Using Lambdas to Write Mixins in Java 8

Dr Heinz M. Kabutz
heinz@javaspecialists.eu

Last updated 2014-11-12
© 2014 Heinz Kabutz – All Rights Reserved
Copyright Notice

- © 2014 Heinz Kabutz, All Rights Reserved
- No part of this talk material may be reproduced without the express written permission of the author, including but not limited to: blogs, books, courses, public presentations.
- A license is hereby granted to use the ideas and source code in this course material for your personal and professional software development.
- Please contact heinz@javaspecialists.eu if you are in any way uncertain as to your rights and obligations.
Who is Heinz Kabutz?

- Java consultant, teacher, programmer
- Born in Cape Town, South Africa, now lives on Crete
- Created The Java Specialists’ Newsletter
  - www.javaspecialists.eu
- One of the first Java Champions
  - www.javachampions.com
- Unfounder of hottest Unconference JCrete (jcrete.org)
Functional Interface
Java 8 Lambda Syntax

- In Java 7, we did this

```java
public void greetConcurrent() {
    new Thread(new Runnable() {
        public void run() { sayHello(); }
    }).start();
}

private void sayHello() { System.out.println("Kalamari!"); }
```

- With Java 8 Lambdas, we can do this

```java
public void greetConcurrent() {
    new Thread(() -> sayHello()).start();
}
```
Functional Interface

- Lambdas have to be functional interfaces
- Definition: *Functional Interface*
  - Interface
  - Exactly one abstract method
    - Methods inherited from Object do not count
Is this a Functional Interface?

```java
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```
Yes it is!

@FunctionalInterface
public interface Runnable {
    public abstract void run();
}

threadPool.submit(() -> sayHello());
@FunctionalInterface
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
Using Lambdas to Write Mixins in Java 8

We first need to look at EventListener

```java
@FunctionalInterface
public interface EventListener {
}
```

EventListener is **not** a Functional Interface
Yes it is!

```java
@FunctionalInterface
public interface EventListener {
}

@FunctionalInterface
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

ActionListener Interface has exactly one abstract method.
@FunctionalInterface
public interface Stringer {
   // force class to implement toString()
   String toString();
}
No, it is not!

```java
@FunctionalInterface
public interface Stringer {
    // force class to implement toString()
    String toString();
}
```

Public methods defined inside Object do not count
public class Object {
    public final Class<?> getClass();
    public int hashCode();
    public boolean equals(Object obj);
    protected Object clone();
    public String toString();
    public final void notify();
    public final void notifyAll();
    public final void wait(long timeout);
    public final void wait(long timeout, int nanos);
    public final void wait();
    protected void finalize();
}
Using Lambdas to Write Mixins in Java 8

Final methods cannot be added to interface

```java
public final Class<?> getClass();
public final void notify();
public final void notifyAll();
public final void wait(long timeout);
public final void wait(long timeout, int nanos);
public final void wait();
```
Public non-final methods for functional interfaces

```java
public int hashCode();
public boolean equals(Object obj);
public String toString();
```
Protected methods count for functional interfaces

```java
protected void finalize();
protected Object clone();
```
Are these Functional Interfaces?

@FunctionalInterface
public interface Foo1 {
    boolean equals(Object obj);
}

@FunctionalInterface
public interface Bar1 extends Foo1 {
    int compare(String o1, String o2);
}
Foo1 is not, but Bar1 is

```java
@FunctionalInterface
public interface Foo1 {
    boolean equals(Object obj);
}

@FunctionalInterface
public interface Bar1 extends Foo1 {
    int compare(String o1, String o2);
}
```
@FunctionalInterface
public interface Comparator<T> {
    public abstract boolean equals(Object obj);
    int compare(T o1, T o2);
}
Using Lambdas to Write Mixins in Java 8

Yes, it is!

```java
@FunctionalInterface
public interface Comparator<T> {
    public abstract boolean equals(Object obj);
    int compare(T o1, T o2);
}
```

Equals(Objects) is already an implicit member

Interface with exactly one abstract method
@FunctionalInterface
public interface CloneableFoo {
    int m();
    Object clone();
}
@FunctionalInterface
public interface CloneableFoo {
    int m();
    Object clone();
}
Using Lambdas to Write Mixins in Java 8

Is this a Functional Interface?

```java
@FunctionalInterface
public interface MouseListener
    extendsEventListener {
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
}
```
No, it is not!

@FunctionalInterface
public interface MouseListener extends EventListener {
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
}

MouseListener has five abstract methods
Using Lambdas to Write Mixins in Java 8

Fundamental Functional Interfaces
Java 8 contains some standard functional interfaces

- Supplier<T>
- Consumer<T>
- Predicate<T>
- Function<T, R>
- UnaryOperator<T>
- BinaryOperator<T>
Supplier<T>

- Use whenever you want to supply an instance of a T
- Can delay object creation, for example:

```java
public void foo() {
    logger.fine("ms since 1970: " + System.currentTimeMillis());
}

public void bar() {
    logger.fine(() -> "ms since 1970: " + System.currentTimeMillis());
}

public void fine(Supplier<String> msgSupplier) {
    log(Level.FINE, msgSupplier);
}
```
Consumer<T>

- Provide an action to be performed on an object

```java
Collection<String> names = Arrays.asList("Kirk", "Andrea", "Szonya", "Anna");
names.forEach(new Consumer<String>() {
    public void accept(String s) {
        System.out.println(s.toUpperCase());
    }
});

names.forEach(s -> System.out.println(s.toUpperCase()));

names.stream().map(String::toUpperCase).forEach(System.out::println);
```
Using Lambdas to Write Mixins in Java 8

Predicate<T>

- A boolean-valued property of an object

```java
Collection<String> names = Stream.of(
    "Heinz","Helene","Maxi","Connie","Bangie","Efi").
    collect(Collectors.toList());
names.removeIf(new Predicate<String>() {
    public boolean test(String s) {
        return s.contains("i");
    }
});
```

```java
names.removeIf(s -> s.contains("i"));
```
Using Lambdas to Write Mixins in Java 8

Function<T, R>

- Transforming a T to an R

```java
Collection<String> names = Arrays.asList("Heinz", "Helene", "Maxi", "Connie", "Bangie", "Efi");
names.stream().map(
    new Function<String, Integer>() {
        public Integer apply(String s) {
            return s == null ? 0 : s.length();
        }
    });
```

```java
names.stream().map(s -> s == null ? 0 : s.length());
```
Using Lambdas to Write Mixins in Java 8

UnaryOperator<T>

- Transforming a T - similar to Function<T, T>

```java
List<String> names = Arrays.asList("Heinz", "Helene", "Maxi", "Connie", "Bangie", "Efi");
names.replaceAll(new UnaryOperator<String>() {
    public String apply(String s) {
        return s.toUpperCase();
    }
});

names.replaceAll(s -> s.toUpperCase());

names.replaceAll(String::toUpperCase);
```
Mixins Using Java 8

Lambdas
Using Lambdas to Write Mixins in Java 8

Mixins using Java 8 Lambdas

- State of the Lambda has this misleading example
  ```java
  public interface ActionListener {
    void actionPerformed(ActionEvent e);
  }
  button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      ui.dazzle(e.getModifiers());
    }
  });
  ```

- With Java 8 Lambdas, this becomes
  ```java
  button.add ActionListener(e -> ui.dazzle(e.getModifiers()));
  ```

- But most AWT Listeners *not* functional interfaces
Using Lambdas to Write Mixins in Java 8

Pre-Lambda Event Listeners

```java
salaryIncreaser.addFocusListener(new FocusAdapter() {
    public void focusGained(FocusEvent e) {
        System.out.println("Almost there!");  
    }
});
salaryIncreaser.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        e.consume();
        System.out.println("Not quite!");
    }
});
salaryIncreaser.addMouseListener(new MouseAdapter() {
    public void mouseEntered(MouseEvent e) {
        shuffleSalaryButton();
    }
});
```
salaryIncreaser.addFocusGainedListener(
    e -> System.out.println("Almost there!"));

salaryIncreaser.addKeyPressedListener(
    e -> {
      e.consume();
      System.out.println("Not quite!");
    });

salaryIncreaser.addMouseEnteredListener(
    e -> shuffleSalaryButton());

This is What We Want

How do we get there?
Focus/Mouse/KeyListeners are not Functional Interfaces

- They have several abstract methods

```java
public interface FocusListener {
    /**
     * Invoked when a component gains the keyboard focus.
     */
    void focusGained(FocusEvent e);

    /**
     * Invoked when a component loses the keyboard focus.
     */
    void focusLost(FocusEvent e);
}
```
In previous example, we MouseAdapter, FocusAdapter and KeyAdapter

```java
public abstract class FocusAdapter
    implements FocusListener {
    public void focusGained(FocusEvent e) {}
    public void focusLost(FocusEvent e) {}
}
```
public interface FocusEventProducerMixin {
    void addFocusListener(FocusListener l);

    default void addFocusGainedListener(Consumer<FocusEvent> c) {
        addFocusListener(new FocusAdapter() {
            public void focusGained(FocusEvent e) { c.accept(e); }
        });
    }

    default void addFocusLostListener(Consumer<FocusEvent> c) {
        addFocusListener(new FocusAdapter() {
            public void focusLost(FocusEvent e) { c.accept(e); }
        });
    }
}
What Just Happened?

- We defined an interface with default methods
  - Both `addFocusGainedListener()` and `addFocusLostListener()` call the abstract method `addFocusListener()` in the interface
  - It is a Functional Interface, but that does not matter in this case
- Let’s see how we can “mixin” this interface into an existing class `JButton`
Using Lambdas to Write Mixins in Java 8

**JButtonLambda Mixin Magic**

- JButton contains method addFocusListener
- We subclass it and implement Mixin interface
  - We could even leave out the constructors and just have
    ```java
    public class JButtonLambda extends JButton implements FocusEventProducerMixin {
    }
    ```
- With our new JButtonLambda, we can now call
  ```java
  salaryIncreaser.addFocusGainedListener(
      e -> System.out.println("Almost there!"))
  ```
public class JButtonLambda extends JButton implements FocusEventProducerMixin {
    public JButtonLambda() {}
    public JButtonLambda(Icon icon) { super(icon); }
    public JButtonLambda(String text) { super(text); }
    public JButtonLambda(Action a) { super(a); }
    public JButtonLambda(String text, Icon icon) {
        super(text, icon);
    }
}
Combining Different Mixins

- Each class in the hierarchy adds new addXXXListener() methods
  - Here are just some of them
- We can define a JComponent mixin that contains all the addXXXListener and other mixins in the classes above
Using Lambdas to Write Mixins in Java 8

JComponent Mixin

```java
public interface JComponentEventProducerMixin extends
    AncestorEventProducerMixin,
    ComponentEventProducerMixin,
    ContainerEventProducerMixin,
    FocusEventProducerMixin,
    HierarchyEventProducerMixin,
    InputMethodEventProducerMixin,
    KeyEventProducerMixin,
    MouseEventProducerMixin,
    MouseMotionEventProducerMixin {
    void addHierarchyListener(HierarchyListener l);
    void addMouseWheelListener(MouseWheelListener l);
    void addPropertyChangeListener(PropertyChangeListener l);
    void addVetoableChangeListener(VetoableChangeListener l);
}
```
AbstractButton Mixin

```java
public interface AbstractButtonEventProducerMixin {
    void addActionListener(ActionListener l);
    void addItemListener(ItemListener l);
    void addChangeListener(ChangeListener l);
}
```

We need this so that we have a common super-interface that we can cast all types of abstract buttons to.
public class JButtonLambda extends JButton implements JComponentEventProducerMixin, AbstractButtonEventProducerMixin {
    public JButtonLambda() {
    }
    // and other constructors
}
Using Lambdas to Write Mixins in Java 8

ComponentPallet Demo
Facade Pattern For Listeners
Facade Pattern for Listeners

Another approach is facades for each listener

```java
public interface FocusListeners {
    static FocusListener forFocusGainedListener(
        Consumer<FocusEvent> c) {
        return new FocusAdapter() {
            public void focusGained(FocusEvent e) {c.accept(e);} ...
        };
    }

    static FocusListener forFocusLostListener(
        Consumer<FocusEvent> c) {
        return new FocusAdapter() {
            public void focusLost(FocusEvent e) { c.accept(e); } ...
        };
    }
}
```
Facade Pattern for Listeners

salaryIncreaser.addFocusListener(
    FocusListeners.forFocusGainedListener(
        e -> System.out.println("Almost there!")));

salaryIncreaser.addKeyListener(
    KeyListeners.forKeyPressedListener(
        e -> {
            e.consume();
            System.out.println("Not quite!");
        }));

salaryIncreaser.addMouseListener(
    MouseListeners.forMouseEntered(
        e -> shuffleSalaryButton()));
Method Call Stacks

Anonymous inner classes use synthetic static methods to access private members

```java
private void showStack() {
    Thread.dumpStack();
}

private void anonymousClassCallStack() {
    Runnable runnable = new Runnable() {
        public void run() {
            showStack();
        }
    };
    runnable.run();
}
```
Method Call Stacks

- **Output of run**
  
  ```java
  java.lang.Exception: Stack trace
  at java.lang.Thread.dumpStack(Thread.java:1329)
  at MethodCallStack.showStack(MethodCallStack.java:3)
  at MethodCallStack.access$000(MethodCallStack.java:1)
  at MethodCallStack$1.run(MethodCallStack.java:9)
  at MethodCallStack.anonymousClassCallStack(MethodCallStack.java:12)
  ```

- **Synthetic method in MethodCallStack.class**

  ```java
  static void MethodCallStack.access$000(MethodCallStack)
  ```
Method Call Stacks

- **Lambdas have more direct access to outer class**

```java
public void lambdaCallStack() {
    Runnable runnable = () -> showStack();
    runnable.run();
}
```

```
java.lang.Exception: Stack trace
at java.lang.Thread.dumpStack(Thread.java:1329)
at MethodCallStack.showStack(MethodCallStack.java:3)
at MethodCallStack.lambda$lambdaCallStack$0(MethodCallStack.java:16)
at MethodCallStack.$$Lambda$$1/455659002.run(Unknown Source)
at MethodCallStack.lambdaCallStack(MethodCallStack.java:17)
```

- **Synthetic \lambda method in MethodCallStack.class**

```java
private void MethodCallStack.lambda$lambdaCallStack$0()
```
Conclusion
Mixins in GitHub

- Code with more details available here
  - https://github.com/kabutz/javaspecialists-awt-event-mixins
  - (http://tinyurl.com/jmixins)
Lambdas, Static and Default Methods

- Java 8 released in March 2014
- Practical use of language will produce idioms
- Mixin idea can be applied in other contexts too
  - e.g. Adding functionality to Enums
- Java 8 will probably take another year to be used
  - Some of my customers are still using Java 1.4
    - One even has modules with Java 1.1