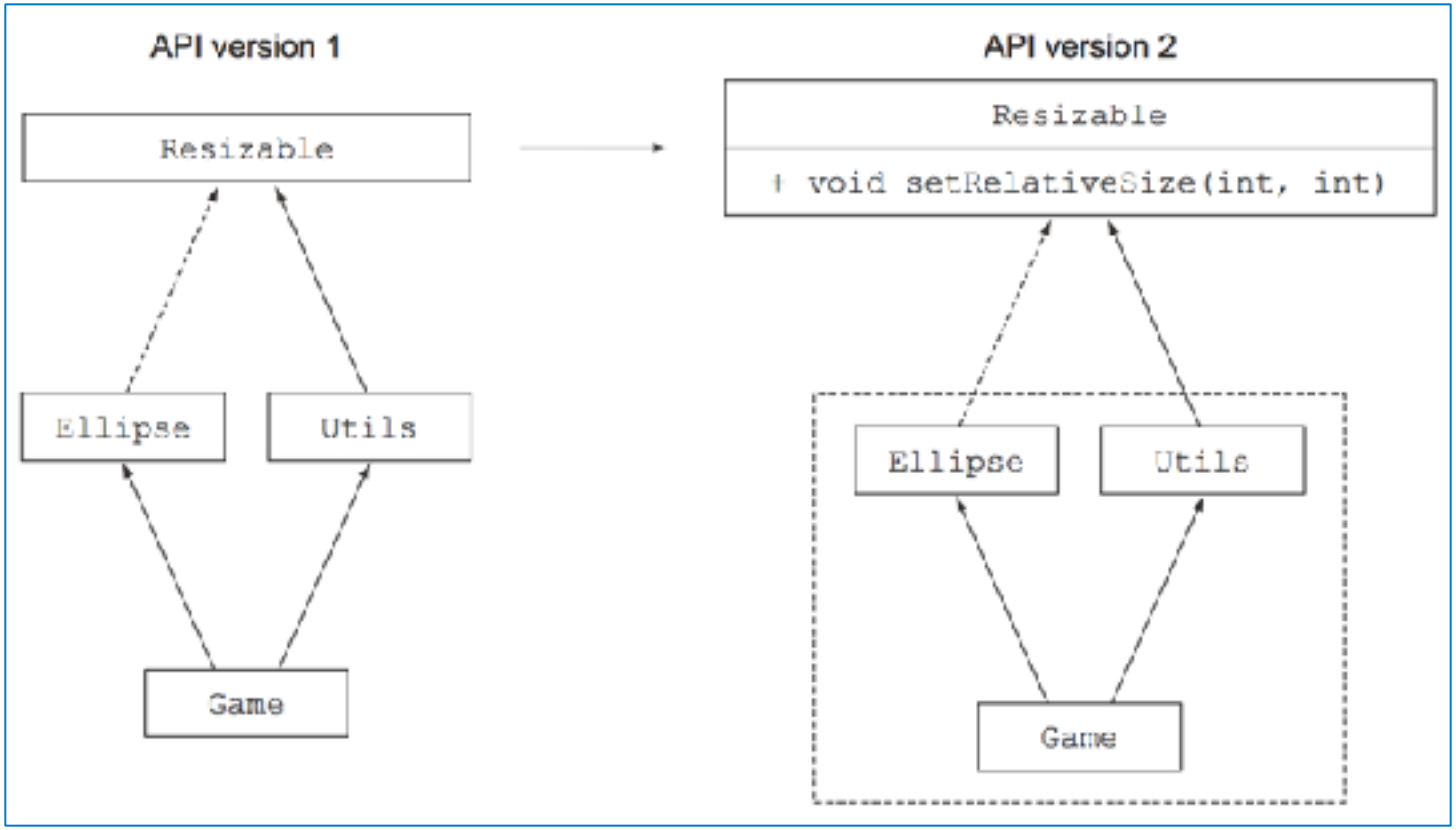
Default methods

Interfaces revisited in Java 8

```
public interface Action {

    static void doSomeStuff() {
        System.out.println("This can't be true");
    };

    default void doAction() {
        System.out.println("What is this default keyword?");
    }

}
```

- In Java 8 the above interface compiles nicely
- static methods are starting from Java 8 allowed on interfaces
- A default method is introduced to be backwards compatible

evolution of an API

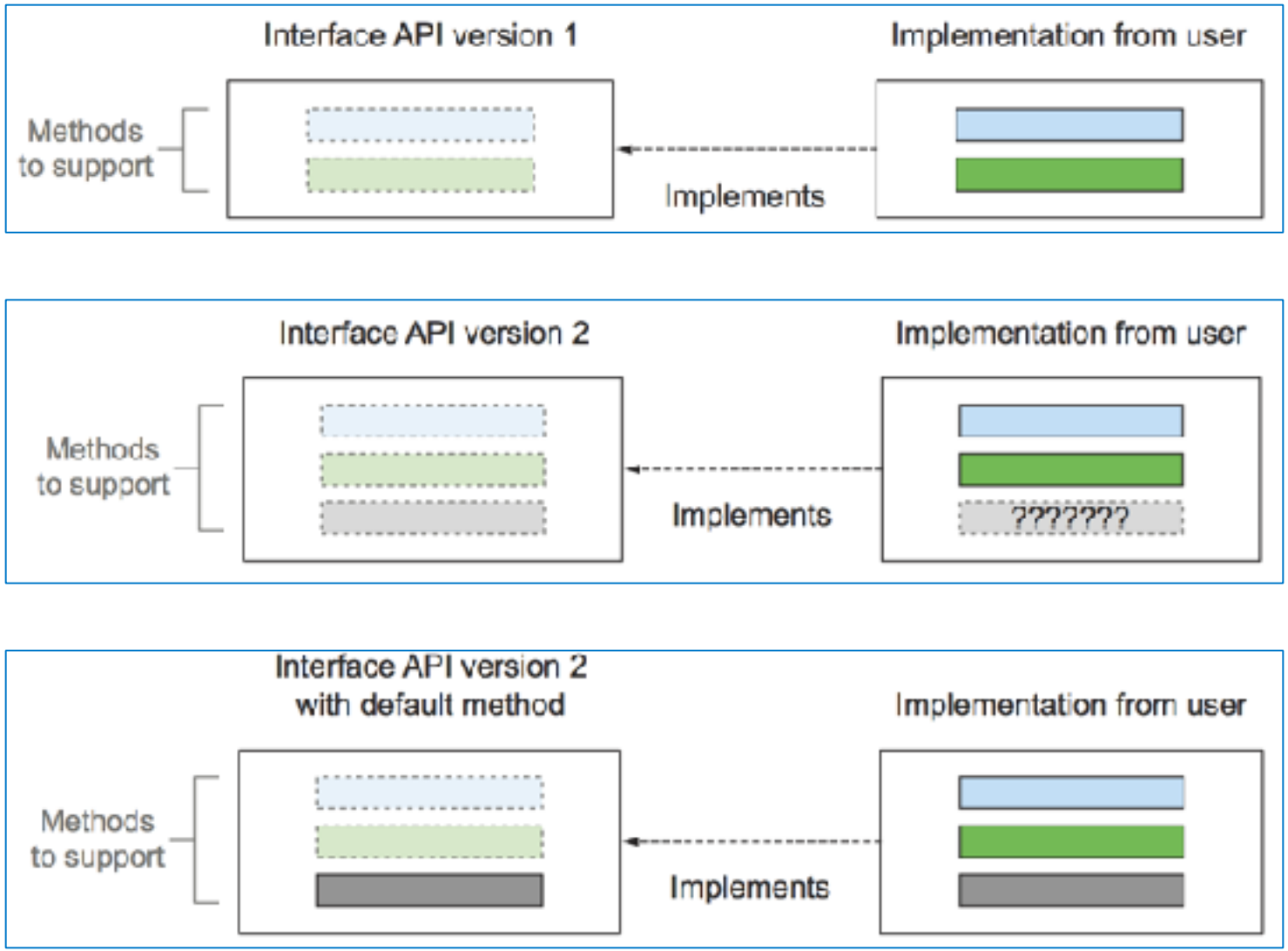


Ellipse and Utils depend on v1, recompiling against v2 results in exception

about default methods

- default methods allow you to provide a default implementation for methods in an interface
- existing classes implementing this interface will inherit the default implementations if they don’t provide one themselves (and they probably don’t)

in pictures


Interface API version 1
Implementation from user
Methods to support
Implements
Interface API version 2
Implementation from user
Methods to support
Implements
???????
Interface API version 2 with default method
Implementation from user
Methods to support
Implements

Different types of compatibilities

- binary
  - existing binaries running without errors continue to link (which involves verification, preparation, and resolution) without error after introducing a change
  - i.e. adding a method to an interface; because if it's not called, existing methods of the interface can still run
- source
  - an existing program will still compile after introducing a change
- behavioural
  - running a program after a change with the same inputs results in the same behaviour

2 Usage patterns for default methods

- optional methods and
- multiple inheritance of behaviour

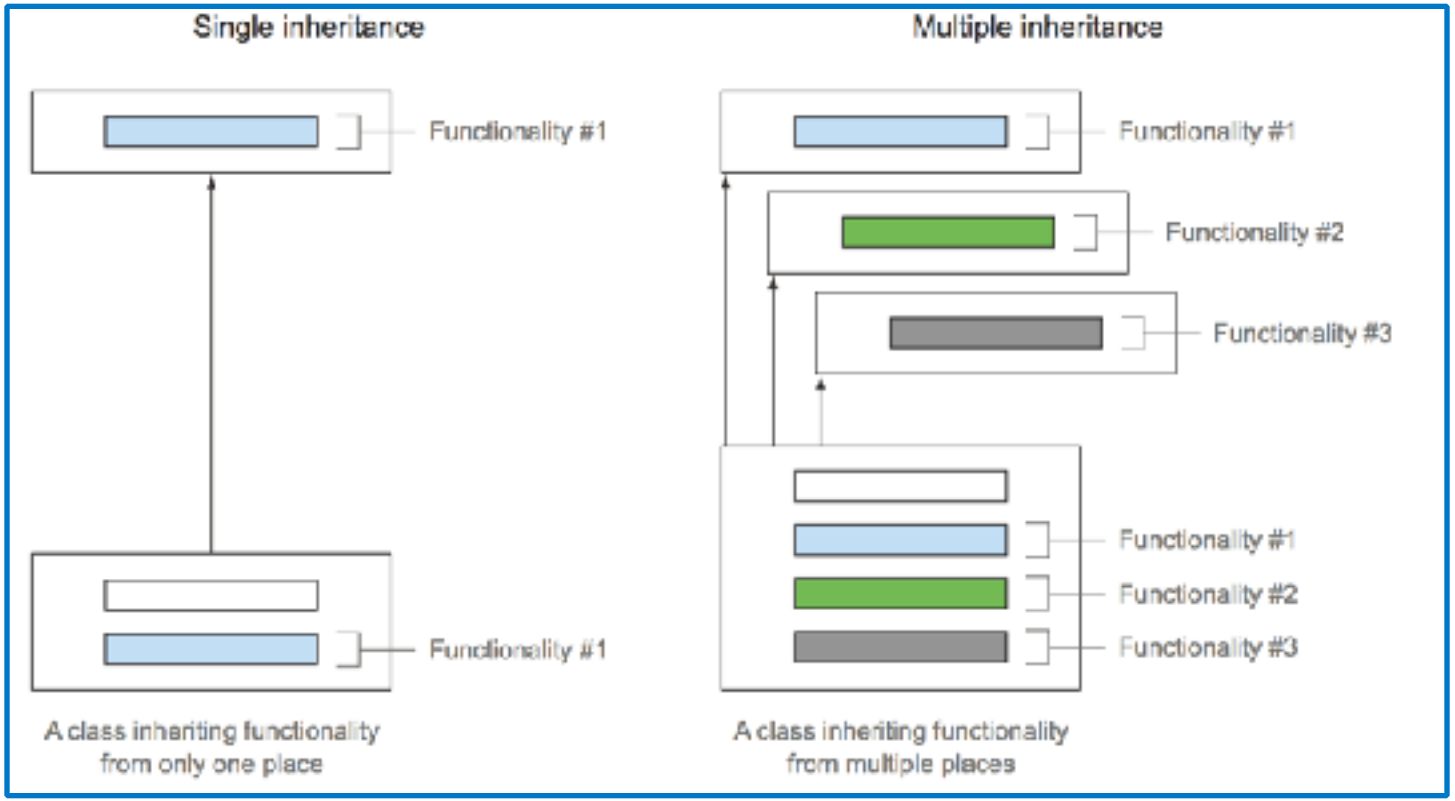
- About optional methods
  - use case: a class implements an interface but some methods are left empty
  - Java 8 solution: provide a default implementation for such methods
  - concrete classes don’t need to explicitly provide an empty implementation

2 Usage patterns for default methods

- optional methods and
- multiple inheritance of behaviour

- About multiple inheritance of behaviour
  - This is the ability of a class to reuse code from multiple places
  - classes can inherit from only one other class, but they can implement multiple interfaces
  - this opens up interesting possibilities

multiple inheritance of behaviour



MINIMAL INTERFACES WITH ORTHOGONAL FUNCTIONALITIES

- Define simple Interfaces around a distinct task
  - call them X,Y and Z
  - define abstract methods (get/setters) in X,Y and Z to parameterize the task
  - define a default method in the interface that executes the task
  - classes can “magically” acquire behaviour by implementing such an interface.

MINIMAL INTERFACES WITH ORTHOGONAL FUNCTIONALITIES

- optional methods and
- multiple inheritance of behaviour

- about multiple inheritance of behaviour
  - this is the ability of a class to reuse code from multiple places
  - classes can inherit from only one other class, but they can implement multiple interfaces
  - this opens up interesting possibilities

Diamond problem in Java 8 ?

- Consider the following setup of interfaces A & B and classes C and D

```
interface A {
	default void hello() {
		System.out.println("hello from A");
	}
}
interface B extends A {
	default void hello() {
		System.out.println("hello from B");
	}
}
class C implements A{
	public void hello() {
		System.out.println("hello from C");
	}
}
class D  extends C implements A, B {
}
public class InterfacePrecedenceTest {
	@Test
	public void whichHelloIsPrinted() {
		new D().hello();
	}
}
```

- Which implementation is chosen?
- hello from C

Diamond problem in Java 8 ?

- Three resolution rules to address this problem
  1. Classes always win: a method in the class or superclass takes priority over any default method declaration
  2. Next, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected
  3. Finally, if the choice is still ambiguous, the class has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly

Diamond problem in Java 8 ?

- Consider the following setup of interfaces A & B and class D

```
interface A {
	default void hello() {
		System.out.println("hello from A");
	}
}
interface B extends A {
	default void hello() {
		System.out.println("hello from B");
	}
}
class D implements A, B {
}
public class InterfacePrecedenceTest {
	@Test
	public void whichHelloIsPrinted() {
		new D().hello();
	}
}
```

- Which implementation is chosen?
- Note class C is removed
- hello from B

Diamond problem in Java 8 ?

- Consider the following setup of interfaces A & B and class D
- Which implementation is chosen?
- Note B does not extend A anymore
- Compilation error

```
interface A {
      default void hello() {
            System.out.println("hello from A");
      }
}
interface B {
      default void hello() {
            System.out.println("hello from B");
      }
}
class D implements A, B {
}
public class InterfacePrecedenceTest {
      @Test
      public void whichHelloIsPrinted() {
            new D().hello();
      }
}
```

Diamond problem in Java 8 ?

- Consider the following setup of interfaces A & B and class D
- Which implementation is chosen?
- Note B does not extend A anymore
- We have to explicitly say which interface to use

```
interface A {
	default void hello() {
		System.out.println("hello from A");
	}
}
interface B {
	default void hello() {
		System.out.println("hello from B");
	}
}
class D  implements A, B {

	public void hello(){
		B.super.hello();
	}
}
public class InterfacePrecedenceTest {
	@Test
	public void whichHelloIsPrinted() {
		new D().hello();
	}
}
```