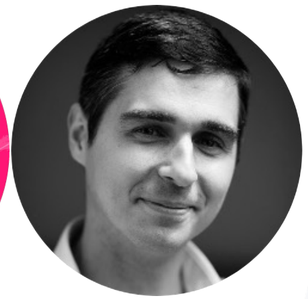
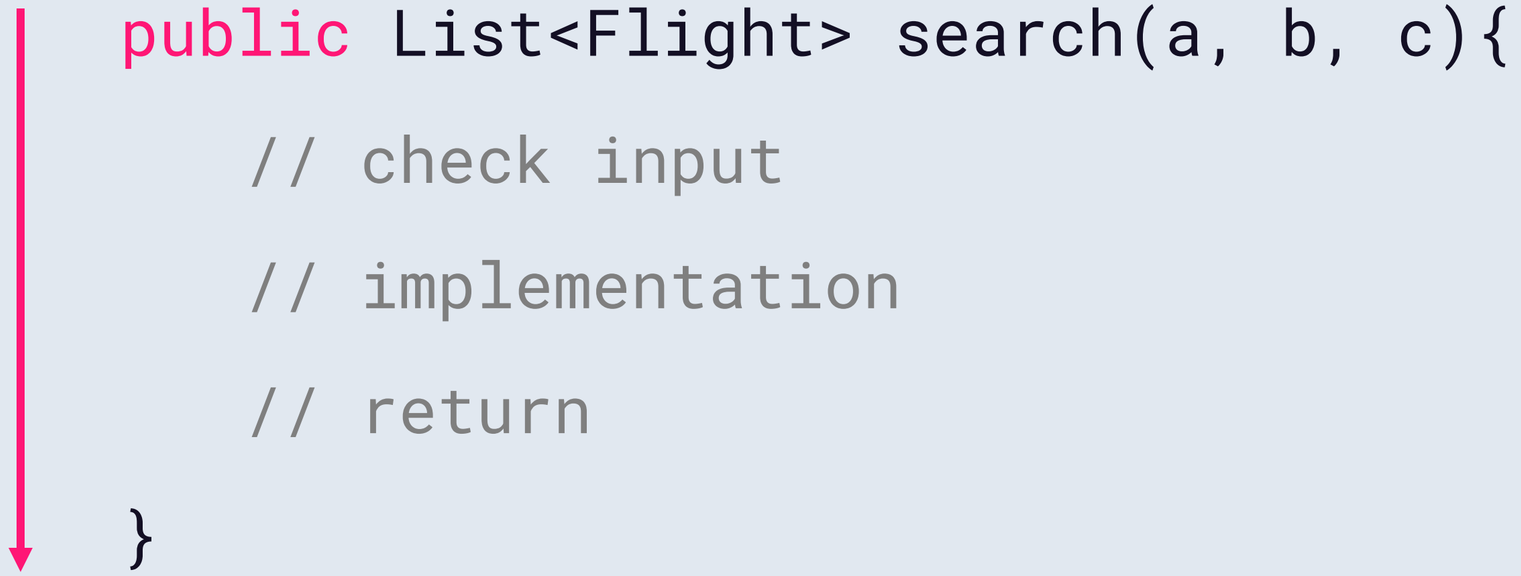
Implementing Methods Best Practices



Andrejs Doronins
Software Developer in Test

```
var flightSearch = FlightSearch.newSearch();

flightSearch.search(...);
```


public List<Flight> search(a, b, c){
// check input
// implementation
// return
}

Overview

- Methods should do one thing
- Avoid too many parameters
- Flag arguments are a poor choice
- Prefer enums over strings
- Fail fast principle in methods
- Useful concepts for local vars
- What to return, and what not to return

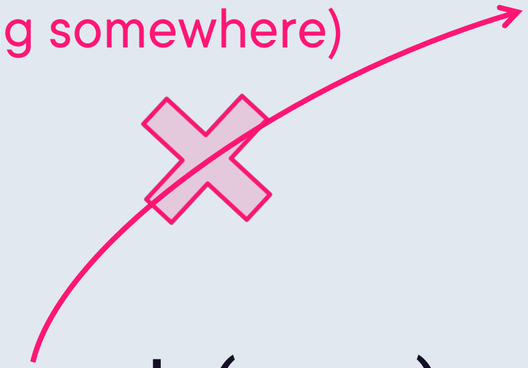
Command Query Separation

```
void update(Thing t) {
    //...
}
```

```
SomeObj obj;

int compute(Thing t) {
   update(obj);
   //...
   return i;
}
```

(and quietly update something somewhere)



```
var flights = flightSearch.search(...);
```

Change state OR return a value

But not both

```
calculatePrice(new Flight(“A”,“B”));
```

```
calculatePrice(10, false,
                    “US”, new Flight(“A”,“B”));
```

How many parameters should we have?

Generally, fewer parameters is better.

Number of Parameters

OK	Avoid	Refactor!
0-2	3	4+

Single parameter examples:

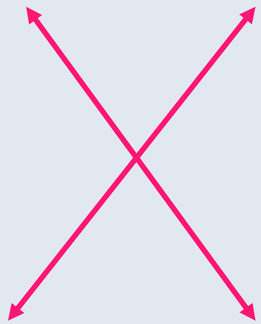
```
canReadFile(“file.csv”);
getFlightsForYear(2021);
getBookingFor(john);
```

Two parameters examples:

```
add(2,3);
assertEquals(a,b);
```

```
// JUnit
assertEquals(expected, actual);

// TestNG
assertEquals(actual, expected);
```



Confusion starts at 2 parameters of the same type

Methods with 3+ Arguments Might:

Do too many things (split it)

Take too many primitive types (pass a single object)

Takes a boolean (flag) argument

```
switchLights(room, true);
```


```
switchLightsOn(room);
```

```
switchLightsOff(room);
```

```
void switchLights(Room room,
                  boolean on){
    if(on){ // ... }
    else { // ... }
}
```

Simpler method signature

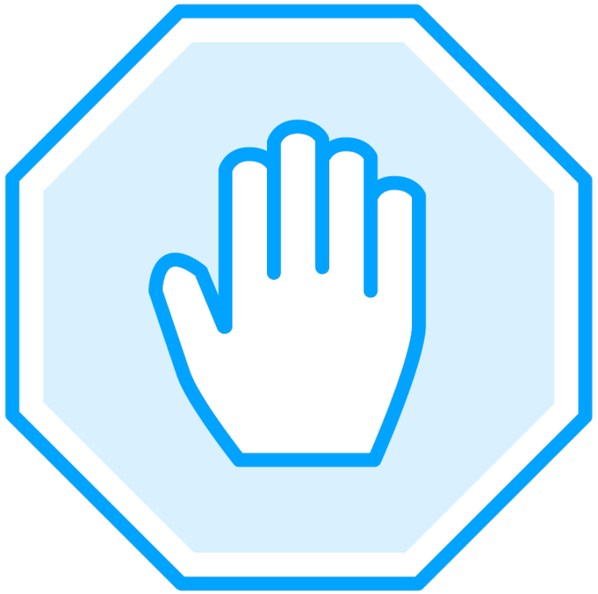
Simpler implementation

Robert C. Martin

“Flag arguments are ugly.

It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing.”

Restriction Is Good



Restriction brings:

- predictability
- safety
- bug prevention

Prefer Date-like types over Strings

Prefer Enums over Strings

Compiler Restrictions

```
List<Integer> ints = List.of("a");
```

```
                    2.5
```

```
int convert(String v) {
    // ...
}
```

```
                                        LocalDate date
search(String from, String to, ~~String date~~)
```

```
class SearchRequest {
   SearchRequest() {
      // fail fast
   }
}

List<Flight> search(SearchRequest request) {
   // fail fast
}
```

Easier troubleshooting

```
class SearchRequest {
    SearchRequest(...) {
    }
}
```

```
if(input == null)
```

```
if(input == null)
```

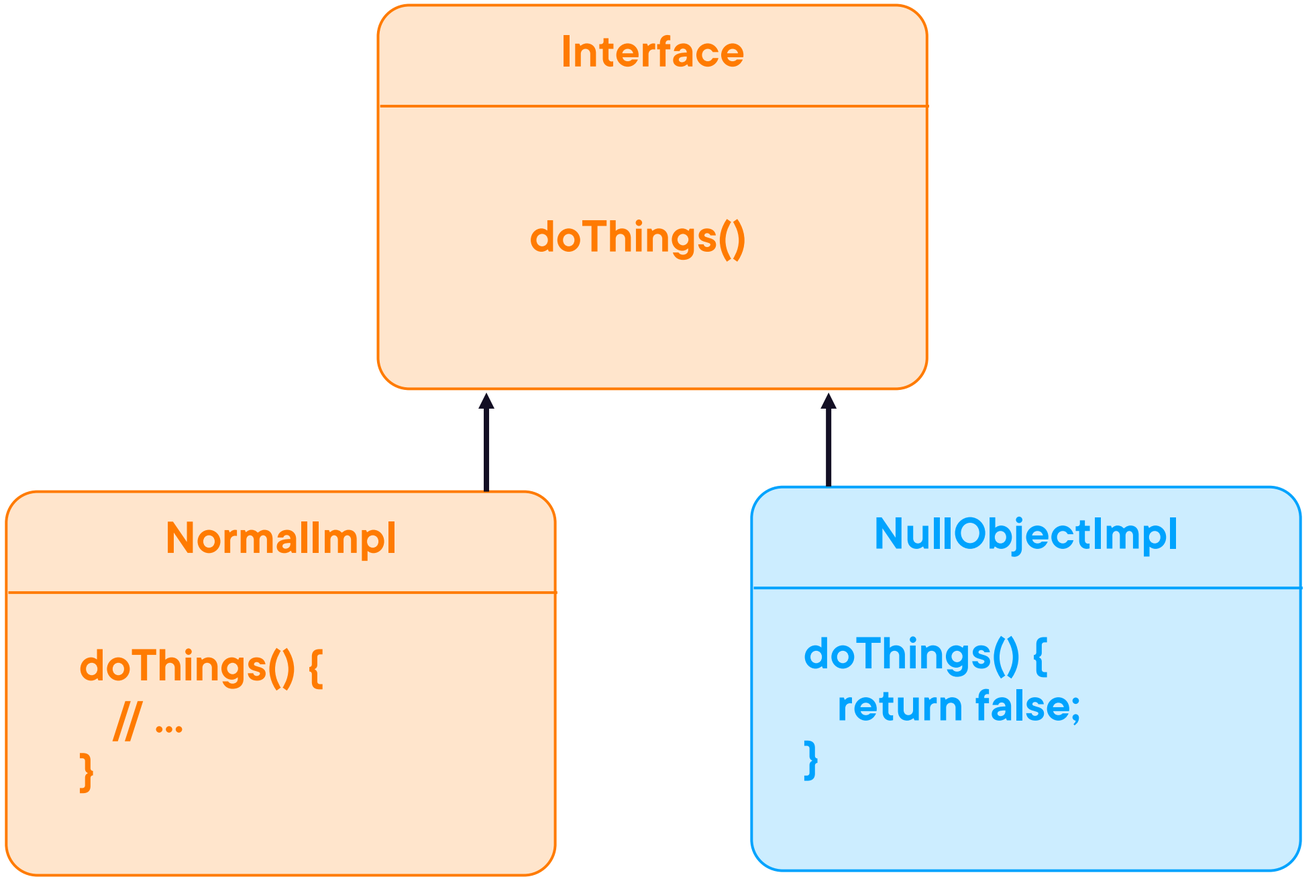
```
if(input == null)
```

```
class SearchRequest {
    SearchRequest(...) {

    requireNonNull(input);
    }
}
```

```
List<Flight> search(...) {
  if(searchReq == null){
  }
}
```

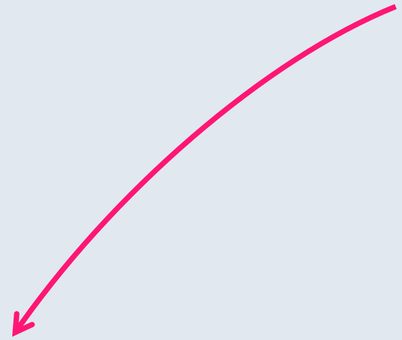
Null Object pattern


Interface
doThings()
NormalImpl
doThings() {
// ...
}
NullObjectImpl
doThings() {
return false;
}

Java is such a verbose language

Not true anymore!

Local Variable Type Inference



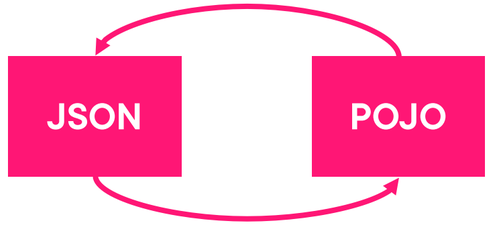
```
var flights = getFlights();


~~String~~ var name; // no improvement


~~BufferedReader~~ var reader = new BufferedReader(...); // yes!
```

Demo

Jackson library



Careful!

```
// java.util.Date ?

// java.time.LocalDate ?

// java.time.LocalDateTime ?

var today = getToday();
```

Expensive Object Creation

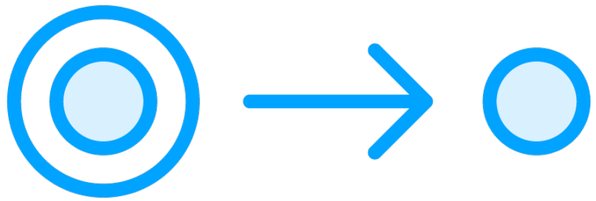
```
int countMatches(String text) {
     Pattern pattern = Pattern.compile("myRegex"); // new obj at every invocation
     Matcher matcher = pattern.matcher(text);
     //...
}
```

Expensive Object Creation

```
// create once and reuse
static final Pattern pattern = Pattern.compile("myRegex");


int countMatches(String text) {
    Matcher matcher = pattern.matcher(text);
    //...
}
```

Returning



Methods may return:

- primitives
- objects
- null

Don’t return null !

Don’t return codes with special meaning (-1, 0)

```
List<String> getSomeData() {

try{ // read from DB }

 catch {

    // operation failed

     return null;

     return Collections.emptyList();

    }

}
```

Leads to either:

NullPointerException

or

```
if (list != null)

if (list != null)

if (list != null)

if (list != null)
```

Client Code Is Simplified

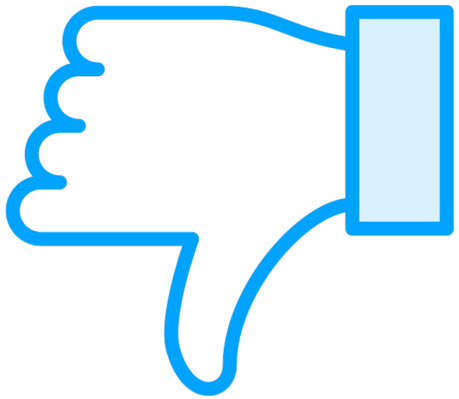
```
if(list != null && list.isEmpty())
```

```
int withdraw(int amount) {
  if (amount > balance) {
    return -1;
  }
  else {
    balance -= amount;
    return 0;
  }
}
```

```
void withdraw(int amount) throws
      InsufficientFundsException {
  if (amount > balance) {
    throw new InsufficientFundsException();
  }
  balance -= amount;
}
```

Also breaks the CQS principle

Magic Numbers



Force programmers to learn their meaning

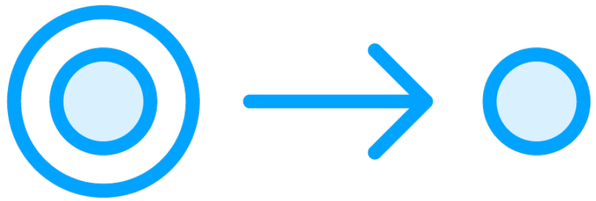
Results in poor client code

Checking for Magic Numbers

```
// resulting balance is -1?
// or does -1 have a special meaning?
if(withdraw(100) == -1) {


}
```

Alternatives to Null



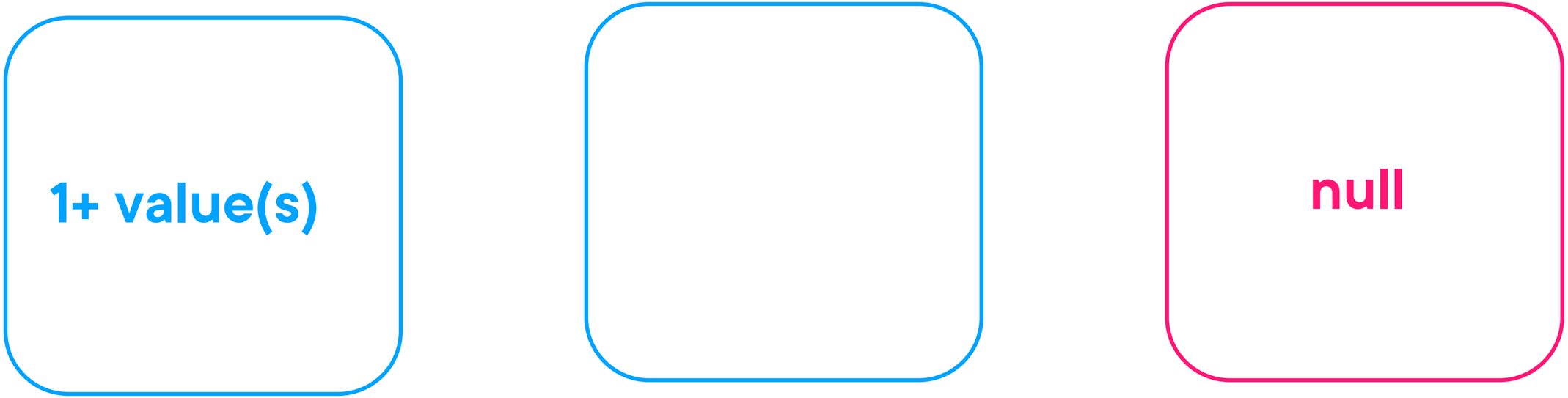
1) throw

2) Sensible default

3) Empty collection

4) Optional<T>

Collection



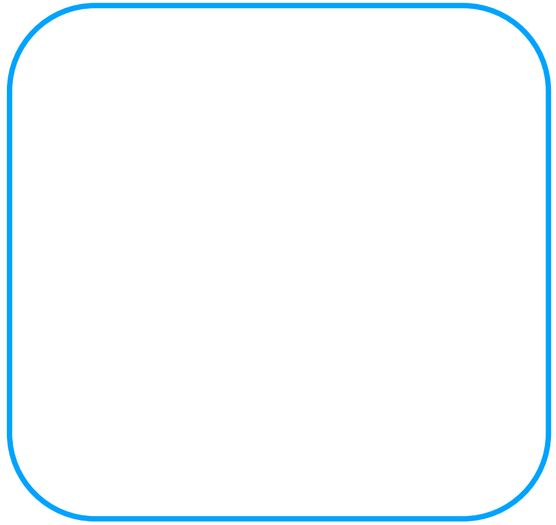
OK

OK

Not OK

Optional<T>

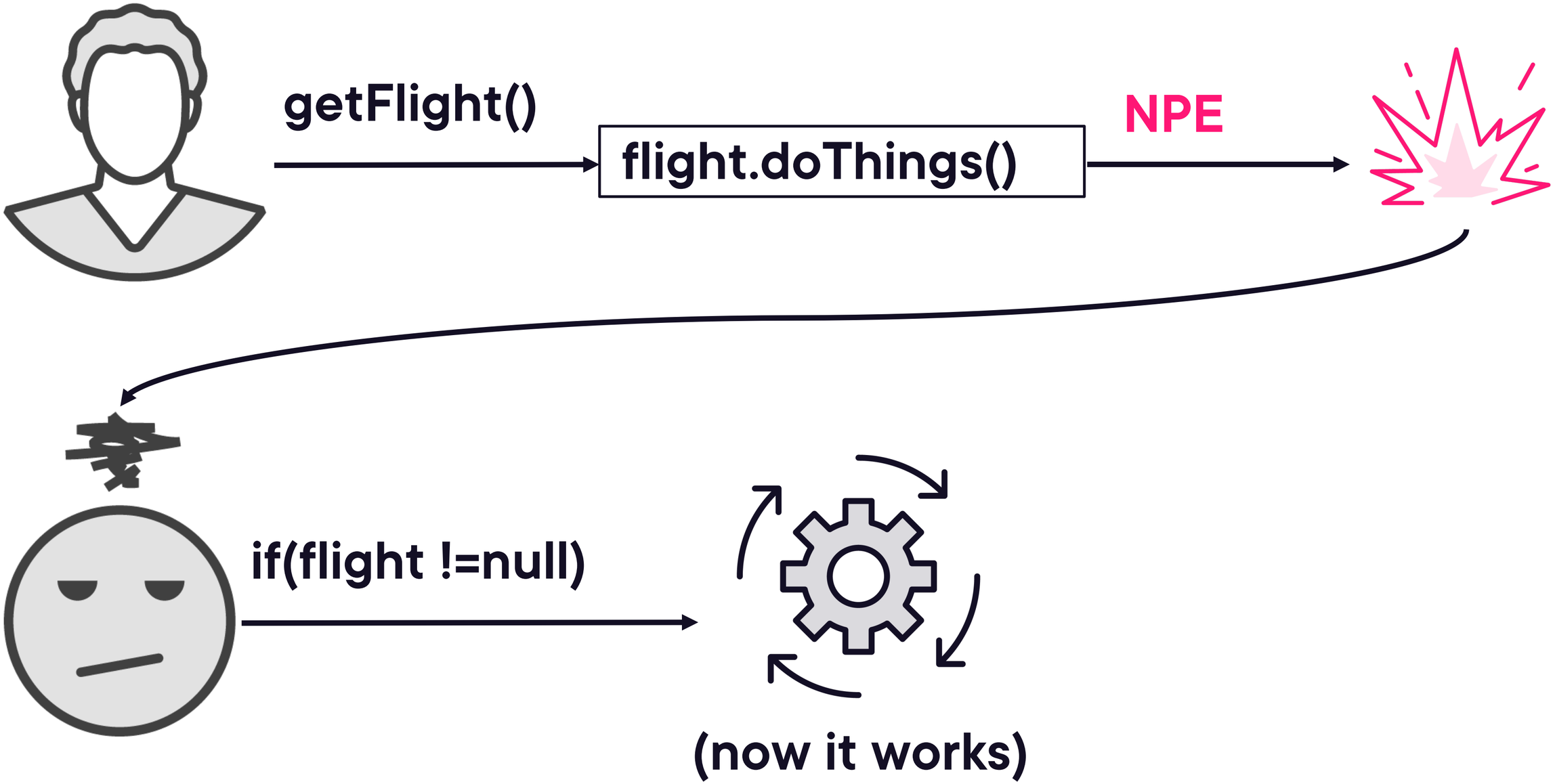
OK

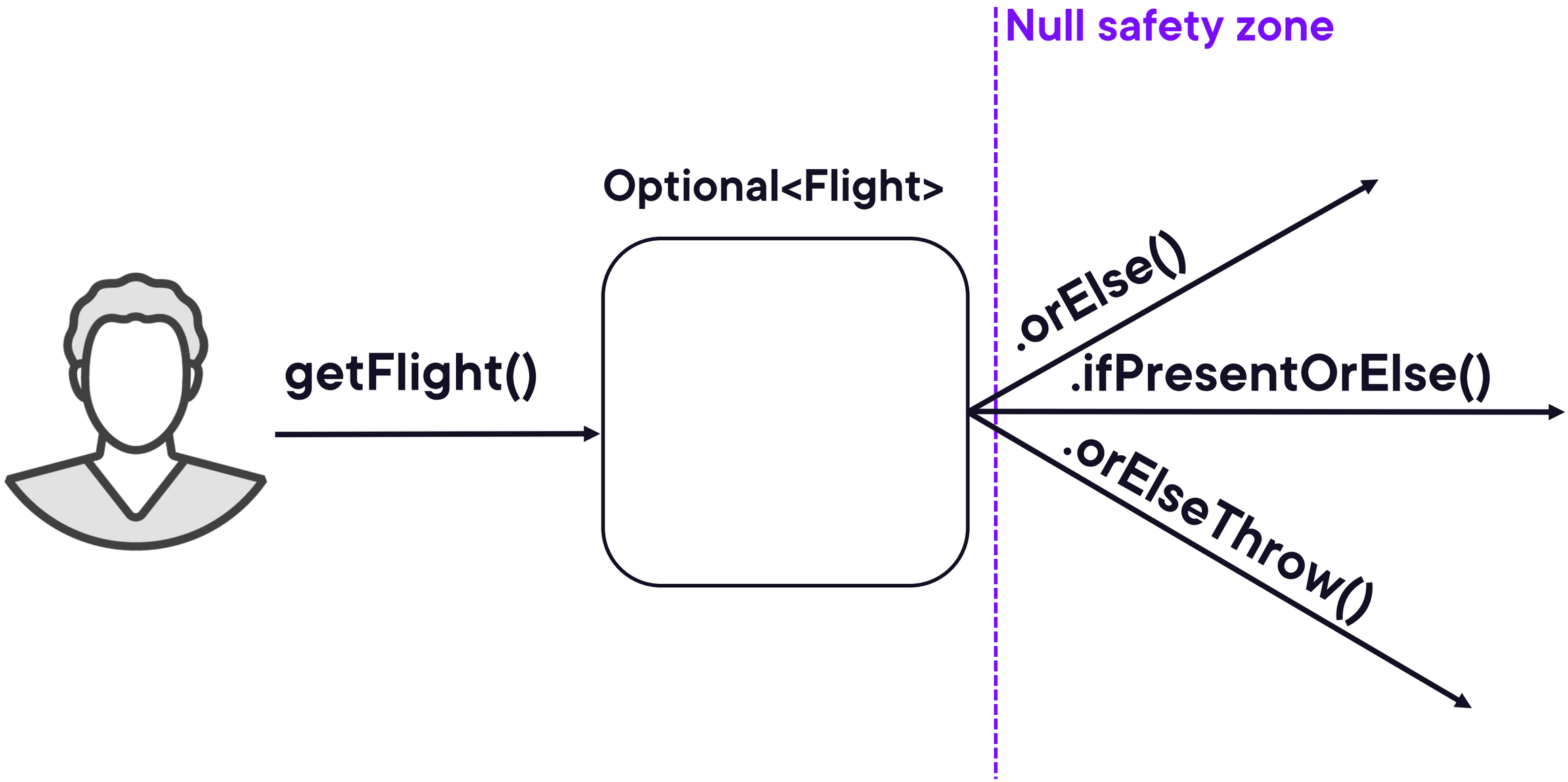


OK

Optional

A container object which may or may not contain a non-null value


getFlight()
flight.doThings()
NPE
if(flight !=null)
(now it works)


Null safety zone
Optional<Flight>
getFlight()
.orElse()
.ifPresentOrElse()
.orElseThrow()

Optional<T> forces the user to confront the fact that there may be no value

Avoiding Nulls with the Optional Type

Applying Functional Programming Techniques in Java

Esteban Herrera

Summary

- CQS: update or return, but not both
- Short(er) parameter lists
- Replace strings with enums, dates or other more restrictive types
- Fail fast
- Use var judiciously
- Don’t create unnecessary objects
- Don’t return nulls or magic numbers

Up Next:

Looking Closely at Strings and Numbers