developerWorks®

Java 8 idioms: Functional interfaces

Learn how to create custom functional interfaces, and why you should use built-ins whenever possible

Venkat Subramaniam September 08, 2017

In Java 8, lambda expressions are treated as a type of functional interface. Learn how this design decision supports backward compatibility with older versions of the language, then see examples of both custom and built-in functional interfaces in a Java program. Find out why using built-in interfaces is usually optimal, even in cases where a custom interface might seem more obvious.

What's the *type* of a lambda expression? Some languages use *function values* or *function objects* to represent lambda expressions, but not the Java™ language. Instead, Java uses functional interfaces to represent lambda expression types. It might seem strange at first, but in fact it is an efficient way to ensure backward compatibility with older versions of Java.

The following piece of code should be familiar:

```
Thread thread = new Thread(new Runnable() {
  public void run() {
    System.out.println("In another thread");
  }
});

thread.start();

System.out.println("In main");
```

The `Thread` class and its constructor were introduced in Java 1.0, well over 20 years ago. Since then, the constructor hasn't changed. It's traditional to pass an anonymous instance of `Runnable` to the constructor. But starting in Java 8 you have the option to pass a lambda expression instead:

```
Thread thread = new Thread(() -> System.out.println("In another thread"));
```

About this series
Java 8 is the most significant update to the Java language since its inception—packed so full of new features that you might wonder where to start. In this series, author and educator Venkat Subramaniam offers an idiomatic approach to Java 8: short explorations that

invite you to rethink the Java conventions you've come to take for granted, while gradually integrating new techniques and syntax into your programs.

The constructor of the `Thread` class is expecting an instance that implements `Runnable`. In this case, rather than passing an object we pass a lambda expression. We have the option to pass the lambda expression with a variety of methods and constructors, including some created prior to Java 8. It works because lambda expressions are represented as functional interfaces in Java.

There are three important rules for functional interfaces:

1. A functional interface has just one abstract method.
2. Any abstract method that's also a public method in the `Object` class is not counted as that method.
3. A functional interface may have default methods and static methods.

Any interface that satisfies the rule of the single abstract method will be automatically considered a functional interface. This includes traditional interfaces such as `Runnable` and `Callable`, and custom interfaces that you build yourself.

Built-in functional interfaces

In addition to the single abstract method interfaces already mentioned, JDK 8 includes several new functional interfaces. The most common ones are `Function<T, R>`, `Predicate<T>`, and `Consumer<T>`, which are defined in the `java.util.function` package. The `map` method of `Stream` takes `Function<T, R>` as a parameter. Likewise, `filter` uses `Predicate<T>` and `forEach` uses `Consumer<T>`. The package also has other functional interfaces like `Supplier<T>`, `BiConsumer<T, U>`, and `BiFunction<T, U, R>`.

It's possible to use a built-in functional interface as a parameter to our own methods. For example, suppose we have a `Device` class with methods like `checkout` and `checkin` to indicate whether a device is in use. When a user requests a new device, the method `getFromAvailable` returns one from a pool of available devices or creates a new one if necessary.

We could implement a function to borrow a device, like so:

```
public void borrowDevice(Consumer<Device> use) {
  Device device = getFromAvailable();

  device.checkout();

  try {
    use.accept(device);
  } finally {
    device.checkin();
  }
}
```

The `borrowDevice` method:

- Takes a `Consumer<Device>` as parameter.
- Gets a device from the pool (we're not concerned with thread safety for this example).

- Calls the `checkout` method to set the device status *checked out*.
- Delivers the device to the consumer.

When a device is returned from the call to the `Consumer`'s `accept` method, its status is changed to *checked in* by calling the `checkin` method.

Here's one way to use the `borrowDevice` method:

```
new Sample().borrowDevice(device -> System.out.println("using " + device));
```

Because the method receives a functional interface as its parameter, it's acceptable to pass in a lambda expression as argument.

Custom functional interfaces

While it's best to use a built-in functional interface whenever possible, you sometimes need a custom functional interface.

To create your own functional interface, do two things:

1. Annotate the interface with `@FunctionalInterface`, which is the Java 8 convention for custom functional interfaces.
2. Ensure that the interface has just one abstract method.

The convention makes it clear that the interface is intended to receive lambda expressions. When the compiler sees the annotation, it will verify that the interface has just one abstract method.

Using the `@FunctionalInterface` annotation ensures that if you accidentally violate the abstract-method-count rule during a future change to the interface, you will get an error message. This is helpful because you will catch the issue immediately, rather than leaving it for another developer to deal with later. No one wants to get an error message when passing a lambda expression to someone else's custom interface.

Creating a custom functional interface

As an example, let's create an `Order` class that has a list of `OrderItem`s and a method to transform and print them. We'll start with an interface.

The code below creates a `Transformer` functional interface.

```
@FunctionalInterface
public interface Transformer<T> {
  T transform(T input);
}
```

The interface is tagged with the `@FunctionalInterface` annotation, conveying that it is a functional interface. Since that annotation is part of the `java.lang` package, no imports are necessary. The interface has a method named `transform` that takes an object of parametrized type `T` and returns a transformed object of the same type. The semantics of the transformation will be decided by the implementation of the interface.

Here's the `OrderItem` class:

```
public class OrderItem {
  private final int id;
  private final int price;

  public OrderItem(int theId, int thePrice) {
    id = theId;
    price = thePrice;
  }

  public int getId() { return id; }
  public int getPrice() { return price; }

  public String toString() { return String.format("id: %d price: %d", id, price); }
}
```

`OrderItem` is a simple class that has two properties: `id` and `price`, and a `toString` method.

Now, take a look at the `Order` class.

```
import java.util.*;
import java.util.stream.Stream;

public class Order {
  List<OrderItem> items;

  public Order(List<OrderItem> orderItems) {
    items = orderItems;
  }

  public void transformAndPrint(
    Transformer<Stream<OrderItem>> transformOrderItems) {

    transformOrderItems.transform(items.stream())
      .forEach(System.out::println);
  }
}
```

The `transformAndPrint` method takes `Transform<Stream<OrderItem>` as parameter, invokes the `transform` method to transform the order items belonging to the `Order` instance, and prints the order items in the transformed sequence.

Here's a sample that uses this method:

```
import java.util.*;
import static java.util.Comparator.comparing;
import java.util.stream.Stream;
import java.util.function.*;

class Sample {
  public static void main(String[] args) {
    Order order = new Order(Arrays.asList(
      new OrderItem(1, 1225),
      new OrderItem(2, 983),
      new OrderItem(3, 1554)
    ));


    order.transformAndPrint(new Transformer<Stream<OrderItem>>() {
      public Stream<OrderItem> transform(Stream<OrderItem> orderItems) {
        return orderItems.sorted(comparing(OrderItem::getPrice));
      }
    });
  }
}
```

We pass an anonymous inner class as argument to the `transformAndPrint` method. Within the `transform` method, we invoke the `sorted` method of the given stream, which will sort the order items. Here's the output of our code, showing the order items sorted in ascending order of price:

```
id: 2 price: 983
id: 1 price: 1225
id: 3 price: 1554
```

The power of lambda expressions

Anywhere a functional interface is expected, we have three choices:

1. Pass an anonymous inner class.
2. Pass a lambda expression.
3. Pass a method reference instead of a lambda expression, in some cases.

Passing an anonymous inner class is verbose, and we can only pass the method reference as an alternative to a pass-through lambda expression. Consider what happens if we rewrite our call to the `transformAndPrint` function to use a lambda expression instead of an anonymous inner class:

```
order.transformAndPrint(orderItems -> orderItems.sorted(comparing(OrderItem::getPrice)));
```

That's much more concise and easier to read than the anonymous inner class we started with.

Custom vs built-in functional interfaces

Our custom functional interface illustrates the advantages and disadvantages of creating custom interfaces. Consider the advantages first:

- You can give your custom interface a descriptive name that helps other developers modifying or reusing it. Names like `Transformer`, `Validator`, and `ApplicationEvaluator` are domain specific, and can help someone reading the interface's methods to infer what's expected as argument.
- You can give the abstract method any syntactically valid name you please. This benefit is only for the receiver of the interface, and only in cases where an abstract method is being passed. A caller passing lambda expressions or method references won't receive this benefit.
- You can use parametrized types in your interface, or keep it simple and specific to a few types. In this case, you could write the `Transformer` interface to use `OrderItems` instead of parameterized type `T`.
- You can write custom default methods and static methods, which can be used by other implementations of the interface.

Of course, there are also disadvantages to using custom functional interfaces:

- Imagine creating several interfaces, all having abstract methods with the same signature, such as taking `String` as parameter and returning `Integer`. While the names of the methods may be different, they are mostly redundant and could be replaced by one interface with a generic name.
- Anyone wanting to use a custom interfaces must make an extra effort to learn, understand, and remember it. All Java programmers are familiar with `Runnable` in the `java.lang` package. We've seen it over and over, so there's no effort to remember its purpose. However, if I use a custom `Executor`, you will have to carefully learn about the interface before using it. That effort is worthwhile in some cases, but it's wasted if `Executor` is too similar to `Runnable`.

Which one is best?

Knowing the pros and cons of custom versus built-in functional interfaces, how would you decide which to use? Let's revisit the `Transformer` interface to find out.

Recall that `Transformer` exists to convey the semantics of transforming one object into another. Here, we're referring it by name:

```
public void transformAndPrint(Transformer<Stream<OrderItem>> transformOrderItems) {
```

The method `transformAndPrint` receives an argument that is in charge of transformation. The transformation may re-sequence elements in the `OrderItems` collection. Alternatively, it could mask out some details of each order item. Or the transformation could decide to do nothing and merely return the original collection. Implementation is left to the caller.

What is essential is that the caller knows they can provide a transformation implementation as argument to the `transformAndPrint` method. The name of the functional interface and its documentation should provide those details. In this case, it's also clear from the name of the parameter (`transformOrderItems`) and should be included with documentation for the `transformAndPrint` function. While the name of the functional interface is useful, it isn't the only window into its purpose and usage.

Looking closely at the `Transformer` interface and comparing its purpose with the JDK's built-in functional interfaces, we see that `Function<T, R>` could replace `Transformer`. To test it out, we remove the `Transformer` functional interface from our code and change the `transformAndPrint` function, like so:

```
public void transformAndPrint(Function<Stream<OrderItem>, Stream<OrderItem>> transformOrderItems) {
  transformOrderItems.apply(items.stream())
    .forEach(System.out::println);
}
```

The change is minor—in addition to changing `Transformer<Stream<OrderItem>>` to `Function<Stream<OrderItem>>`, `Stream<OrderItem>>`, we changed the method call from `transform()` to `apply()`.

Had the call to `transformAndPrint` used an anonymous inner class, we would have to change that, too. However, we've already changed the call to use a lambda expression:

```
order.transformAndPrint(orderItems -> orderItems.sorted(comparing(OrderItem::getPrice)));
```

The name of the functional interface is irrelevant to the lambda expression—it would only be relevant for the compiler, which ties the lambda expression argument to the method parameter. The name of the method `transform` versus `apply` is equally irrelevant to the caller.

Using a built-in functional interface has left us with one less interface, and the call to the method works just the same. We also haven't compromised code readability. This exercise tells us that we could easily replace our custom functional interface with a built-in one. We would just need to provide documentation for the `transformAndPrint` (not shown) and name the argument more descriptively.

Conclusion

The design decision to make lambda expressions a type of functional interface facilitates backward compatibility between Java 8 and earlier versions of Java. It's possible to pass a lambda expression to any older function that would typically receive a single abstract method interface. To receive lambda expressions, a method's parameter type should be a functional interface.

In some cases, it makes sense to create your own functional interface, but you should do so cautiously. Consider a custom functional interface only if your application requires highly specialized methods, or if no existing interface will meet your needs. Always check whether the functionality exists in one of the JDK's built-in functional interfaces. Use built-in functional interfaces wherever you can.

Related topics

- Functional interfaces in Java 8
- Java programming with lambda expressions
- Java 8 language changes
- Functional Programming in Java: The Pragmatic Bookshelf, 2014