Functional Idioms with Java

Fabian Böller

13.09.2018

Overview

Overview

Pure functions in the Stream API

```
list.map(/* Pure function only! */)
    .flatMap(/* Pure function only! */)
    .filter(/* Pure function only! */)
    .reduce(/* Pure function only! */)
    .collect(/* Pure function only! */)

list.forEach(/* Impure function */)
list.peek(/* Impure function */)
```

Pure functions in the Stream API

```
list.map(/* Pure function only! */)
    .flatMap(/* Pure function only! */)
    .filter(/* Pure function only! */)
    .reduce(/* Pure function only! */)
    .collect(/* Pure function only! */)

list.forEach(/* Impure function */)
list.peek(/* Impure function */)
```

Reasons for impurity

1. Execution of side effects

Reasons for impurity

1. Execution of side effects
2. Modification of input parameters

Reasons for impurity

1. Execution of side effects
2. Modification of input parameters
3. Output does not only depend on input

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Impure function

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    userDAO.save(user);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    // Mock UserDAO here
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    // Retrieve User here
    assertEquals("Heinz", user.getName());
}
```

Removing side effects

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    assertEquals("Heinz", user.getName());
}
```

Removing side effects

```
static void createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    collector.addAll(/* Some events */);
}

@Test
public void testUserCreation() {
    EventCollector eventCollector = new EventCollector();
    createUser("Heinz", eventCollector);
    assertEquals("Heinz", user.getName());
}
```

Return user

```
static User createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    collector.addAll(/* Some events */);
    return user;
}

@Test
public void testUserCreation() {
    EventCollector eventCollector = new EventCollector();
    User user = createUser("Heinz", eventCollector);
    assertEquals("Heinz", user.getName());
}
```

Return user

```
static User createUser(String name, EventCollector collector) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    collector.addAll(/* Some events */);
    return user;
}

@Test
public void testUserCreation() {
    EventCollector eventCollector = new EventCollector();
    User user = createUser("Heinz", eventCollector);
    assertEquals("Heinz", user.getName());
}
```

Removing modification of input

```
static Pair<User, List<Event>> createUser(String name) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    return Pair.of(user, /* Some events */);
}

@Test
public void testUserCreation() {
    Pair<User, List<Event>> pair = createUser("Heinz");
    assertEquals("Heinz", user.getName());
}
```

Simplifying

```
static User createUser(String name) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    return user;
}

static List<Event> userCreationEvents
    (User user) { /* Some events */ }

@Test
public void testUserCreation() {
    User user = createUser("Heinz");
    assertEquals("Heinz", user.getName());
}
```

Simplifying

```
static User createUser(String name) {
    User user = new User();
    /* Some complex setup */
    user.setId(randomUUID());
    user.setName(name);
    return user;
}

static List<Event> userCreationEvents
    (User user) { /* Some events */ }

@Test
public void testUserCreation() {
    User user = createUser("Heinz");
    assertEquals("Heinz", user.getName());
}
```

Simplifying

```
static User createUser(String name, UUID uuid) {
    User user = new User();
    /* Some complex setup */
    user.setId(uuid);
    user.setName(name);
    return user;
}

@Test
public void testUserCreation() {
    User user = createUser("Heinz", randomUUID());
    assertEquals("Heinz", user.getName());
}
```

Testing for user equality

```
static User createUser(String name, UUID uuid) {
    User user = new User();
    /* Some complex setup */
    user.setId(uuid);
    user.setName(name);
    return user;
}

@Test
public void testUserCreation() {
    User user = createUser("Heinz", randomUUID());
    assertEquals(USER_HEINZ, user);
}
```

Testing for user equality

```
static User createUser(String name, UUID uuid) {
    User user = new User();
    /* Some complex setup */
    user.setId(uuid);
    user.setName(name);
    return user;
}

@Test
public void testUserCreation() {
    User user = createUser("Heinz", randomUUID());
    assertEquals(USER_HEINZ, user);
}
```

Documentation by signature

```
static <T> List<T> f(List<T> l, Predicate<T> p) { /* ... */ }

static <T> Optional<T> g(List<T> l, Predicate<T> p) { /* ... */ }

static <T> int h(List<T> l) { /* ... */ }
```

Documentation by signature

```
static <T> List<T> f(List<T> l, Predicate<T> p) { /* ... */ }

static <T> Optional<T> g(List<T> l, Predicate<T> p) { /* ... */ }

static <T> int h(List<T> l) { /* ... */ }
```

Documentation by signature

```
static <T> List<T> f(List<T> l, Predicate<T> p) { /* ... */ }

static <T> Optional<T> g(List<T> l, Predicate<T> p) { /* ... */ }

static <T> int h(List<T> l) { /* ... */ }
```

Overview

Working on a list the traditional way

```
static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    Map<User, Double> results = new HashMap<>();
    for (User user: users) {
        if (user.getName().equals("Heinz")) {
            double sumAge = 0;
            for (User friend: user.getFriends()) {
                sumAge += friend.getAge();
            }
            double averageAge = sumAge / user.getFriends().size();
            results.put(user, averageAge);
        }
    }
    return results;
}
```

Working on a list the traditional way

```
static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    Map<User, Double> results = new HashMap<>();
    for (User user: users) {
        if (user.getName().equals("Heinz")) {
            double sumAge = 0;
            for (User friend: user.getFriends()) {
                sumAge += friend.getAge();
            }
            double averageAge = sumAge / user.getFriends().size();
            results.put(user, averageAge);
        }
    }
    return results;
}
```

Working on a list the traditional way

```
static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    Map<User, Double> results = new HashMap<>();
    for (User user: users) {
        if (user.getName().equals("Heinz")) {
            double sumAge = 0;
            for (User friend: user.getFriends()) {
                sumAge += friend.getAge();
            }
            double averageAge = sumAge / user.getFriends().size();
            results.put(user, averageAge);
        }
    }
    return results;
}
```

Working on a list the traditional way

```
static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    Map<User, Double> results = new HashMap<>();
    for (User user: users) {
        if (user.getName().equals("Heinz")) {
            double sumAge = 0;
            for (User friend: user.getFriends()) {
                sumAge += friend.getAge();
            }
            double averageAge = sumAge / user.getFriends().size();
            results.put(user, averageAge);
        }
    }
    return results;
}
```

Working on a list the traditional way

```
static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    Map<User, Double> results = new HashMap<>();
    for (User user: users) {
        if (user.getName().equals("Heinz")) {
            double sumAge = 0;
            for (User friend: user.getFriends()) {
                sumAge += friend.getAge();
            }
            double averageAge = sumAge / user.getFriends().size();
            results.put(user, averageAge);
        }
    }
    return results;
}
```

Working on a list the traditional way

```
static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    Map<User, Double> results = new HashMap<>();
    for (User user: users) {
        if (user.getName().equals("Heinz")) {
            double sumAge = 0;
            for (User friend: user.getFriends()) {
                sumAge += friend.getAge();
            }
            double averageAge = sumAge / user.getFriends().size();
            results.put(user, averageAge);
        }
    }
    return results;
}
```

Working on a list the traditional way

```
static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    Map<User, Double> results = new HashMap<>();
    for (User user: users) {
        if (user.getName().equals("Heinz")) {
            double sumAge = 0;
            for (User friend: user.getFriends()) {
                sumAge += friend.getAge();
            }
            double averageAge = sumAge / user.getFriends().size();
            results.put(user, averageAge);
        }
    }
    return results;
}
```

Using the Stream API badly

```
static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    Map<User, Double> results = new HashMap<>();
    users.forEach(user -> {
        if (user.getName().equals("Heinz")) {
            double sumAge = 0;
            user.getFriends().forEach(friend -> {
                sumAge += friend.getAge();
            });
            double averageAge = sumAge / user.getFriends().size();
            results.put(user, averageAge);
        }
    });
    return results;
}
```

Using the Stream API

```
public static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    return users
        .stream()
        .filter(user -> user.getName().equals("Heinz"))
        .collect(toMap(
            user -> user,
            user -> user
                    .getFriends()
                    .stream()
                    .map(friend -> friend.getAge())
                    .collect(averagingInt(age -> age))
        ));
}
```

Using the Stream API

```
public static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    return users
        .stream()
        .filter(user -> user.getName().equals("Heinz"))
        .collect(toMap(
            user -> user,
            user -> user
                    .getFriends()
                    .stream()
                    .map(friend -> friend.getAge())
                    .collect(averagingInt(age -> age))
        ));
}
```

Using the Stream API

```
public static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    return users
        .stream()
        .filter(user -> user.getName().equals("Heinz"))
        .collect(toMap(
            user -> user,
            user -> user
                    .getFriends()
                    .stream()
                    .map(friend -> friend.getAge())
                    .collect(averagingInt(age -> age))
        ));
}
```

Using the Stream API

```
public static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    return users
        .stream()
        .filter(user -> user.getName().equals("Heinz"))
        .collect(toMap(
            user -> user,
            user -> user
                    .getFriends()
                    .stream()
                    .map(friend -> friend.getAge())
                    .collect(averagingInt(age -> age))
        ));
}
```

Using the Stream API

```
public static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    return users
        .stream()
        .filter(user -> user.getName().equals("Heinz"))
        .collect(toMap(
            user -> user,
            user -> user
                    .getFriends()
                    .stream()
                    .map(friend -> friend.getAge())
                    .collect(averagingInt(age -> age))
        ));
}
```

Using the Stream API

```
public static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    return users
        .stream()
        .filter(user -> user.getName().equals("Heinz"))
        .collect(toMap(
            user -> user,
            user -> user
                    .getFriends()
                    .stream()
                    .map(friend -> friend.getAge())
                    .collect(averagingInt(age -> age))
        ));
}
```

Using the Stream API

```
public static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    return users
        .stream()
        .filter(user -> user.getName().equals("Heinz"))
        .collect(toMap(
            user -> user,
            user -> user
                    .getFriends()
                    .stream()
                    .map(friend -> friend.getAge())
                    .collect(averagingInt(age -> age))
        ));
}
```

Using the Stream API

```
public static Map<User, Double>
    findFriendsAverageAgeOfUsersWithNameHeinz(List<User> users) {
    return users
        .stream()
        .filter(user -> user.getName().equals("Heinz"))
        .collect(toMap(
            user -> user,
            user -> user
                    .getFriends()
                    .stream()
                    .map(friend -> friend.getAge())
                    .collect(averagingInt(age -> age))
        ));
}
```

List vs. Streams

- List: Finite number of arbitrary elements

List vs. Streams

- List: Finite number of arbitrary elements
- Stream:

List vs. Streams

- List: Finite number of arbitrary elements
- Stream:
  - Finite number of arbitrary elements

List vs. Streams

- List: Finite number of arbitrary elements
- Stream:
  - Finite number of arbitrary elements
  - Infinitely many elements if they are finitely representable

Finitely representable

- $[1, 2, 3]$ is finite

Finitely representable

- $[1, 2, 3]$ is finite
- $[1, 2, 3, ...]$ is finitely representable

Finitely representable

- $[1, 2, 3]$ is finite
- $[1, 2, 3, ...]$ is finitely representable
- $[1, 2, 4, 8, ...]$ is finitely representable

Finitely representable

- $[1, 2, 3]$ is finite
- $[1, 2, 3, ...]$ is finitely representable
- $[1, 2, 4, 8, ...]$ is finitely representable
- $(1, i \rightarrow i * 2)$ is a different finite representation

Finitely representable

- $[1, 2, 3]$ is finite
- $[1, 2, 3, ...]$ is finitely representable
- $[1, 2, 4, 8, ...]$ is finitely representable
- $(1, i \rightarrow i * 2)$ is a different finite representation
- $[56, 44, 78, 15, ...]$ is not finitely representable

Why infinite lists

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    return primenumbers(/* How many? */)
        .stream()
        .filter(isAwesome)
        .collect(toList());
}
```

Why infinite lists

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    return primenumbers(/* How many? */)
        .stream()
        .filter(isAwesome)
        .collect(toList());
}
```

Why infinite lists

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    return primenumbers(/* How many? */)
        .stream()
        .filter(isAwesome)
        .collect(toList());
}
```

A solution without infinite streams

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    int multiplier = 1;
    List<Integer> result = new ArrayList<>();
    while (result.size() < n) {
        result = primenumbers(n * multiplier)
            .stream()
            .filter(isAwesome)
            .limit(n)
            .collect(toList());
        multiplier *= 2;
    }
    return result;
}
```

A solution without infinite streams

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    int multiplier = 1;
    List<Integer> result = new ArrayList<>();
    while (result.size() < n) {
        result = primenumbers(n * multiplier)
            .stream()
            .filter(isAwesome)
            .limit(n)
            .collect(toList());
        multiplier *= 2;
    }
    return result;
}
```

A solution without infinite streams

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    int multiplier = 1;
    List<Integer> result = new ArrayList<>();
    while (result.size() < n) {
        result = primenumbers(n * multiplier)
            .stream()
            .filter(isAwesome)
            .limit(n)
            .collect(toList());
        multiplier *= 2;
    }
    return result;
}
```

A solution without infinite streams

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    int multiplier = 1;
    List<Integer> result = new ArrayList<>();
    while (result.size() < n) {
        result = primenumbers(n * multiplier)
            .stream()
            .filter(isAwesome)
            .limit(n)
            .collect(toList());
        multiplier *= 2;
    }
    return result;
}
```

A solution without infinite streams

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    int multiplier = 1;
    List<Integer> result = new ArrayList<>();
    while (result.size() < n) {
        result = primenumbers(n * multiplier)
            .stream()
            .filter(isAwesome)
            .limit(n)
            .collect(toList());
        multiplier *= 2;
    }
    return result;
}
```

A solution without infinite streams

```
public static List<Integer> primenumbers(int n) { /* ... */ }

public static List<Integer> onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    int multiplier = 1;
    List<Integer> result = new ArrayList<>();
    while (result.size() < n) {
        result = primenumbers(n * multiplier)
            .stream()
            .filter(isAwesome)
            .limit(n)
            .collect(toList());
        multiplier *= 2;
    }
    return result;
}
```

How infinite streams can help us

```
public static IntStream primenumbers() {
    return IntStream
        .iterate(2, x -> x + 1)
        .filter(x -> isPrime(x));
}

public static IntStream onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    return primenumbers()
        .filter(isAwesome)
        .limit(n);
}
```

How infinite streams can help us

```
public static IntStream primenumbers() {
    return IntStream
        .iterate(2, x -> x + 1)
        .filter(x -> isPrime(x));
}

public static IntStream onlyAwesomePrimenumbers
    (Predicate<Integer> isAwesome, int n) {
    return primenumbers()
        .filter(isAwesome)
        .limit(n);
}
```

Overview

Null usage

```
interface User { String getMiddleName(); }

public static User findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   return findUser(id).getMiddleName();
}
```

Null usage

```
interface User { String getMiddleName(); }

public static User findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   return findUser(id).getMiddleName();
}
```

Null usage

```
interface User { String getMiddleName(); }

public static User findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   return findUser(id).getMiddleName();
}
```

Null usage

```
interface User { String getMiddleName(); }

public static User findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   return findUser(id).getMiddleName();
}
```

Null usage

```
interface User { String getMiddleName(); }

public static User findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   User user = findUser(id);
   if (user == null) {
       return null;
   }
   return user.getMiddleName();
}
```

Null usage

```
interface User { String getMiddleName(); }

public static User findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   User user = findUser(id);
   if (user == null) {
       return null;
   }
   return user.getMiddleName();
}
```

With Optional

```
interface User { Optional<String> getMiddleName(); }

public static User findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   User user = findUser(id);
   if (user == null) {
       return null;
   }
   return user.getMiddleName();
}
```

With Optional

```
interface User { Optional<String> getMiddleName(); }

public static User findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   User user = findUser(id);
   if (user == null) {
       return null;
   }
   return user.getMiddleName();
}
```

With Optional

```
interface User { Optional<String> getMiddleName(); }

public static Optional<User> findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   User user = findUser(id);
   if (user == null) {
       return null;
   }
   return user.getMiddleName();
}
```

With Optional

```
interface User { Optional<String> getMiddleName(); }

public static Optional<User> findUser(int id) { /* ... */ }

public static String getUserMiddleName(int id) {
   User user = findUser(id);
   if (user == null) {
       return null;
   }
   return user.getMiddleName();
}
```

With Optional

```
interface User { Optional<String> getMiddleName(); }

public static Optional<User> findUser(int id) { /* ... */ }

public static Optional<String> getUserMiddleName(int id) {
   User user = findUser(id);
   if (user == null) {
       return null;
   }
   return user.getMiddleName();
}
```

With Optional

```
interface User { Optional<String> getMiddleName(); }

public static Optional<User> findUser(int id) { /* ... */ }

public static Optional<String> getUserMiddleName(int id) {
   User user = findUser(id);
   if (user == null) {
       return null;
   }
   return user.getMiddleName();
}
```

With Optional

```
interface User { Optional<String> getMiddleName(); }

public static Optional<User> findUser(int id) { /* ... */ }

public static Optional<String> getUserMiddleName(int id) {
    return findUser(id)
        .flatMap(user -> user.getMiddleName());
}
```

Without Null

```java
// No!
static Optional<String>
    toLowercase(Optional<String> str) { /* ... */ }

// input != null and output != null
static String toLowercase(String str) { /* ... */ }

static Optional<String> readInput() { /* ... */ }

static Optional<String> readLowercaseInput() {
    return readInput()
        .map(in -> toLowercase(in));
}
```

Without Null

```
// No!
static Optional<String>
    toLowercase(Optional<String> str) { /* ... */ }

// input != null and output != null
static String toLowercase(String str) { /* ... */ }

static Optional<String> readInput() { /* ... */ }

static Optional<String> readLowercaseInput() {
    return readInput()
        .map(in -> toLowercase(in));
}
```

Without Null

```
// No!
static Optional<String>
    toLowercase(Optional<String> str) { /* ... */ }

// input != null and output != null
static String toLowercase(String str) { /* ... */ }

static Optional<String> readInput() { /* ... */ }

static Optional<String> readLowercaseInput() {
    return readInput()
        .map(in -> toLowercase(in));
}
```

Without Null

```
// No!
static Optional<String>
    toLowercase(Optional<String> str) { /* ... */ }

// input != null and output != null
static String toLowercase(String str) { /* ... */ }

static Optional<String> readInput() { /* ... */ }

static Optional<String> readLowercaseInput() {
    return readInput()
        .map(in -> toLowercase(in));
}
```

Interacting with legacy code

```java
/** Does something. If str == null,
    does something even more awesome. */
static void doSomething(String str) { /* ... */ }

Optional<String> maybeString;
doSomething(maybeString.orElse(null));

/** Finds something. Returns null, if nothing is found */
static String findSomething() { /* ... */ }

Optional<String> maybeString =
    Optional.ofNullable(findSomething());
```

Interacting with legacy code

```java
/** Does something. If str == null,
    does something even more awesome. */
static void doSomething(String str) { /* ... */ }

Optional<String> maybeString;
doSomething(maybeString.orElse(null));

/** Finds something. Returns null, if nothing is found */
static String findSomething() { /* ... */ }

Optional<String> maybeString =
    Optional.ofNullable(findSomething());
```

Interacting with legacy code

```java
/** Does something. If str == null,
    does something even more awesome. */
static void doSomething(String str) { /* ... */ }

Optional<String> maybeString;
doSomething(maybeString.orElse(null));

/** Finds something. Returns null, if nothing is found */
static String findSomething() { /* ... */ }

Optional<String> maybeString =
    Optional.ofNullable(findSomething());
```

Interacting with legacy code

```java
/** Does something. If str == null,
    does something even more awesome. */
static void doSomething(String str) { /* ... */ }

Optional<String> maybeString;
doSomething(maybeString.orElse(null));

/** Finds something. Returns null, if nothing is found */
static String findSomething() { /* ... */ }

Optional<String> maybeString =
    Optional.ofNullable(findSomething());
```

Overview

Checked Exceptions vs. Lambdas

```
static String doAwesomeThing(String str)
    throws IOException { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> {
            try {
                return doAwesomeThing(str);
            } catch (IOException ex) {
                // throw new RuntimeException(ex); ?
                // Ignore ?
            }
        }
    );
```

Checked Exceptions vs. Lambdas

```
static String doAwesomeThing(String str)
    throws IOException { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> {
            try {
                return doAwesomeThing(str);
            } catch (IOException ex) {
                // throw new RuntimeException(ex); ?
                // Ignore ?
            }
        }
    );
```

Checked Exceptions vs. Lambdas

```
static String doAwesomeThing(String str)
    throws IOException { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> {
            try {
                return doAwesomeThing(str);
            } catch (IOException ex) {
                // throw new RuntimeException(ex); ?
                // Ignore ?
            }
        }
    );
```

Checked Exceptions vs. Lambdas

```
static String doAwesomeThing(String str)
    throws IOException { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> {
            try {
                return doAwesomeThing(str);
            } catch (IOException ex) {
                // throw new RuntimeException(ex); ?
                // Ignore ?
            }
        }
    );
```

Checked Exceptions vs. Lambdas

```
static String doAwesomeThing(String str)
    throws IOException { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> {
            try {
                return doAwesomeThing(str);
            } catch (IOException ex) {
                // throw new RuntimeException(ex); ?
                // Ignore ?
            }
        }
    );
```

Either to the rescue

```
static Either<String, IOException>
    doAwesomeThing(String str) { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> {
            try {
                return doAwesomeThing(str);
            } catch (IOException ex) {
                // throw new RuntimeException(ex); ?
                // Ignore ?
            }
        }
    );
```

Either to the rescue

```
static Either<String, IOException>
    doAwesomeThing(String str) { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> {
            try {
                return doAwesomeThing(str);
            } catch (IOException ex) {
                // throw new RuntimeException(ex); ?
                // Ignore ?
            }
        }
    );
```

Either to the rescue

```
static Either<String, IOException>
    doAwesomeThing(String str) { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> doAwesomeThing(str))
    .filter(either -> either.isLeft());
```

Either to the rescue

```
static Either<String, IOException>
    doAwesomeThing(String str) { /* ... */ }

Arrays.asList("abc","d","ef")
    .stream()
    .map(str -> doAwesomeThing(str))
    .filter(either -> either.isLeft());
```

Overview

Functors

$$List.of(1, 2, 3).map(x \rightarrow x + 1) = List.of(2, 3, 4)$$

Functors

$$List.of(1, 2, 3).map(x \rightarrow x + 1) = List.of(2, 3, 4)$$
$$List.empty().map(x \rightarrow x + 1) = List.empty()$$

Functors

$$List.of(1,2,3).map(x \rightarrow x+1) = List.of(2,3,4)$$
$$List.empty().map(x \rightarrow x+1) = List.empty()$$
$$Optional.of(1).map(x \rightarrow x+1) = Optional.of(2)$$

Functors

$$List.of(1,2,3).map(x \rightarrow x+1) = List.of(2,3,4)$$
$$List.empty().map(x \rightarrow x+1) = List.empty()$$
$$Optional.of(1).map(x \rightarrow x+1) = Optional.of(2)$$
$$Optional.empty().map(x \rightarrow x+1) = Optional.empty()$$

Functors

$$List.of(1,2,3).map(x \rightarrow x+1) = List.of(2,3,4)$$
$$List.empty().map(x \rightarrow x+1) = List.empty()$$
$$Optional.of(1).map(x \rightarrow x+1) = Optional.of(2)$$
$$Optional.empty().map(x \rightarrow x+1) = Optional.empty()$$
$$Future.of(1).map(x \rightarrow x+1) = Future.of(2)$$

Functor properties

$$List.of(1, 2, 3).map(x \rightarrow x) = List.of(1, 2, 3)$$

Functor properties

$$List.of(1,2,3).map(x \rightarrow x) = List.of(1,2,3)$$
$$List.of(1,2,3).map(x \rightarrow x) \neq List.empty()$$

Functor properties

$$List.of(1,2,3).map(x \to x) = List.of(1,2,3)$$

$$List.of(1,2,3).map(x \to x) \neq List.empty()$$

$$List.of(1,2,3).map(x \to x+1).map(x \to x*2)$$

$$= List.of(1,2,3).map(x \to (x+1)*2)$$

Definition of a Monad

$$map : (A \to B, \text{Monad}\langle A \rangle) \to \text{Monad}\langle B \rangle$$

Definition of a Monad

$$map : (A \to B, \text{Monad}\langle A \rangle) \to \text{Monad}\langle B \rangle$$
$$of : A \to \text{Monad}\langle A \rangle$$

Definition of a Monad

$$map : (A \to B, \text{Monad}\langle A \rangle) \to \text{Monad}\langle B \rangle$$
$$of : A \to \text{Monad}\langle A \rangle$$
$$flatten : \text{Monad}\langle \text{Monad}\langle A \rangle \rangle \to \text{Monad}\langle A \rangle$$

Definition of a Monad

$$map : (A \to B, \text{Monad}\langle A \rangle) \to \text{Monad}\langle B \rangle$$
$$of : A \to \text{Monad}\langle A \rangle$$
$$flatten : \text{Monad}\langle \text{Monad}\langle A \rangle \rangle \to \text{Monad}\langle A \rangle$$
$$(flatMap(f, monad) = flatten(map(f, list)))$$

Of Examples

$$List.of(1)$$

Of Examples

$$List.of(1)$$

$$Optional.of(1)$$

Of Examples

$$List.of(1)$$

$$Optional.of(1)$$

$$Future.of(1)$$

Flatten Examples

$$List.of(List.of(1,2), List.empty(), List.of(3)).flatten() = List.of(1,2,3)$$

Flatten Examples

$$List.of(List.of(1,2), List.empty(), List.of(3)).flatten() = List.of(1,2,3)$$

$$Optional.of(Optional.of(2)).flatten() = Optional.of(2)$$

Flatten Examples

$$List.of(List.of(1,2), List.empty(), List.of(3)).flatten() = List.of(1,2,3)$$

$$Optional.of(Optional.of(2)).flatten() = Optional.of(2)$$

$$Optional.of(Optional.empty()).flatten() = Optional.empty()$$

Flatten Examples

$$List.of(List.of(1,2), List.empty(), List.of(3)).flatten() = List.of(1,2,3)$$

$$Optional.of(Optional.of(2)).flatten() = Optional.of(2)$$

$$Optional.of(Optional.empty()).flatten() = Optional.empty()$$

$$Optional.empty().flatten() = Optional.empty()$$

Flatten Examples

$$List.of(List.of(1,2), List.empty(), List.of(3)).flatten() = List.of(1,2,3)$$

$$Optional.of(Optional.of(2)).flatten() = Optional.of(2)$$

$$Optional.of(Optional.empty()).flatten() = Optional.empty()$$

$$Optional.empty().flatten() = Optional.empty()$$

$$Future.of(Future.of(1)).flatten() = Future.of(1)$$

Monad properties

$$flatten(map(f, of(value))) = f(value)$$

Monad properties

$$flatten(map(f, of(value))) = f(value)$$

$$flatten(map(of, monad)) = monad$$

Monad properties

$$flatten(map(f, of(value))) = f(value)$$

$$flatten(map(of, monad)) = monad$$

$$flatten(map(g, flatten(map(f, monad))))$$

$$= flatten(map(x \rightarrow flatten(map(g, f(x))), monad))$$

Monad Examples

- Optional: One or no element

Monad Examples

- Optional: One or no element
- List/Stream: Arbitrary number of ordered elements

Monad Examples

- Optional: One or no element
- List/Stream: Arbitrary number of ordered elements
- Set: Arbitrary number of distinct elements

Monad Examples

- Optional: One or no element
- List/Stream: Arbitrary number of ordered elements
- Set: Arbitrary number of distinct elements
- Either: An element or an error

Monad Examples

- Optional: One or no element
- List/Stream: Arbitrary number of ordered elements
- Set: Arbitrary number of distinct elements
- Either: An element or an error
- Future: An element which might not be there yet

Monad Examples

- Optional: One or no element
- List/Stream: Arbitrary number of ordered elements
- Set: Arbitrary number of distinct elements
- Either: An element or an error
- Future: An element which might not be there yet
- IO: A side effect

Monad Examples

- Optional: One or no element
- List/Stream: Arbitrary number of ordered elements
- Set: Arbitrary number of distinct elements
- Either: An element or an error
- Future: An element which might not be there yet
- IO: A side effect
- State: A value that might be modified

Overview

Mutable bank account

```
public class BankAccount {

    private int euros = 0;

    public void addSavings(int euros) {
        this.euros += euros;
    }

    public void withdraw(int euros) {
        this.euros -= euros;
    }

}
```

Mutable bank account

```
public class BankAccount {

    private int euros = 0;

    public void addSavings(int euros) {
        this.euros += euros;
    }

    public void withdraw(int euros) {
        this.euros -= euros;
    }

}
```

Mutable bank account

```
public class BankAccount {

    private int euros = 0;

    public void addSavings(int euros) {
        this.euros += euros;
    }

    public void withdraw(int euros) {
        this.euros -= euros;
    }

}
```

Mutable bank account

```
public class BankAccount {

    private int euros = 0;

    public void addSavings(int euros) {
        this.euros += euros;
    }

    public void withdraw(int euros) {
        this.euros -= euros;
    }

}
```

The problem with mutability

```
static void keepCareOfMyAccount
    (BankAccount account) { /* ... */ }

BankAccount account = new BankAccount();
account.addSavings(100);
keepCareOfMyAccount(account);
// Account might be empty!
// (or magically filled)
```

The problem with mutability

```
static void keepCareOfMyAccount
    (BankAccount account) { /* ... */ }

BankAccount account = new BankAccount();
account.addSavings(100);
keepCareOfMyAccount(account);
// Account might be empty!
// (or magically filled)
```

The problem with mutability

```
static void keepCareOfMyAccount
    (BankAccount account) { /* ... */ }

BankAccount account = new BankAccount();
account.addSavings(100);
keepCareOfMyAccount(account);
// Account might be empty!
// (or magically filled)
```

The problem with mutability

```
static void keepCareOfMyAccount
    (BankAccount account) { /* ... */ }

BankAccount account = new BankAccount();
account.addSavings(100);
keepCareOfMyAccount(account);
// Account might be empty!
// (or magically filled)
```

Immutable bank account

```
public class BankAccount {

    private final int euros;

    public BankAccount(int euros) {
        this.euros = euros;
    }

    public BankAccount addSavings(int euros) {
        return new BankAccount(this.euros + euros);
    }

    public BankAccount withdraw(int euros) {
        return new BankAccount(this.euros + euros);
    }

}
```

Immutable bank account

```
public class BankAccount {

    private final int euros;

    public BankAccount(int euros) {
        this.euros = euros;
    }

    public BankAccount addSavings(int euros) {
        return new BankAccount(this.euros + euros);
    }

    public BankAccount withdraw(int euros) {
        return new BankAccount(this.euros + euros);
    }

}
```

Immutable bank account

```
public class BankAccount {

    private final int euros;

    public BankAccount(int euros) {
        this.euros = euros;
    }

    public BankAccount addSavings(int euros) {
        return new BankAccount(this.euros + euros);
    }

    public BankAccount withdraw(int euros) {
        return new BankAccount(this.euros + euros);
    }

}
```

Immutable bank account

```
public class BankAccount {

    private final int euros;

    public BankAccount(int euros) {
        this.euros = euros;
    }

    public BankAccount addSavings(int euros) {
        return new BankAccount(this.euros + euros);
    }

    public BankAccount withdraw(int euros) {
        return new BankAccount(this.euros + euros);
    }

}
```

Immutability to the rescue

```
static void keepCareOfMyAccount
    (BankAccount account) { /* ... */ }

BankAccount account = new BankAccount(0).addSavings(100);
keepCareOfMyAccount(account);
// Account will still have 100 euros
```

Immutability to the rescue

```
static void keepCareOfMyAccount
    (BankAccount account) { /* ... */ }

BankAccount account = new BankAccount(0).addSavings(100);
keepCareOfMyAccount(account);
// Account will still have 100 euros
```

Immutability to the rescue

```
static void keepCareOfMyAccount
    (BankAccount account) { /* ... */ }

BankAccount account = new BankAccount(0).addSavings(100);
keepCareOfMyAccount(account);
// Account will still have 100 euros
```

Immutability to the rescue

```
static void keepCareOfMyAccount
    (BankAccount account) { /* ... */ }

BankAccount account = new BankAccount(0).addSavings(100);
keepCareOfMyAccount(account);
// Account will still have 100 euros
```

Conclusion

- Pure functions make testing easier

Conclusion

- Pure functions make testing easier
- Pure functions are the way to go with the Stream API

Conclusion

- Pure functions make testing easier
- Pure functions are the way to go with the Stream API
- The Stream API makes it possible to think in terms of a pipeline

Conclusion

- Pure functions make testing easier
- Pure functions are the way to go with the Stream API
- The Stream API makes it possible to think in terms of a pipeline
- Infinite streams can lead to cleaner solutions

Conclusion

- Pure functions make testing easier
- Pure functions are the way to go with the Stream API
- The Stream API makes it possible to think in terms of a pipeline
- Infinite streams can lead to cleaner solutions
- Optional erases the source of NullPointerExceptions

Conclusion

- Pure functions make testing easier
- Pure functions are the way to go with the Stream API
- The Stream API makes it possible to think in terms of a pipeline
- Infinite streams can lead to cleaner solutions
- Optional erases the source of NullPointerExceptions
- Either let you handle exceptions on type level

Conclusion

- Pure functions make testing easier
- Pure functions are the way to go with the Stream API
- The Stream API makes it possible to think in terms of a pipeline
- Infinite streams can lead to cleaner solutions
- Optional erases the source of NullPointerExceptions
- Either let you handle exceptions on type level
- Monads combine many concerns into a general structure

Conclusion

- Pure functions make testing easier
- Pure functions are the way to go with the Stream API
- The Stream API makes it possible to think in terms of a pipeline
- Infinite streams can lead to cleaner solutions
- Optional erases the source of NullPointerExceptions
- Either let you handle exceptions on type level
- Monads combine many concerns into a general structure
- Monad laws help you refactoring

Conclusion

- Pure functions make testing easier
- Pure functions are the way to go with the Stream API
- The Stream API makes it possible to think in terms of a pipeline
- Infinite streams can lead to cleaner solutions
- Optional erases the source of NullPointerExceptions
- Either let you handle exceptions on type level
- Monads combine many concerns into a general structure
- Monad laws help you refactoring
- Immutability makes it easier to reason about code