Quick Lambda Tricks

A five minute survival guide on Java 8 lambdas

@PeterHendriks80

mindloops

JFALL

Inner classes done right

```
public void helloWorldNoLambda() {
    SortedSet<Person> sortedPersons = new TreeSet<>(new Comparator<Person>() {
        @Override
        public int compare(Person person1, Person person2) {
            return person1.getName().compareTo(person2.getName());
        }
    });
}
```

```
public void helloWorldLambda() {
    SortedSet<Person> sortedPersons =
        new TreeSet<>((person1, person2) -> person1.getName().compareTo(person2.getName()));
}
```

Lambdas need a Lambda API

```
Optional<Person> dude = findDude("Keanu");
dude.ifPresent(realDude -> System.out.println(realDude.getName()));
```

```
double averageAge = persons.stream()
        .mapToInt(Person::getAge)
        .average().getAsDouble();
```

```
CompletableFuture.runAsync(() -> System.out.println("Hello JFall"))
        .thenAccept(x -> System.out.println("Done!"));
```

Keeping lambdas clean

The inline rocket

```
persons.stream()
        .filter(person -> person.getAge() < 20)
        .forEach(person -> System.out.print("Java is older than: " + person.getName()));
```

Recursive rockets

```
private Function<Turtle, Turtle> turtleMapper;

public void mapTurtle() {
    turtleMapper = turtle -> {
        Optional<Turtle> nextTurtle = allTheWayDown(turtle);
        return nextTurtle.map(recursiveTurtle -> turtleMapper.apply(recursiveTurtle)).get();
    };
}
```

Method references

```
public List<Person> methodReference(List<String> names) {
    return names.stream()
            .map(Person::new)
            .sorted(this::compareByName)
            .collect(toList());
}
```

```
public int compareByName(Person person1, Person person2) {
    return person1.getName().compareTo(person2.getName());
}
```

```
class Person {

    private String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Method references problems

```
public void problematicMethodReferences(List<Person> persons) {
    PersonProcessor instanceMustExist = new PersonProcessor();
    persons.stream()
            .filter(ReallyLongDescriptiveSupportUtil::isOlderThanJava)
            .sorted((person1, person2) -> compareByNameCaseSensitive(person1, person2, false))
            .forEach(instanceMustExist::process);
}
```

```
public int compareByNameCaseSensitive(Person person1, Person person2, boolean caseSensitive) {
    return …
}
```

Factory methods

```
public void findYoungChuckNorris() {
    persons.stream()
            .filter(isChuckNorris())
            .filter(isOlderThanJava())
            .findAny().ifPresent(person ->
            System.out.println(person.getName() + " not missing in action!"));
}
```

```
public Predicate<Person> isChuckNorris() {
    return person -> person.getName().equals("Chuck Norris");
}
```

```
public static Predicate<Person> isOlderThanJava() {
    return ReallyLongDescriptiveSupportUtil::isOlderThanJava;
}
```

Functional interfaces rock

```
public void findNormalPeople() {
    persons.stream()
            .filter(isChuckNorris().negate())
            .findAny().ifPresent(person ->
            System.out.println("Not Norris"));
}
```

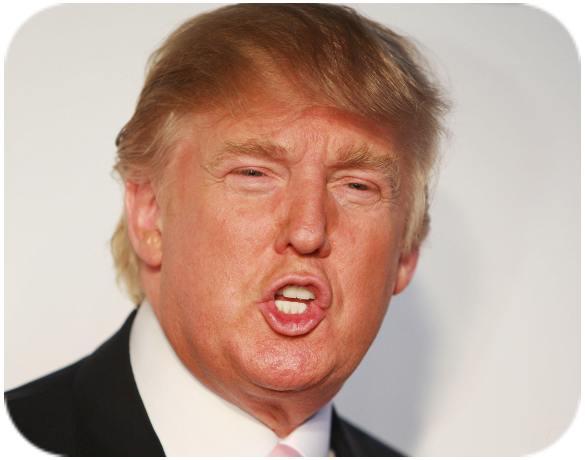
```
package java.util.function;

@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
    }

    default Predicate<T> negate() {
    }

    default Predicate<T> or(Predicate<? super T> other) {
    }
}
```

Double type inference jeopardy

```
public <T> void send(String address, Object message, Handler<AsyncResult<Message<T>>>
replyHandler);
```

```
void doesNotCompileUnknownMessageType() {
    eventBus.send("addr", new JsonObject(), result ->
result.result().body().getJsonObject("answer"));
}
```

```
Compiler error: cannot find symbol
  symbol:   method getJsonObject(java.lang.String)
  location: class java.lang.Object
```



More thinking with less typing

```
void longWindedCastSend() {
    eventBus.send("addr", new JsonObject(), (AsyncResult<Message<JsonObject>> result) ->
result.result().body().getJsonObject("answer"));
}
```

```
void explicitGenericSend() {
    eventBus.<JsonObject>send("addr", new JsonObject(), result ->
result.result().body().getJsonObject("answer"));
}
```

```
public <T> void send(String address, Object message, Handler<AsyncResult<Message<T>>>
replyHandler);
```

Dealing with legacy code

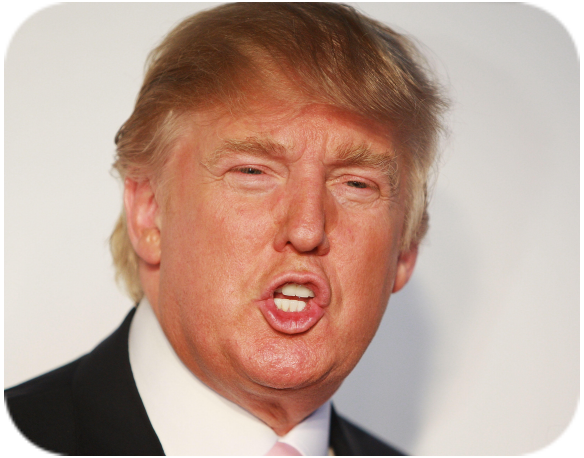
Abstract classes don’t work

```
public abstract class OldAbstractListener {

    public abstract void eventFired(Object event);
}
```

```
void subscribe(OldAbstractListener listener) {
}
```

```
void subscribeListener() {
    subscribe(event -> System.out.println("Event fired!"));
}
```

Compiler error: incompatible types:
OldAbstractListener is not a functional interface



More indirection solves everything

```
private static final class OldAbstractListenerAdapter extends OldAbstractListener {

    private final Consumer<Object> eventFiredConsumer;

    public OldAbstractListenerAdapter(Consumer<Object> eventFiredConsumer) {
        this.eventFiredConsumer = eventFiredConsumer;
    }

    @Override
    public void eventFired(Object event) {
        eventFiredConsumer.accept(event);
    }
}
```

```
public static OldAbstractListener listener(Consumer<Object> eventFiredConsumer) {
    return new OldAbstractListenerAdapter(eventFiredConsumer);
}
```

```
void subscribeListener() {
    subscribe(listener(event -> System.out.println("Event fired!")));
}
```

Checked exceptions *sigh*

```
public void checkedExceptionLeadsToMessyLambda() {
    persons.stream().forEach(person -> {
        try {
            Files.write(Paths.get(person.getName() + ".txt"), singleton(person.getName()));
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    });
}
```

A lambda for your exception

```
public static <T> Consumer<T> adaptIo(IOConsumer<T> ioConsumer) {
    return t -> {
        try {
            ioConsumer.accept(t);
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    };
}
```

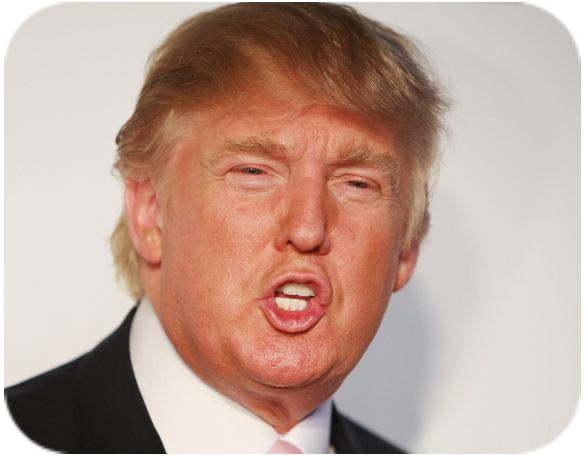
```
@FunctionalInterface
public interface IOConsumer<T> {

    void accept(T t) throws IOException;
}
```

```
public void wrappedLambda() {
    persons.stream().forEach(performIO(person ->
            Files.write(Paths.get(person.getName() + ".txt"), singleton(person.getName()))));
}
```

Serialization

```
public TreeSet<Person> getNonSerializableTreeSet() {
    return new TreeSet<>(this::compareByName);
}
```

Runtime error: TreeSet cannot be serialized because of non-serializable comparator



Serializable, your lambda is

```
public TreeSet<Person> getSerializableTreeSet() {
    return new TreeSet<>((Comparator<Person> & Serializable) this::compareByName);
}
```

```
private Comparator<Person> getSerializableCompare() {
    return (Comparator<Person> & Serializable) this::compareByName;
}
```

Good luck!

J-FALL nl jug

@PeterHendriks80

mindloops