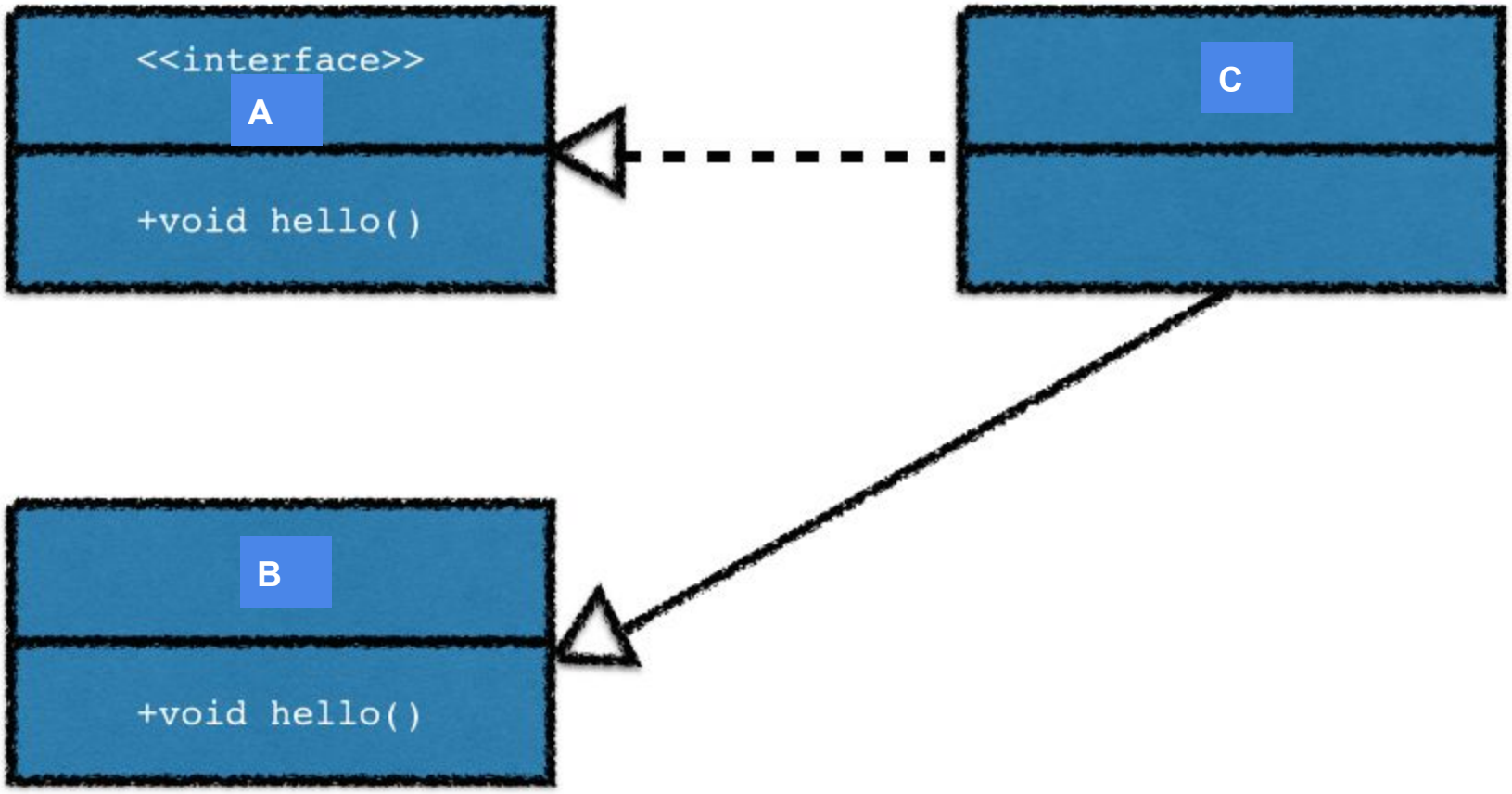
Resolution Rules

1. Classes always win

2. Otherwise, subinterfaces win. Method with the same signature in most specific interface is selected

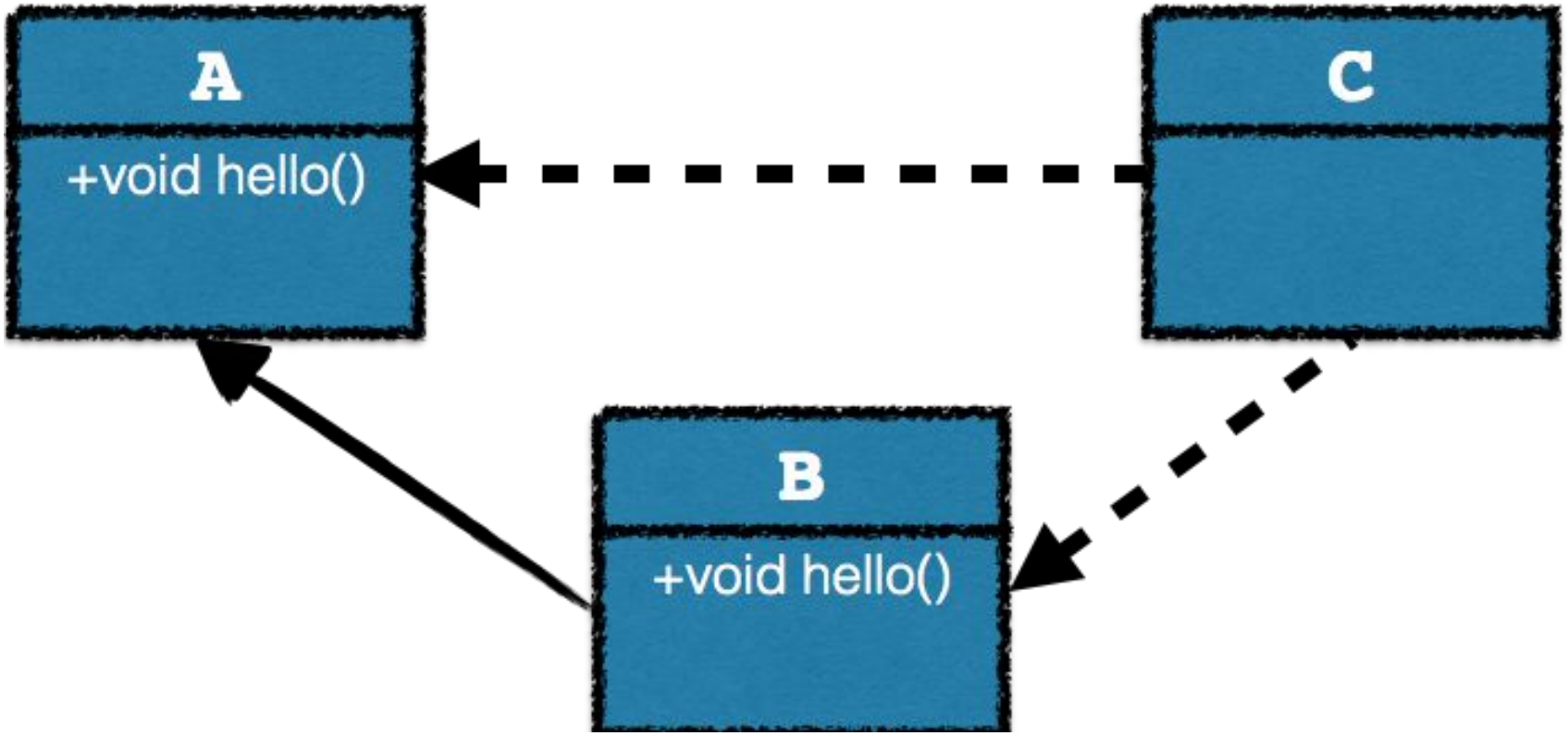
3. If the choice is still ambiguous, the class inheriting must be explicit

Rule 1



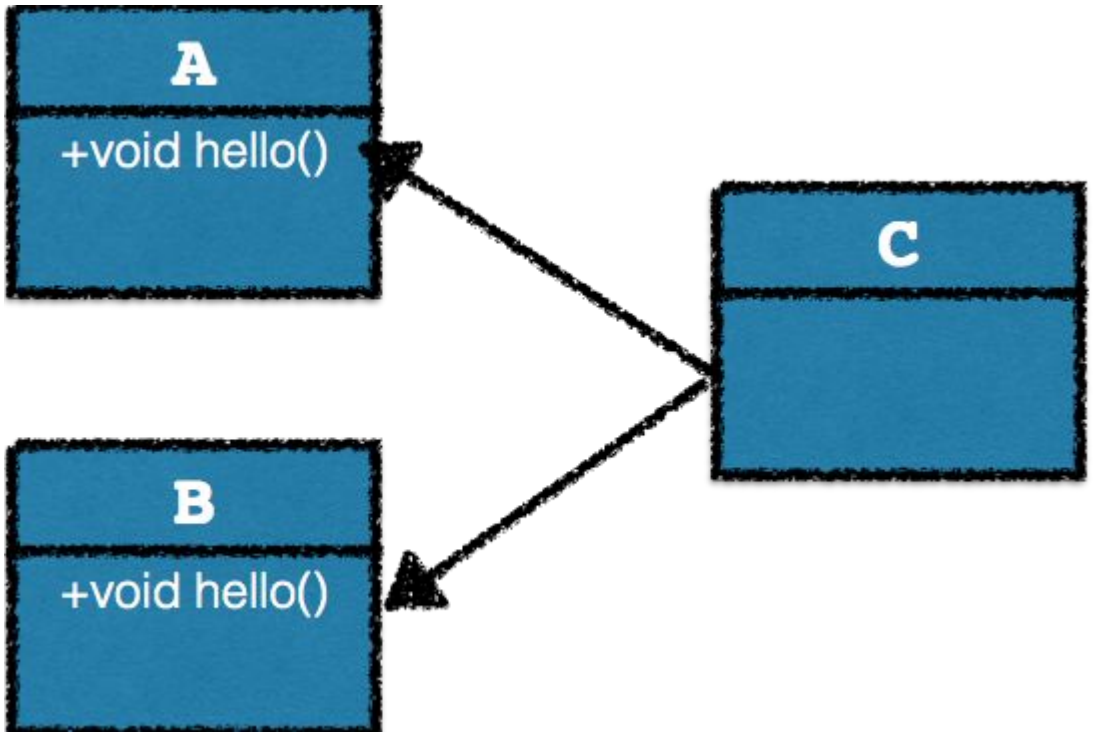
Classes always win

Rule 2



Otherwise, subinterfaces win. Method with the same signature in most specific interface is selected

Rule 3



- If neither rule 1 or 2 can be applied there is ambiguity
- Must explicitly use `super` and select implementation

```
A.super.hello()
```

Quiz - what is printed?

```
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 D implements A {
    public void hello() {
       System.out.println("Hello from D");
    }
}
public class C extends D implements B, A {
    public static void main(String... args) {
       new C().hello();
    }
}
```

Patterns for default methods

Abstract Class vs Interface

Feature	Abstract Class	Interface
Abstract Methods	✓	✓
Behaviour	✓	✓
Implement Multiple	×	✓
Instance Variables	✓	×
Protected/Package Scoped Methods	✓	×
(**Java 9**) Private methods	✓	✓
Static methods	✓	✓

Optional Methods

- Classes sometimes don’t need all methods from an interface. E.g. `remove()` on `Iterator`
- Leads to unnecessary boilerplate code
- Can now provide optional methods by giving a default implementation

Behaviour Inheritance

- Default methods enable multiple inheritance of behaviour
- Can be used to define small subsets of characteristics you want objects to adhere to
- Can be used to apply a form of template design pattern
- See `Movable/Resizable` example

Inheritance Warning

- Inheritance should be used sparingly
- Inheritance means “Is-A”
- If used incorrectly `default` methods can become a strong anti pattern by:
  - Coupling code at multiple levels
- Keep interfaces **small** and **specific**

Remaining Incompatibility

There's still a potential error if our implementing class had a method with the same name and parameter types, but different return type.

```
default void sort(Comparator<? super E> c);
```

vs

```
default List<E> sort(Comparator<? super E> c);
```

Static methods in interfaces

Static Methods

- Method that is on the **class** level
- No access to member variables or state
- Useful for helper methods, e.g. Sorting

```
static <T extends Comparable<? super T>> void
sort(List<T> list) {
   Object[] a = list.toArray();
   Arrays.sort(a);
   ListIterator<T> i = list.listIterator();
   for (int j=0; j<a.length; j++) {
      i.next();
      i.set((T)a[j]);
   }
}
```

Static Methods

- A pattern in Java has been to have companion classes
- `Collection` and `Collections` is an example
- The companion class is often a collection of useful methods
- Can now define static methods on interfaces

Summary

Summary

- Defaults methods are a new tool to evolve APIs in a compatible way
- Ambiguities are always compile time resolved.
- Defaults methods also enhance the available Java software design options.
- Static methods on interfaces also available

Exercises

`com.java_8_training.problems.defaultmethods.DefaultMethodsLabTest`

The End

Composition of Interfaces

- Composing interfaces now composes *behaviour*
- Classes doesn't need to boiler plate a simple implementation, but can override it if necessary
- If the mechanism changes for the default behaviour all classes now adhere to that change
- Solves problems in Java that previously required more boilerplate code due to lack of multiple inheritance