Static and Default Methods in Java 8 Interfaces

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

λ

coreservlets.com – custom onsite training

For customized training related to Java or JavaScript, please email hall@coreservlets.com

Marty is also available for consulting and development support

The instructor is author of several popular Java EE books, two of the most popular Safari videos on Java and JavaScript, and this tutorial.

Courses available at public venues, or
custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JSF 2.3, PrimeFaces, Java programming (using Java 8, for those new to Java), Java 8 (for Java 7 programmers), JavaScript, jQuery, Angular 2, Ext JS, Spring Framework, Spring MVC, Android, GWT, custom mix of topics.
  - Java 9 training coming soon.
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Spark, Hibernate/JPA, HTML5, RESTful Web Services

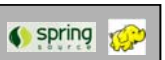
Contact hall@coreservlets.com for details

Topics in This Section

- Static methods
- Examples
- Default methods
- Examples
- Resolving conflicts with default methods





Static Methods in Interfaces

Big Idea

- **Static methods in interfaces**
  - Java 7 and earlier
    - No
  - Java 8 and later
    - Yes
  - New rules violate the spirit of interfaces?
    - No (arguably)
- **Concrete (default) methods in interfaces**
  - Java 7 and earlier
    - No
  - Java 8 and later
    - Yes
  - New rules violate the spirit of interfaces?
    - Yes (arguably)

Java 8: Interfaces and Abstract Classes

	Java 7 and Earlier	Java 8 and Later
Abstract Classes	• Can have concrete methods and abstract methods • Can have static methods • Can have instance variables • Class can directly extend one	(Same as Java 7)
Interfaces	• Can only have abstract methods – no concrete methods • Cannot have static methods • Cannot have mutable instance variables • Class can implement any number	• Can have concrete (default) methods and abstract methods • Can have static methods • Cannot have mutable instance variables • Class can implement any number

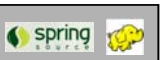
Conclusion: there is little reason to use abstract classes in Java 8. Except for instance variables, Java 8 interfaces can do everything that abstract classes can do, plus are more flexible since classes can implement more than one interface. This means (arguably) that Java 8 has real multiple inheritance.

Static Methods in Interfaces

- **Idea**
  - Java 7 and earlier prohibited static methods in interfaces. Java 8 now allows this
- **Motivation**
  - Seems natural to put operations related to the general type in the interface
  - Does not violate the "spirit" of interfaces
    - `Shape.sumAreas(arrayOfShapes);`
- **Notes**
  - You must use interface name in the method call, even from code within a class that implements the interface
    - Shape.sumAreas, not sumAreas
  - The static methods cannot manipulate static variables
    - Java 8 interfaces continue to prohibit mutable fields





Example: Shape

Example from OOP Section

- **Goal**
  - Want to be able to make mixed collections of Circle, Square, etc.
- **Standard solution**
  - Define Shape interface and have Circle, Square, etc. implement it
- **Goal**
  - Want to be able to sum up the areas of an array of mixed Shapes
- **Standard solution**
  - Put abstract getArea method in the interface, define it in the classes
  - Make static method that takes a Shape[] and sums the areas
- **Java 8 twist**
  - Put static method directly in Shape instead of in a utility class as would have been done in Java 7

Shape

```
public interface Shape {
  double getArea(); // All real shapes must define a getArea

  public static double sumAreas(Shape[] shapes) {
    double sum = 0;
    for(Shape s: shapes) {
      sum = sum + s.getArea();
    }
    return(sum);
  }
}
```

Circle

```
public class Circle implements Shape {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  @Override
  public double getArea() {
    return(Math.PI * radius * radius);
  }

  ...
}
```

Rectangle and Square are similar.

ShapeTest

```
public class ShapeTest {
  public static void main(String[] args) {
    Shape[] shapes = { new Circle(10),       // Area is about 314.159
                       new Rectangle(5, 10), // Area is 50
                       new Square(10) };     // Area is 100
    System.out.println("Sum of areas: " +
                       Shape.sumAreas(shapes));
                                          // Area is about 464.159
  }
}
```





Example: Op

Example from First Lambda Section

- **Goal**
  - Want to be able to time various operations without repeating code
- **Java 8 solution**
  - Define functional (1-abstract-method) Op interface
  - Define static method that takes an Op, calls its method, and times it
  - Pass lambdas to the static method

```
TimingUtils.timeOp(() -> someLongOperation(...));
```

- **New twist**
  - Put static method directly in Op instead of in a utility class (TimingUtils)

```
Op.timeOp(() -> someLongOperation(...));
```

Old Approach: The Op Interface

```
@FunctionalInterface
public interface Op {
  void runOp();
}
```


Old Approach: The TimingUtils Class

```
public class TimingUtils {
  private static final double ONE_BILLION =
    1_000_000_000;

  public static void timeOp(Op operation) {
    long startTime = System.nanoTime();
    operation.runOp();
    long endTime = System.nanoTime();
    double elapsedSeconds = (endTime - startTime)/ONE_BILLION;
    System.out.printf("  Elapsed time: %.3f seconds.%n",
                      elapsedSeconds);
  }
}
```

Old Approach: Test Code

```
public class TimingTests {
  public static void main(String[] args) {
    for(int i=3; i<8; i++) {
      int size = (int)Math.pow(10, i);
      System.out.printf("Sorting array of length %,d.%n", size);
      TimingUtils.timeOp(() -> sortArray(size));
    }
  }

  // Supporting methods like sortArray
}
```


Second Approach: The Op Interface

```
@FunctionalInterface
public interface Op {
  static final double ONE_BILLION = 1_000_000_000;

  void runOp();

  static void timeOp(Op operation) {
    long startTime = System.nanoTime();
    operation.runOp();
    long endTime = System.nanoTime();
    double elapsedSeconds = (endTime - startTime)/ONE_BILLION;
    System.out.printf("  Elapsed time: %.3f seconds.%n",
                      elapsedSeconds);
  }
}
```

Second Approach: The TimingUtils Class

- **None!**

Second Approach: Test Code

```
public class TimingTests {
  public static void main(String[] args) {
    for(int i=3; i<8; i++) {
      int size = (int)Math.pow(10, i);
      System.out.printf("Sorting array of length %,d.%n", size);
      Op.timeOp(() -> sortArray(size));
    }
  }

  // Supporting methods like sortArray
}
```

Default Methods in Interfaces

Default (Concrete) Methods in Interfaces

- **Idea**
  - Java 7 and earlier prohibited concrete methods in interfaces. Java 8 now allows this.
- **Motivation**
  - Java needed to add methods like stream and forEach to List.
  - No problem for builtin classes: Java could update the definition of the List interface and all builtin classes that implemented List (ArrayList, etc.)
  - Big problem for custom (user-defined) classes that implemented List: they would fail in Java 8. Would very seriously violate the rule that new Java versions do not break existing code.
- **Note**
  - Some people argue that this breaks the spirit of interfaces, and interfaces are now more like abstract classes. Perhaps (but arguable), but it was a useful trick, and default methods in interfaces are useful in *your* code as well.

Updating the Op Interface

- **Make method to combine two Ops**
  - To produce single Op that runs the code of two other Ops
- **Natural place to put it is in Op itself**

```
Op op1 = () -> someCode(...);
Op op2 = () -> someOtherCode(...);
Op op3 = op1.combinedOp(op2);
Op.timeOp(op3);
```

- **Requires a default method**

```
public interface Op {
  ...
  default Op combinedOp(...) { ... }
}
```


Third Approach: The Op Interface

```
@FunctionalInterface
public interface Op {
  void runOp();

  static void timeOp(Op operation) {
    // Unchanged from last example
  }

  default Op combinedOp(Op secondOp) {
    return(() -> { runOp();
                   secondOp.runOp(); });
  }
}
```

Third Approach: Test Code

```
public static void main(String[] args) {
   for(int i=3; i<8; i++) {
     int size = (int)Math.pow(10, i);
     Op sortArray = () -> sortArray(size);
     Op wasteTime = () -> wasteTime(size);
     Op doBoth = sortArray.combinedOp(wasteTime);
     System.out.printf("Sorting array of length %,d.%n", size);
     Op.timeOp(sortArray);
     System.out.printf("Wasting time (%,d repeats).%n", size);
     Op.timeOp(wasteTime);
     System.out.printf("Doing both (%,d repeats).%n", size);
     Op.timeOp(doBoth);
   }
 }
 // Supporting methods like sortArray and wasteTime
}
```


The Builtin Function Interface

- **Used in section on lambdas part 3**
  - The mapSum method used apply (the main *abstract* method)
  - Code that called mapSum used lambdas or method references
    - `int numEmployees = mapSum(employees, Employee::getSalary);`
- **Used in sections on streams**
  - The builtin map method used apply (the main *abstract* method)
  - Code that called map used lambdas or method references
    - `List<Employee> emps = ids.map(Utils:findEmployee).collect(...);`
- **Used in section on lambdas part 4**
  - Will use the *static* identity method
    - `int sumOfNumbers =  mapSum(listOfIntegers, Function.identity());`
  - Will use the *default* compose & andThen methods
    - `...function1.compose(function2)...`

Source Code for Builtin Function Interface

```
@FunctionalInterface
public interface Function<T, R> {

  R apply(T t);

  default <V> Function<V,R> compose(Function<...> before) {
    ...
  }

  default <V> Function<T, V> andThen(...) { ... }

  static <T> Function<T, T> identity() {
    return t -> t;
  }
}
```


Resolving Conflicts with Default Methods

No Conflicts: Java 7

- **Interfaces Int1 and Int2 specify someMethod**

```
public interface Int1 { int someMethod(); }
public interface Int2 { int someMethod(); }
```

- **Class ParentClass defines someMethod**

```
public class ParentClass {
  public int someMethod() { return(3); }
}
```

- **Examples**

```
public class SomeClass implements Int1, Int2 { ... }
```

  - No conflict: SomeClass must define someMethod, and by doing so, satisfies both interfaces

```
public class ChildClass extends ParentClass implements Int1 { ... }
```

  - No conflict: the child class inherits someMethod from ParentClass, and interface is satisfied

Potential Conflicts: Java 8

- **Interfaces Int1 and Int2 define someMethod**

```
public interface Int1 { default int someMethod() { return(5); } }
public interface Int2 { default int someMethod() { return(7); } }
```

- **Class ParentClass defines someMethod**

```
public class ParentClass {
  public int someMethod() { return(3); }
}
```

- **Examples**

```
public class SomeClass implements Int1, Int2 { ... }
```

  - Potential conflict: whose definition of someMethod wins, the one from Int1 or the one from Int2?

```
public class ChildClass extends ParentClass implements Int1 { ... }
```

  - Potential conflict: whose definition of someMethod wins, the one from ParentClass or the one from Int1?

Resolving Conflicts

- **Classes win over interfaces**

  `public class ChildClass extends ParentClass implements Int1`

  – Conflict resolved: the version of someMethod from ParentClass wins over the version from Int1

  – This rule also means that interfaces cannot provide default implementations for methods from Object (e.g., toString)

    - The methods from the interface could *never* be used, so Java prohibits you from even writing them

- **Conflicting interfaces: you must redefine**

  `public class SomeClass implements Int1, Int2`

  – The conflict cannot be resolved automatically, and SomeClass must give a new definition of someMethod

  – But, this new method can refer to one of the existing methods with Interface1.super.someMethod(…) or Interface2.super.someMethod

Wrap-Up

Summary

- **Static methods**
  - Use for methods that apply *to* instances of that interface
    - Shape.sumAreas(Shape[] shapes)
    - Op.timeOp(Op opToTime)
- **Default methods**
  - Use to add behavior to existing interfaces without breaking classes that already implement the interface
  - Use for operations that are called *on* instances of your interface type
  - Resolving conflicts
    - Classes win over interfaces
    - If two interfaces conflict, class must reimplement the method
      - But the new method can refer to old method by using InterfaceName.super.methodName