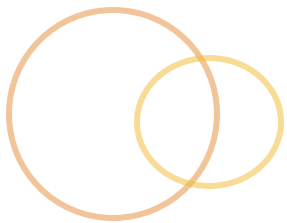
Static and Default Methods In Interfaces

Behavior for the behaviorless

Objectives



- Describe the purpose, and general behavior, of default methods in interfaces
- Outline the potential problem of multiple inheritance with default methods, and the approach Java 8 uses to address this
- Create and use static methods in interfaces

Upgrades To Interfaces In Java 8

- Prior to Java 8, interfaces only declare **`abstract`** methods and **`public static final`** data items
- Java 8 allows two concrete method types to be defined in interfaces:
  - **`default`** methods
  - **`static`** methods

What Are Default Methods For?

- Default methods are primarily about “interface evolution”
  - Allowing new requirements to be added to an interface without breaking existing implementation
  - Several details of default methods reflect this

What Are Default Methods For?

- Java 8 adds many new features to the collections API, adding behavior to key interfaces such as **`List`**
  - This would have broken existing implementations of those interfaces
  - **`default`** methods allow the interface itself to define a method implementation that will be inherited by the existing implementations, but may be overridden by the class implementing the method if desired
    - This is just like a concrete method in an abstract class, except that multiple interface inheritance still applies
    - Allows behavior required by an upgraded interface to be provided to existing, not-yet-upgraded, implementations

What Are Default Methods For?

- Default methods can provide generalized behavior for a whole category of implementations
  - Similar to abstract classes with base methods
  - Implementation typically written in terms of the interface's instance methods, but can use other accessible behavior

Implementing Default Methods

- Implementation of default methods is simple

```
interface W {
    default void doStuff() {
        System.out.println("X.doStuff()");
    }
}
```

- Modifier `default` is used
- Modifier `abstract` must not be used
- Method body (rather than semicolon) is required

Default Methods And Multiple Inheritance

- Interfaces were originally a solution to the perceived ugliness of multiple inheritance
- Default methods would appear to re-introduce this problem, but reasonable rules apply to disambiguate
  - Remember, however, default methods are generally used to get out of an API upgrade problem, so the rules and concerns should not be a major concern

Disambiguating Multiple Inheritance

- Concrete class methods ***always*** override default methods
  - Normal inheritance rules apply to the concrete inheritance tree

Disambiguating Multiple Inheritance

- Multiple inheritance of default methods gets the "nearest" definition

```
interface X {
default void doStuff() {…}
}
```

Y is "nearer" than X (X is two levels of inheritance away)

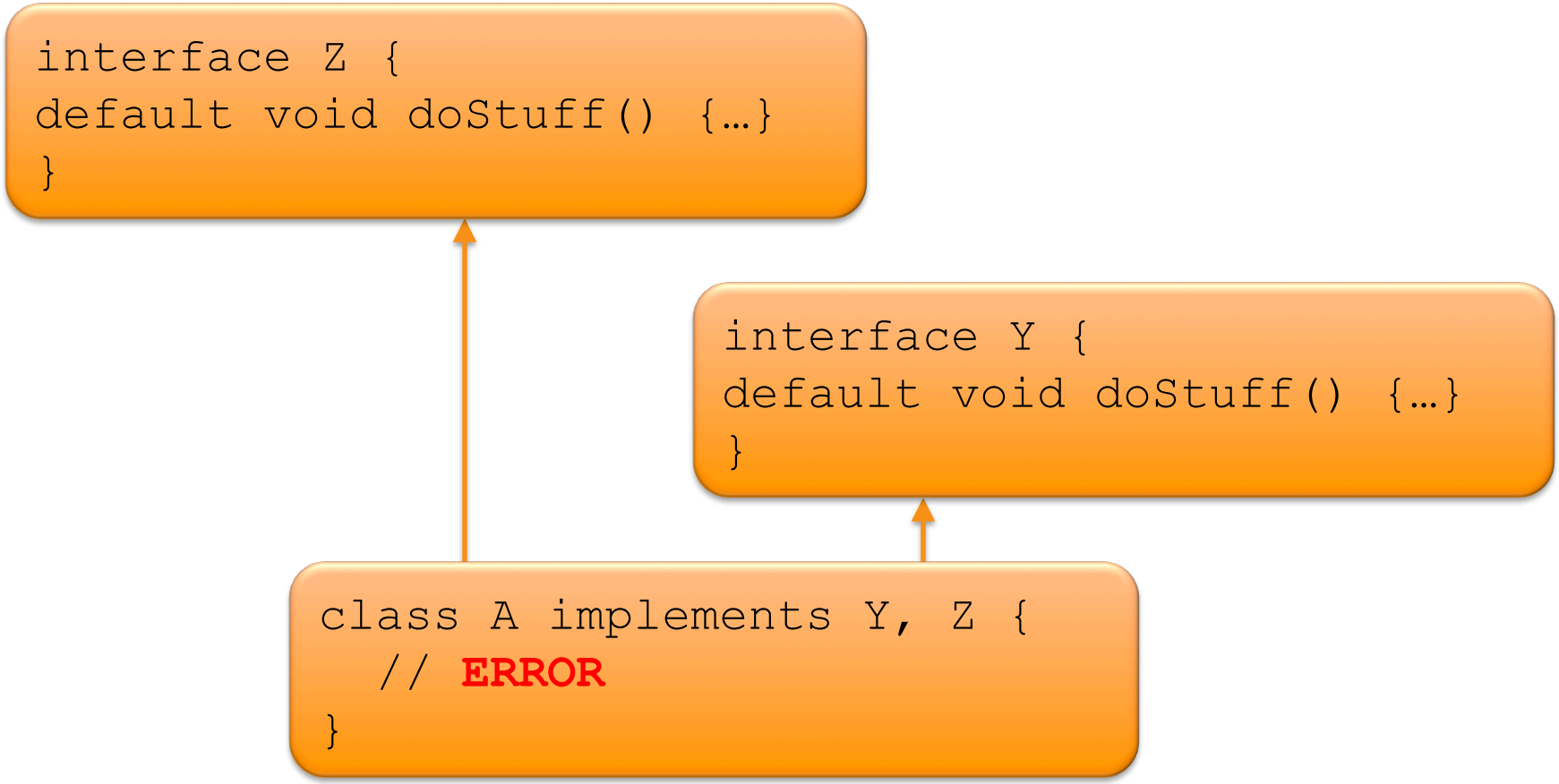
```
interface Z
  extends X {
}
```

```
interface Y {
default void doStuff() {…}
}
```

```
class A implements Y, Z {
  // doStuff from Y
}
```

Disambiguating Multiple Inheritance

- Equivalent default methods inherited from interfaces at the same “distance” cause an error



Resolving Conflicting Defaults

- If a class attempts to inherit conflicting default methods, an explicit override can be used, which can select a specific implementation using an explicit invocation

```
Class MyClass
  implements AnInterface, AnotherInterface
{
  @Override
  public void aMethod() {
    AnInterface.super.aMethod();
  }
}
```

Static Methods In Interfaces

- Since Java 8, interfaces may have `static` method definitions, much the same as for abstract classes
- Very useful for providing general utilities applicable for the concept represented by the interface
  - Compare with utility methods of `Collections` class
- Should be coded in terms of generally accessible behaviors
  - Normal interface methods are instance methods, not accessible in a static context

Defining Static Methods

- Method definition is the same as for any other class

```
public interface Nameable {
   String getName();
   void setName(String name);

   static boolean sameName(Nameable n1, Nameable n2){
      return n1.getName().equals(n2.getName());
   }
}
```

Invoking Static Methods

- Static methods declared in an interface belong ***only*** to that interface; they can only be invoked on that interface, not on implementing classes
- Given

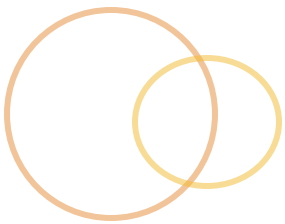
```
interface IntX { static doStuff() {…} }
class ClassY implements IntX {…}

IntX.doStuff(); // OK
ClassY.doStuff(); // Compiler error
```

- This differs from static methods in classes which can, but shouldn't be, referred to in a subclass

Lab Exercise

- Define an interface `Addressable`, which identifies a type that has properties name, street, city, and zip, with suitable accessor and mutator (get and set) methods
- Define a static method in this interface. The method should return a textual representation, with new-line characters, of an address label for the `Addressable` object

Lab Exercise



- Define classes `Customer` and `Supplier`, both of which implement `Addressable`.
- In a `main` method, create instances of `Customer` and `Supplier`, and print out addresses for each