



Writing better Functional Java code

@BrianVerm



EVERY THING



aussiemadness.com

@BrianVerm

 snyk



IF IT LOOKS STUPID BUT WORKS

IT AIN'T STUPID

TRUST ME....



**...I'M AN
ENGINEER.**

Brian Vermeer

Developer Advocate



snyk



Oracle
Groundbreaker
Ambassador



@BrianVerm



Don't do to much in a single lambda

Lambda Expression

In computer programming, a **lambda expression** is a function definition that is not bound to an identifier. Anonymous functions are often:^[1]

- arguments being passed to higher-order functions, or
- used for constructing the result of a higher-order function that needs to return a function.

Lambda in Java

Anonymous Inner functions

Satisfy a Functional Interface

They only exist in runtime

Single Line **<input> -> <output>** or Block **<input> -> { function body }**

Example

```
public void execute() {  
    Beer grolsch = new Beer("Grolsch", 4.3);  
    String result = handleBeer(grolsch,  
        beer -> grolsch.getName() + " - " + grolsch.getAlcohol());  
    System.out.println(result);  
}  
  
private String handleBeer(Beer beer, Function<Beer, String> func) {  
    System.out.println("handling beer " + beer.getName());  
    return func.apply(beer);  
}  
  
//Output : Grolsch-4.3
```

Example

```
public void execute() {  
    Beer grolsch = new Beer("Grolsch", 4.3);  
    String result = handleBeer(grolsch,  
        beer -> grolsch.getName() + " - " + grolsch.getAlcohol());  
    System.out.println(result);  
}  
  
private String handleBeer(Beer beer, Function<Beer, String> func) {  
    System.out.println("handling beer " + beer.getName());  
    return func.apply(beer);  
}  
  
//Output : Grolsch-4.3
```

Don't do Block Lambda

```
beer -> {
    String name;
    if (beer.getName().contains(" ")) {
        name = beer.getName().replace(" ", "");
    } else {
        name = beer.getName();
    }

    try {
        name += Integer.parseInt(beer.getAlcoholPercentage().toString());
    } catch (NumberFormatException nfe) {
        name += beer.getAlcoholPercentage();
    }

    return name;
}
```

Transform to a methode

```
private String createFullName (Beer beer) {  
    String name;  
    if (beer.getName () .contains (" ")) {  
        name = beer.getName () .replace (" ", "");  
    } else {  
        name = beer.getName ();  
    }  
  
    try {  
        name += Integer.parseInt(beer.getAlcoholPercentage () .toString ()) ;  
    } catch (NumberFormatException nfe) {  
        name += beer.getAlcoholPercentage () ;  
    }  
  
    return name;  
}  
  
String result = handleBeer(grolsch, this::createFullName) ;
```

Returning a Stream

A stream is NOT a data structure

A stream is NOT a data structure

A stream is NOT a data structure

What is Stream [in Java]

Flow of data derived from a Collection

Can create a pipeline of function that can be evaluated

Intermediate result

Lazy evaluated by nature

Can transform data, cannot mutate data

JAVA Streams

```
stringLists.stream()  
    .map(str -> str.toUpperCase())  
    .collect(Collectors.toList());
```

Intermediate

filter
distinct
map
flatMap
limit
skip
sorted
peek

Terminal

reduce
collect
toArray
count
max
min
findAny
findFirst
forEach
allMatch
anyMatch

Only use a Stream once

```
List<Beer> beers = getBeers();  
Stream<Beer> beerStream = beers.stream();  
  
beerStream.forEach(b ->System.out.println(b.getName())); //1  
  
beerStream.forEach(b ->System.out.println(b.getAlcohol())); //2
```

Line 2 will give:

java.lang.IllegalStateException: stream has already been operated upon or closed

Be careful with returning a Stream

```
public Stream<Beer> getMeMyBeers ()  
  
public void execute () {  
    getMeMyBeers () //don't know if it is consumed yet!!!!  
    ...  
}
```

Returning a Stream

Private intermediate function

When new stream is created every time

When result is very large or might be infinite

By default return a collection



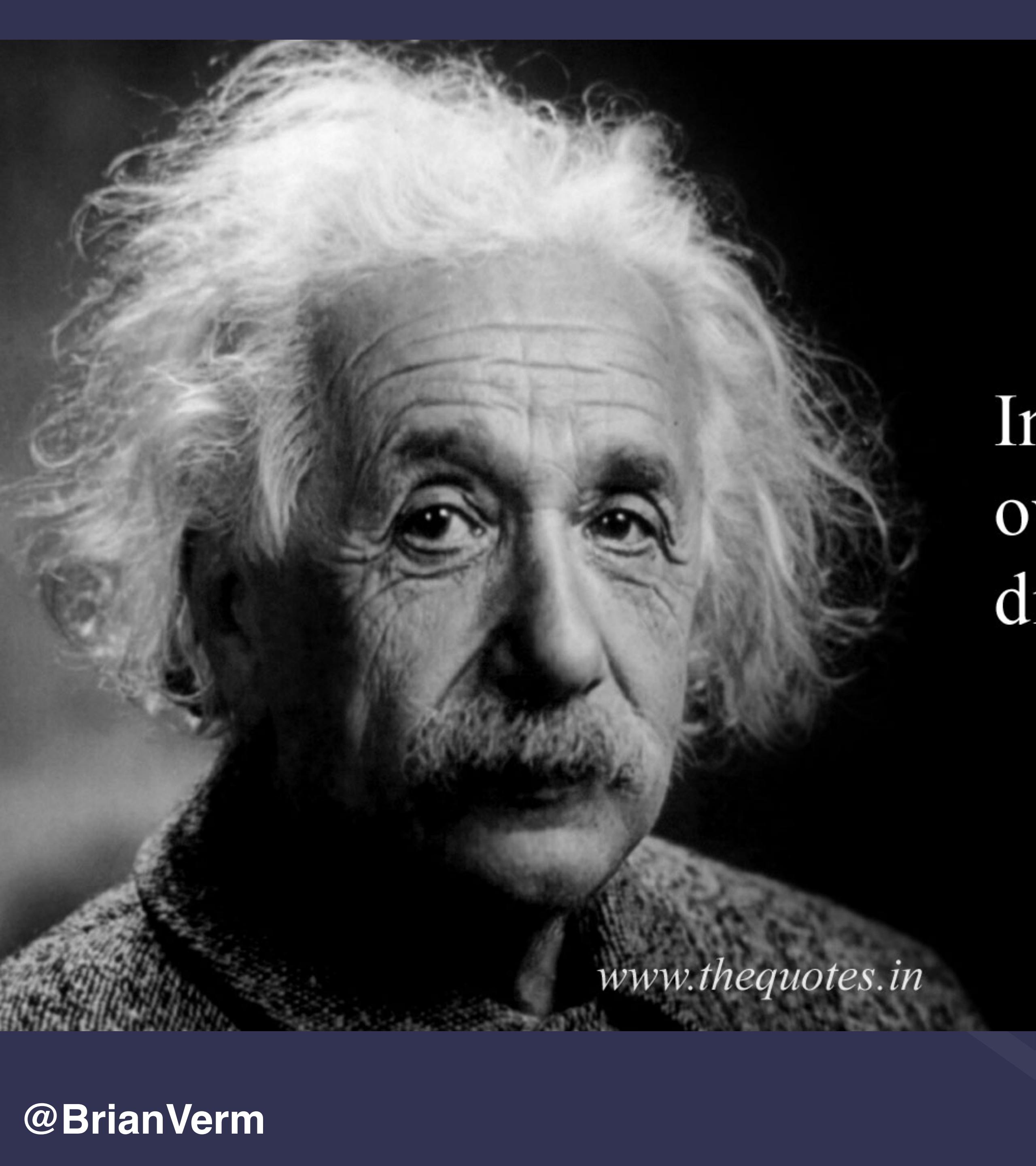
Mutable Object

Functional Programming

In computer science, functional programming is a programming paradigm style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical **functions** and **avoids changing-state** and **mutable data**.

It is a **declarative** programming paradigm, which means programming is done with **expressions** or declarations instead of statements.

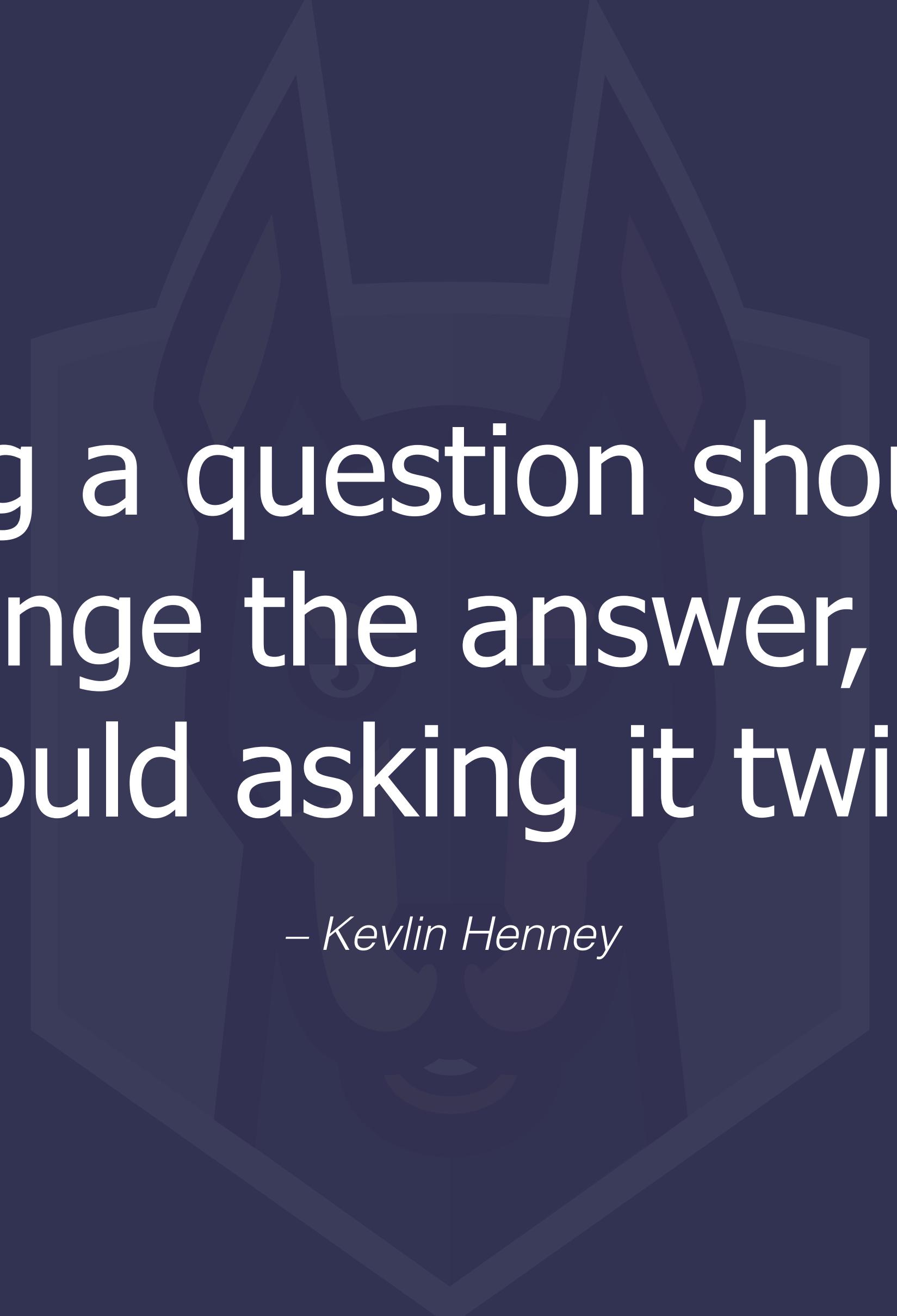
— wikipedia

A black and white close-up portrait of Albert Einstein. He has his characteristic wild, white hair and a full, grey beard. His eyes are looking slightly to the right of the camera with a thoughtful expression.

Insanity: doing the same thing
over and over again and expecting
different results.

Albert Einstein

www.thequotes.in



“Asking a question should not
change the answer, nor
should asking it twice”

– Kevlin Henney

Immutable objects

Less moving parts

Easier to reason about code

No need to keep a mental map of the state an object is in.

Stream cannot mutate

```
private final List<Beer> beers = List.of(new Beer("Heineken", 5.2),  
new Beer("Amstel", 5.1));  
  
public void execute() {  
    List<Beer> beersNew = beers.stream()  
        .map(beer -> beer.setName("foo")) //not allowed  
        .collect(Collectors.toList());  
}
```

Stream should not mutate !

```
private class Beer {  
    String name;  
    Double alcohol;  
  
    public Beer setName(String name) {  
        this.name = name;  
        return this;  
    }  
}  
  
private final List<Beer> beers = List.of(new Beer("Heineken", 5.2), new Beer("Amstel",  
5.1));  
  
public void execute() {  
    List<Beer> beersNew = beers.stream()  
        .map(beer -> beer.setName("foo"))  
        .collect(Collectors.toList());  
  
    System.out.println(beers);  
    System.out.println(beersNew);  
}
```



Stream should not mutate !

```
private class Beer {  
    String name;  
    Double alcohol;  
  
    public Beer setName(String name) {  
        this.name = name;  
        return this;  
    }  
}  
  
private final List<Beer> beers = List.of(new Beer("Heineken", 5.2), new Beer("Amstel",  
5.1));  
  
public void execute() {  
    List<Beer> beersNew = beers.stream()  
        .map(beer -> beer.setName("foo"))  
        .collect(Collectors.toList());  
  
    System.out.println(beers);  
    System.out.println(beersNew);  
}
```

Return a new copy

```
private class Beer {  
    String name;  
    Double alcohol;  
  
    public Beer withName(String name) {  
        return new Beer(name, this.alcohol);  
    }  
}  
  
private final List<Beer> beers = List.of(new Beer("Heineken", 5.2), new Beer("Amstel",  
5.1));  
  
public void execute() {  
    List<Beer> beersNew = beers.stream()  
        .map(beer -> beer.withName("foo"))  
        .collect(Collectors.toList());  
  
    System.out.println(beers);  
    System.out.println(beersNew);  
}
```



Overusing forEach

forEach()

Terminal functional on a stream.

Takes a consumer.

Can be used to apply side effects.

for-loop without the external iterator

simple forEach example

```
List<String> names = List.of("James", "Trisha", "Joshua",
"Jessica", "Simon", "Heather", "Roberto");

names.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

mutation with forEach

```
List<Beer> beers = List.of(new Beer("Heineken", 5.2), new  
Beer("Amstel", 5.1));  
  
beers.stream()  
    .forEach(beer -> beer.setAlcohol(0.0));
```

overusing forEach

```
private class Beer {  
    String name;  
    Double alcohol;  
    List<String> reviews;  
    Integer rating;  
}  
  
List<Beer> beers = List.of(new Beer("Heineken", 5.2), new Beer("Amstel", 5.1));  
  
//enrich with ratings  
beers.stream().forEach(beer -> beer.setRating(findRating(beer.getName())));  
  
//enrich with reviews  
beers.stream().forEach(beer -> beer.setReviews(findReviews(beer.getName())));  
  
...
```



Order of operations

Assignment

Beers -> Brewer -> Country

I want the first 3 unique brewers countries from the beer library as comma separated String

```
beerLib.stream()
    .map(Beer::getBrewer)
    .distinct()
    .limit(3)
    .map(Brewer::getCountry)
    .map(String::toUpperCase)
    .collect(Collectors.joining(","));
```

Assignment

Beers -> Brewer -> Country

I want the first 3 unique brewers countries from the beer library as comma separated String

```
beerLib.stream()
    .map(Beer::getBrewer)
    .distinct()
    .limit(3)
    .map(Brewer::getCountry)
    .map(String::toUpperCase)
    .collect(Collectors.joining(","));
// wrong
```

Assignment

Beers -> Brewer -> Country

I want the first 3 unique brewers countries from the beer library as comma separated String

```
beerLib.stream()
    .map(Beer::getBrewer)
    .map(Brewer::getCountry)
    .map(String::toUpperCase)
    .distinct()
    .limit(3)
    .collect(Collectors.joining(","));
// correct
```

Infinite stream

```
IntStream.iterate(0, i -> (i + 1) % 2)
    .distinct()
    .limit(10)
    .forEach(i -> System.out.println(i));
```

Infinite stream

```
IntStream.iterate(0, i -> (i + 1) % 2)
    .distinct()
    .limit(10)
    .forEach(i -> System.out.println(i));

// will run forever
```

```
IntStream.iterate(0, i -> (i + 1) % 2)
    .limit(10)
    .distinct()
    .forEach(i -> System.out.println(i));

//will terminate
```

Infinite stream

```
IntStream.iterate(0, i -> (i + 1) % 2)
    .parallel()
    .distinct()
    .limit(10)
    .forEach(i -> System.out.println(i));

// will run forever on all threads.
```

Solution

1. Look closely at the order of operations
 - prevent incorrect answers
2. Only use infinite streams when absolutely necessary.

rather use:

```
IntStream.range(0,10);  
IntStream.rangeClosed(0,10);  
IntStream.iterate(0, i -> i < 10, i -> i + 1);      //java 9 and up
```



Create higher-order functions

Higher-Order function

Taking a function as a parameter of a method

Insert behaviour instead of just data

A new level of abstraction

Reduce boilerplate / duplicate code

Duplicate code

```
private void logUpper(String str) {  
    //super interesting code  
    System.out.println(str.toUpperCase());  
    //even more interesting code  
}  
  
private void logLower(String str) {  
    //super interesting code  
    System.out.println(str.toLowerCase());  
    //even more interesting code  
}
```

Insert behaviour

```
private void logUpper(String string) {  
    doSuperCoolStuff(string, s -> s.toUpperCase());  
}  
  
private void logLower(String string) {  
    doSuperCoolStuff(string, s -> s.toLowerCase());  
}  
  
private void doFoo(String str, Function<String, String> func) {  
    //super interesting code  
    System.out.println(func.apply(str));  
    //even more interesting code  
}
```

Real situation (Webshop Basket)

MongoDB primary (read/write), multiple secondary (read)

Writing only on the primary

Mutating a basket:

Read from Primary

Edit

Run rule engine

Save to Primary

Real situation (Webshop Basket)

MongoDB primary (read/write), multiple secondary (read)

Writing only on the primary

Mutating a basket:

Read from Primary

Edit

Run rule engine

Save to Primary

Mutator

```
public void mutatorById(String id, Consumer<Basket> action) {  
    Basket basket = repoPrimary.findById(id);  
    action.accept(basket);  
    RuleEngine.run(basket)  
    repoPrimary.save(basket);  
}
```

Mutator

```
public void mutatorById(String id, Consumer<Basket> action) {  
    Basket basket = repoPrimary.findById(id);  
    action.accept(basket);  
    RuleEngine.run(basket)  
    repoPrimary.save(basket);  
}  
  
public void addItem(String uuid, Item item) {  
    mutatorById(uuid, basket -> basket.addItem(item));  
}  
  
public void removeItem(String uuid, Item item) {  
    mutatorById(uuid, basket -> basket.removeItem(item));  
}
```



Getting an Optional

Optional

Java's implementation of the Maybe Monad

Encapsulation to handle possible null value

Consider it a wrapper where a value can be absent

Force the user to unpack the Optional before using it.

Optional

```
Optional<String> c = null //please avoid this
```

Unpack Optional

```
public void execute() {  
    Optional<String> maybeString = getText();  
    String unpacked = maybeString.get();  
}
```

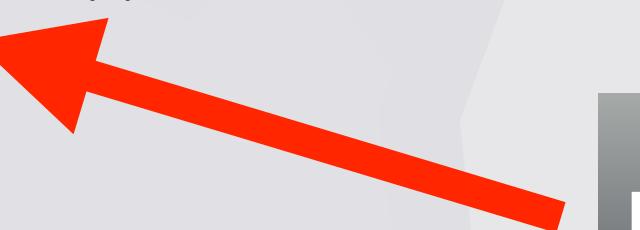
Unpack Optional

```
public void execute() {  
    Optional<String> maybeString = getText();  
    String unpacked = maybeString.get();  
}
```

NoSuchElementException

Unpack Optional

```
public void execute() {  
    Optional<String> maybeString = getText();  
    String unpacked = maybeString.get();  
}
```



NoSuchElementException

```
public void execute() {  
    Optional<String> maybeString = getText();  
    if (maybeString.isPresent()) {  
        String unpacked = maybeString.get();  
    }  
}
```

Unpack Optional

```
public void execute() {  
    Optional<String> maybeString = getText();  
    maybeString.ifPresent( str -> /* doSomething */ );  
}
```

```
public void execute() {  
    Optional<String> maybeString = getText();  
    maybeString.map(str -> str + ".");  
}
```



What else ...??

Alternative flow

- orElse()
- orElseGet()
- orElseThrow()



orElseThrow

```
public void execute() {  
    Optional<String> maybeString = Optional.empty();  
    maybeString  
        .map(this::runIfExist)  
        .orElseThrow(() -> new RuntimeException("Optional was empty"));  
}
```

orElse

```
public void execute() {  
    Optional<String> maybeString = Optional.of("foo");  
    String newString = maybeString  
        .map(this::runIfExist)  
        .orElse(runIfEmpty());  
    System.out.println(newString);  
}  
  
private String runIfExist(String str) {  
    System.out.println("only run if optional is filled ");  
    return str;  
}  
  
private String runIfEmpty() {  
    System.out.println("only run if empty");  
    return "empty";  
}
```

orElse

```
public void execute() {  
    Optional<String> maybeString = Optional.of("foo");  
    String newString = maybeString  
        .map(this::runIfExist)  
        .orElse(runIfEmpty());  
    System.out.println(newString);  
}  
  
private String runIfExist(String str) {  
    System.out.println("only run if optional is filled ");  
    return str;  
}  
  
private String runIfEmpty() {  
    System.out.println("only run if empty");  
    return "empty";  
}
```

only run if optional is filled
only run if empty
foo

orElseGet

```
public void execute() {  
    Optional<String> maybeString = Optional.of("foo");  
    String newString = maybeString  
        .map(this::runIfExist)  
        .orElseGet(() -> runIfEmpty());  
    System.out.println(newString);  
}  
  
private String runIfExist(String str) {  
    System.out.println("only run if optional is filled ");  
    return str;  
}  
  
private String runIfEmpty() {  
    System.out.println("only run if empty");  
    return "empty";  
}
```

orElseGet

```
public void execute() {  
    Optional<String> maybeString = Optional.of("foo");  
    String newString = maybeString  
        .map(this::runIfExist)  
        .orElseGet(() -> runIfEmpty());  
    System.out.println(newString);  
}  
  
private String runIfExist(String str) {  
    System.out.println("only run if optional is filled ");  
    return str;  
}  
  
private String runIfEmpty() {  
    System.out.println("only run if empty");  
    return "empty";  
}
```

only run if optional is filled
foo

what else?

- When using `orElse(x)`, make sure x doesn't contain any side effects
- Only use `orElse()` to assign a default value
- Use `orElseGet()` to run an alternative flow

List of Optionals

```
beerLib.stream()  
    .map(Beer::getDescription) //returns optional
```

List of Optionals

```
//java 8 style  
  
beerLib.stream()  
    .map(Beer::getDescription) //returns optional  
    .filter(Optional::isPresent)  
    .map(Optional::get)  
    .forEach(System.out::println);
```

List of Optionals

```
//java 8 flatMap  
  
beerLib.stream()  
    .map(Beer::getDescription) //returns optional  
    .flatMap(o -> o.map(Stream::of).orElse(Stream.empty()))  
    .forEach(System.out::println);
```

List of Optionals

```
//java 9 flatMap  
  
beerLib.stream()  
    .map(Beer::getDescription) //returns optional  
    .flatMap(Optional::stream)  
    .forEach(System.out::println);
```



Exceptions

Checked Exceptions & Lambda

```
public Beer doSomething(Beer beer) throws IsEmptyException { ... }

Function <Beer,Beer> fBeer = beer -> doSomething(beer)
```

Checked Exceptions & Lambda

```
public Beer doSomething(Beer beer) throws IsEmptyException { ... }

Function <Beer,Beer> fBeer = beer -> doSomething(beer)
```

Checked Exceptions & Lambda

```
public Beer doSomething(Beer beer) throws IsEmptyException { ...}

beerLib.stream()
    .map(beer -> {
        try{
            return doSomething(beer);
        } catch (IsEmptyException e) {
            throw new RuntimeException(e);
        }
    })
    .collect(Collectors.toList());

//not very pretty
```

Checked Exceptions & Lambda

```
public Beer doSomething(Beer beer) throws IsEmptyException { ...}

private Beer wrappedDoSomeThing(Beer beer) {
    try{
        return doSomething(beer);
    } catch (IsEmptyException e) {
        throw new RuntimeException(e);
    }
}

beerLib.stream()
    .map(this::wrappedDoSomeThing)
    .collect(Collectors.toList());
```

Exception Utility

```
@FunctionalInterface
public interface CheckedFunction<T, R> {
    public R apply(T t) throws Exception;
}

public static <T, R> Function<T, R> wrap(CheckedFunction<T, R> function) {
    return t -> {
        try {
            return function.apply(t);
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
    };
};

beerLib.stream()
    .map(wrap(beer -> doSomething(beer)))
    .collect(Collectors.toList());
```

Either type

```
public class Either<L, R> {

    private final L left;
    private final R right;

    private Either(L left, R right) {
        this.left = left;
        this.right = right;
    }

    public static <L,R> Either<L,R> Left( L value) {
        return new Either(value, null);
    }

    public static <L,R> Either<L,R> Right( R value) {
        return new Either(null, value);
    } ...
}
```

Either type

```
private Either<Exception, String> canGoWrong(Integer input) {  
    if (input > 10) {  
        return Either.Left(new RuntimeException("larger then 10"));  
    }  
    return Either.Right("["+input+"]");  
}  
  
List<Either<Exception, String>> canGoWrongs = IntStream.range(0,12)  
    .mapToObj(i -> canGoWrong(i))  
    .collect(Collectors.toList());
```

Either type

```
private Either<Exception, String> canGoWrong(Integer input) {  
    if (input > 10) {  
        return Either.Left(new RuntimeException("larger than 10"));  
    }  
    return Either.Right("["+input+"]");  
}  
  
List<Either<Exception, String>> canGoWrongs = IntStream.range(0,12)  
    .mapToObj(i -> canGoWrong(i))  
    .collect(Collectors.toList());  
  
canGoWrongs.stream()  
    .map(e -> e.mapRight(s -> s.toUpperCase()))  
    .flatMap(o -> o.map(Stream::of).orElseGet(Stream::empty))  
    .forEach(System.out::println);
```

Try

- Failure (Exception)
- Success (Type)
- Vavr



Try

- Failure (Exception)
- Success (Type)
- VAVR

```
List<Try<String>> output = teams.stream()  
    .map(CheckedFunction1.liftTry(this::mayThrowException))  
    .collect(Collectors.toList());
```

THE SECURE DEVELOPER



An inclusive community
that educates and
enables developers in
application security

<https://www.thesecuredveloper.com>

Brian Vermeer

@BrianVerm
brianvermeer@snyk.io



snyk

@BrianVerm

WIN
Apple®
AirPods®
DRAWING * FRIDAY * 1PM



Compliments of:

snyk