Property Based Testing

Online-Session:

- Please switch off your camera
- Please mute your microphone
  … unless it's *time for questions*
- Use chat for all other stuff

Clone if you want to try out the examples:

`https://github.com/jlink/pbt-workshop`

Property Based Testing

in Java
with jqwik

@johanneslink

johanneslink.net

Example-based Tests

An *example* shows that the code delivers
a specific result
for a specific set of inputs

Example-based Tests

An *example* shows that the code delivers
a specific result
for a specific set of inputs

```
@Test
void reverseList() {
   List<Integer> aList = Arrays.asList(1, 2, 3);
   Collections.reverse(aList);
   assertThat(aList).containsExactly(3, 2, 1);
}
```

Example-based Tests

An *example* shows that the code delivers
a specific result
for a specific set of inputs

```
@Example
void reverseList() {
   List<Integer> aList = Arrays.asList(1, 2, 3);
   Collections.reverse(aList);
   assertThat(aList).containsExactly(3, 2, 1);
}
```

Does *reverse()* only work for the tested examples?

How representative are our examples?

How many examples does it take to create enough trust?

```
@Example void emptyList() {
    List<Integer> aList = Collections.emptyList();
    assertThat(Collections.reverse(aList)).isEmpty();
}

@Example void oneElement() {
    List<Integer> aList = Collections.singletonList(1);
    assertThat(Collections.reverse(aList)).containsExactly(1);
}

@Example void manyElements() {
    List<Integer> aList = asList(1, 2, 3, 4, 5, 6);
    assertThat(Collections.reverse(aList)).containsExactly(6, 5, 4, 3, 2, 1);
}

@Example void duplicateElements() {
    List<Integer> aList = asList(1, 2, 2, 4, 6, 6);
    assertThat(Collections.reverse(aList)).containsExactly(6, 6, 4, 2, 2, 1);
}
```

Properties

A *Property* claims that
for a class of inputs (*preconditions*)
certain generic qualities (*postconditions, invariants*) hold

```
Collections.reverse(List aList):
   // preconditions?
   // postconditions and invariants?
```

```
Collections.reverse(List aList):
    // preconditions?
    // postconditions and invariants?
```

```
Collections.reverse(List aList):
    // preconditions?
    // postconditions and invariants?
```

Preconditions

- Any non-null list

```
Collections.reverse(List aList):
   // preconditions?
   // postconditions and invariants?
```

Preconditions

- Any non-null list

Invariants

- Size of list remains the same
- All elements stay in list
- After reversing the first element becomes the last
- Applying reverse twice produces the original list

A Property in Java Code

```
boolean theSizeRemainsTheSame(List<Integer> original) {
    List<Integer> reversed = reverse(original);
    return original.size() == reversed.size();
}


private <T> List<T> reverse(List<T> original) {
    List<T> clone = new ArrayList<>(original);
    Collections.reverse(clone);
    return clone;
}
```

Jqwik

```
@Property
boolean theSizeRemainsTheSame(@ForAll List<Integer> original) {
   List<Integer> reversed = reverse(original);
   return original.size() == reversed.size();
}
```

```java
@Property
boolean theSizeRemainsTheSame(@ForAll List<Integer> original) {
   List<Integer> reversed = reverse(original);
   return original.size() == reversed.size();
}
```

```java
@Property
boolean theSizeRemainsTheSame(@ForAll List<Integer> original) {
    List<Integer> reversed = reverse(original);
    return original.size() == reversed.size();
}
```

```java
@Property
void allElementsStay(@ForAll List<Integer> original) {
    List<Integer> reversed = reverse(original);
    Assertions.assertThat(original).allMatch(
        element -> reversed.contains(element)
    );
}
```

Demo

- Reverse List
- Length of String
- Absolute value of Integer
- Sum of two integers
- Integration in Gradle und IntelliJ

What jqwik is…

https://jqwik.net

- Test engine for the JUnit 5 platform
- Generator for test cases creating
  - random and typical input data
  - sometimes even an exhaustive set of all possible input combinations
- Current version: 1.2.6

What jqwik is not…

- It's not a fully randomized testing tool, which can be applied on your software without thinking
- Properties cannot be proven, they can only be falsified

```java
@Property
void squareOfRootIsOriginalValue(@ForAll double aNumber) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```java
@Property
void squareOfRootIsOriginalValue(@ForAll double aNumber) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
java.lang.AssertionError:
Expecting:
  <NaN>
to be close to:
  <-1.0>
by less than 1% but difference was NaN%.
(a difference of exactly 1% being considered valid)
```

Constraining Value Generation

Often a Property is only valid for a constrained subset of a given type

```java
@Property
void squareOfRootIsOriginalValue(
    @ForAll @DoubleRange(min=0) double aNumber
) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```

```java
@Property
void squareOfRootIsOriginalValue(
    @ForAll @Positive double aNumber
) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```

```java
@Property
void squareOfRootIsOriginalValue(
    @ForAll("positiveDoubles") double aNumber
) {
    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}

@Provide
Arbitrary<Double> positiveDoubles() {
    return Arbitraries.doubles().between(0, Double.MAX_VALUE);
}
```

```
tries = 1000,
checks = 1000,
seed = 7890962728489990406
```

Arbitrary:
"Monad-like" factory for Generators
of Arbitrary Values

```
public interface Arbitrary<T> {
    RandomGenerator<T> generator(int genSize);

    default Arbitrary<T> filter(final Predicate<T> filterPredicate) { … }
    default <U> Arbitrary<U> map(final Function<T, U> mapper) { … }
    …
}
```

Arbitrary:
"Monad-like" factory for Generators
of Arbitrary Values

```
public interface Arbitrary<T> {
    RandomGenerator<T> generator(int genSize);

    default Arbitrary<T> filter(final Predicate<T> filterPredicate) { … }
    default <U> Arbitrary<U> map(final Function<T, U> mapper) { … }
    …
}
```

```
public interface RandomGenerator<T> {
    Shrinkable<T> next(Random random);
}
```

```java
@Property
void squareOfRootIsOriginalValue(@ForAll double aNumber) {
    Assume.that(aNumber > 0);

    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```java
@Property
void squareOfRootIsOriginalValue(@ForAll double aNumber) {
    Assume.that(aNumber > 0);

    double sqrt = Math.sqrt(aNumber);
    Assertions.assertThat(sqrt * sqrt).isCloseTo(aNumber, withPercentage(1));
}
```

```
tries = 1000,
checks = 489,
seed = -1808546598028468149
```

```
static <E> List<E> brokenReverse(List<E> aList) {
    
}

@Property(shrinking = ShrinkingMode.OFF)
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
static <E> List<E> brokenReverse(List<E> aList) {
    
}

@Property(shrinking = ShrinkingMode.OFF)
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
Property [reverseShouldSwapFirstAndLast] falsified with sample
  [[0, -2147483648, 2147483647, -7997, 7997, -3223, -6474, 1915, -7151,
  3102, 4362, 714, 3053, 1919, -445, 7498, -2424, 3016, -5127, -7401, -7946,
  -3801, -305]]
```

```
static <E> List<E> brokenReverse(List<E> aList) {
    
}

@Property(shrinking = ShrinkingMode.OFF)
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
static <E> List<E> brokenReverse(List<E> aList) {
    
}

@Property(shrinking = ShrinkingMode.OFF)
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
Property [reverseShouldSwapFirstAndLast] falsified with sample
   [[0, 0, 0, -1]]
```

```java
static <E> List<E> brokenReverse(List<E> aList) {
    if (aList.size() < 4) {
        aList = new ArrayList<>(aList);
        reverse(aList);
    }
    return aList;
}

@Property(shrinking = ShrinkingMode.OFF)
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
Property [reverseShouldSwapFirstAndLast] falsified with sample
   [[0, 0, 0, -1]]
```

```java
static <E> List<E> brokenReverse(List<E> aList) {
    if (aList.size() < 4) {
        aList = new ArrayList<>(aList);
        reverse(aList);
    }
    return aList;
}

@Property
boolean reverseShouldSwapFirstAndLast(@ForAll List<Integer> aList) {
    Assume.that(!aList.isEmpty());
    List<Integer> reversed = brokenReverse(aList);
    return aList.get(0) == reversed.get(aList.size() - 1);
}
```

```
org.opentest4j.AssertionFailedError:
Property [reverseShouldSwapFirstAndLast] falsified with sample
   [[0, 0, 0, -1]]
```

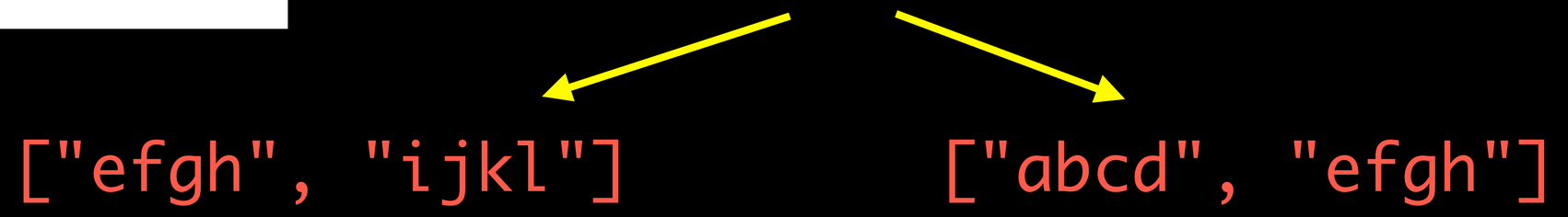
The Importance of Being Shrunk

- Shrinking of falsified property: Trying to find the simplest set of inputs to make the property fail
- Sometimes there is no "simplest" failing example or finding it would take very long
- Use heuristics to shrink values
  - try integer closer to 0
  - try collection with fewer elements
- Requires full determinism of property method

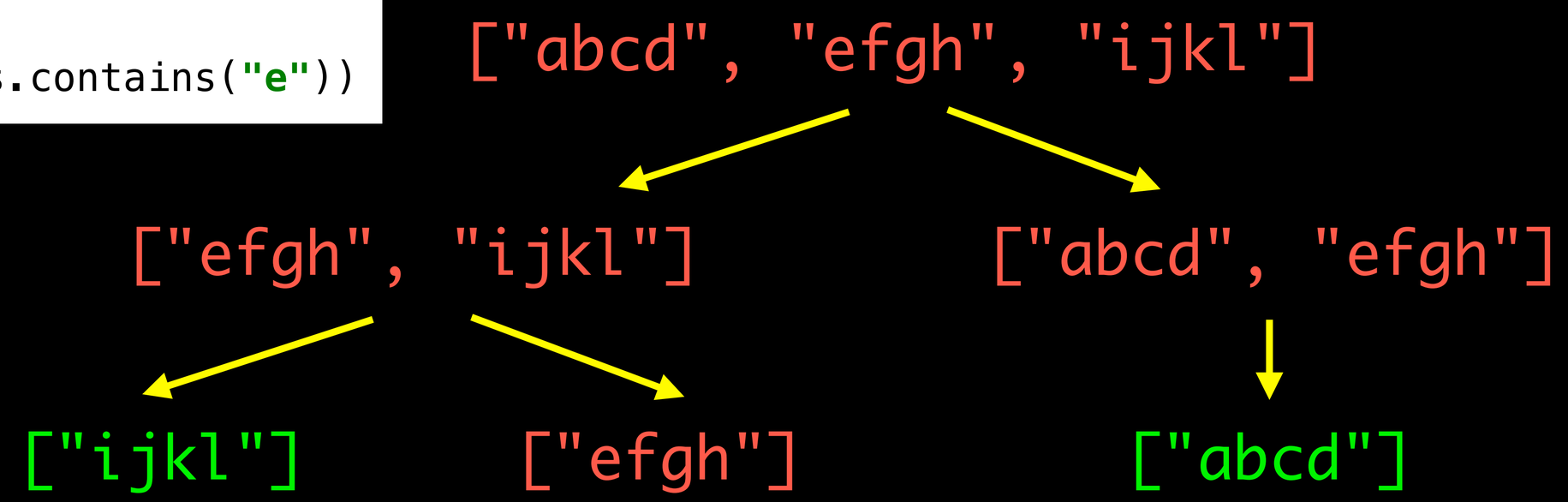
```
list.stream()
    .noneMatch(s -> s.contains("e"))
```

["abcd", "efgh", "ijkl"]

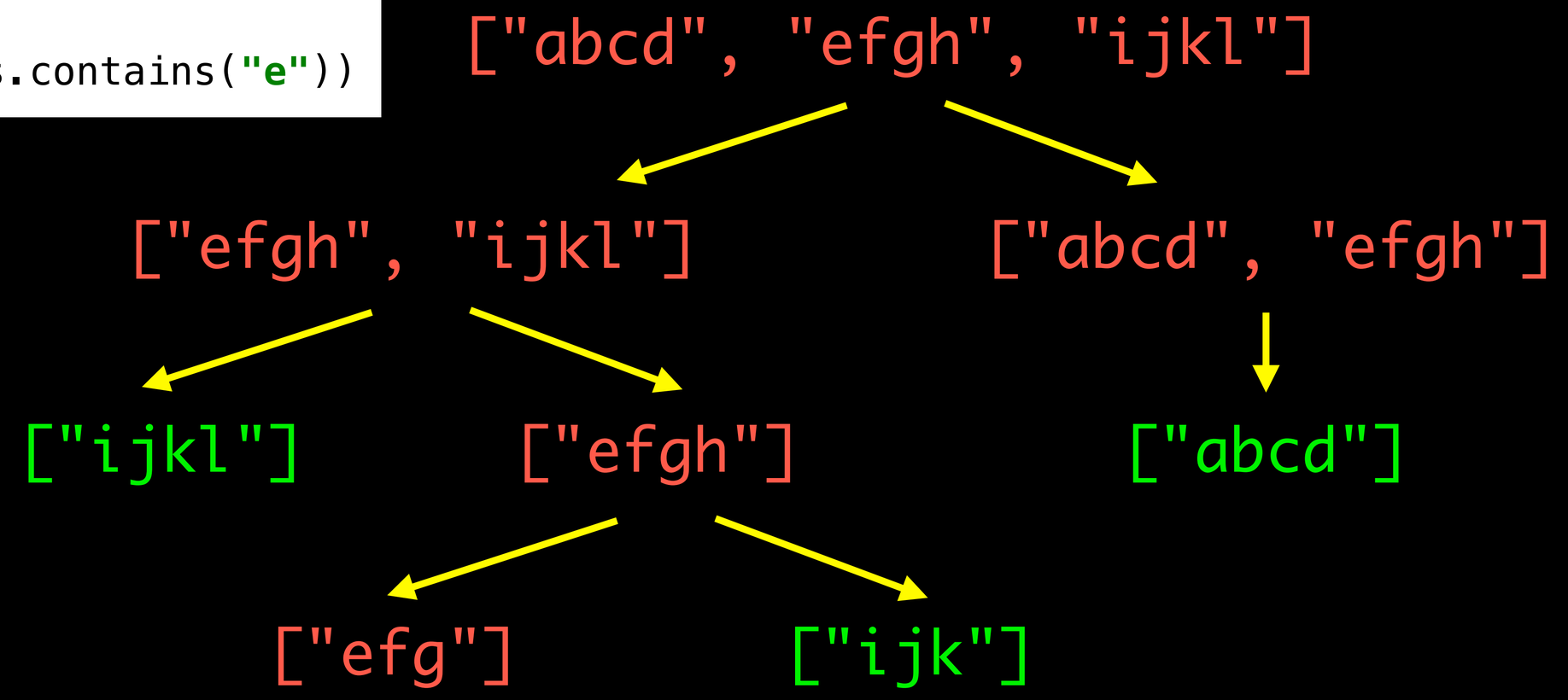
```
list.stream()
    .noneMatch(s -> s.contains("e"))
```



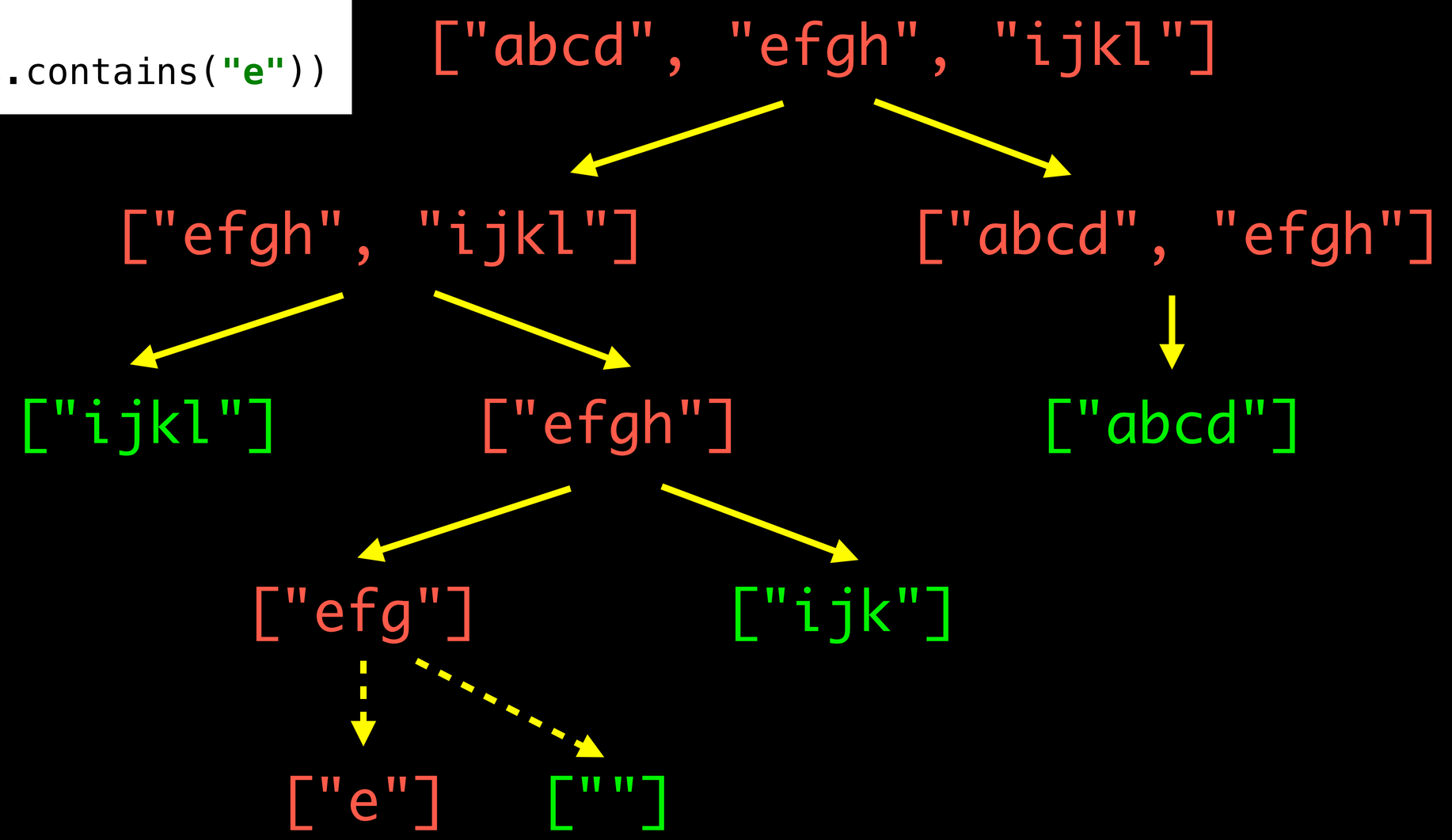
```
list.stream()
    .noneMatch(s -> s.contains("e"))
```



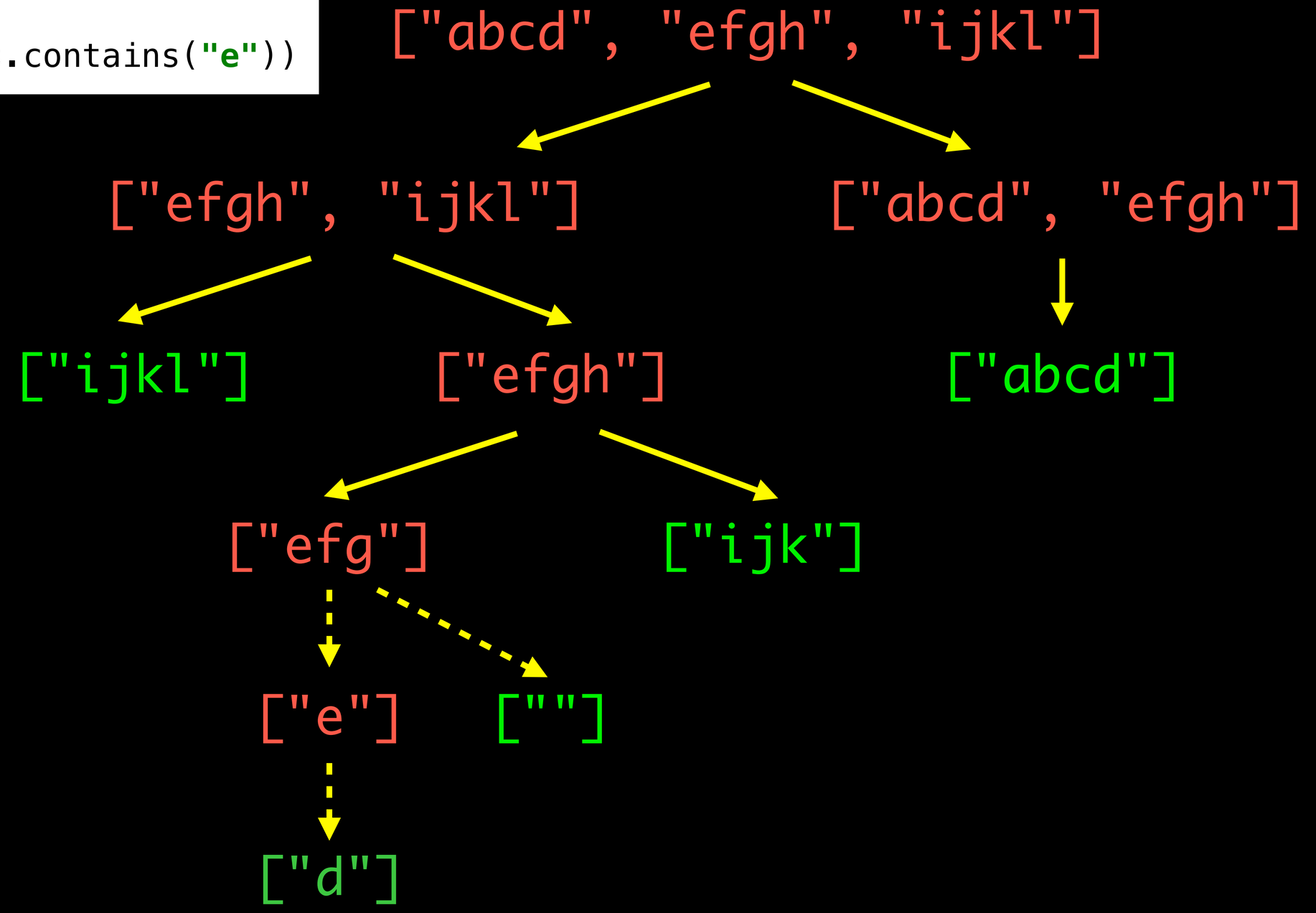
```
list.stream()
    .noneMatch(s -> s.contains("e"))
```



```
list.stream()
    .noneMatch(s -> s.contains("e"))
```



```
list.stream()
    .noneMatch(s -> s.contains("e"))
```



Type-based vs Integrated Shrinking

- Type-Based Shrinking: Only type is used as constraint for shrinking attempts
  - Problem: Shrinking can create results that would have been excluded during generation
- Integrated Shrinking: All steps and conditions of generation are also considered during shrinking
- jqwik implements integrated shrinking

How to Generate Values

Fluent Interfaces

How to Generate Values

Fluent Interfaces

`Arbitraries` are the beginning of everything...

How to Generate Values

Fluent Interfaces

Arbitraries are the beginning of everything…

```
Arbitraries
  .strings()
  .integers()
  .floats()
  …
  .of(value1, value2, …) // choosing among values
  .oneOf(…)              // choosing among arbitrariness
  …
```

Configuring Generators

Fluent Interfaces

Configuring Generators

Fluent Interfaces

`Arbitraries` can be configured

Configuring Generators

Fluent Interfaces

Arbitraries can be configured

```
@Provide
StringArbitrary fluentString() {
   return Arbitraries.strings()
                     .alpha()
                     .numeric()
                     .withChars('?', '!', '.')
                     .ofMinLength(2)
                     .ofMaxLength(10);
}
```

Changing Generated Values

- Sometimes you want to filter generate values yourself
- Sometimes you want to map generated values to others
- Sometimes you want to combine generated values with each other

Filtering

```
@Property
boolean evenNumbersAreEven(@ForAll("evenUpTo10000") int anInt) {
    return anInt % 2 == 0;
}

@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 10000)
                .filter(i -> i % 2 == 0);
}
```

Mapping

```
@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 5000)
                .map(i -> i * 2);
}
```

Mapping

```
@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 5000)
                .map(i -> i * 2);
}
```

```
@Provide
Arbitrary<Integer> evenUpTo10000() {
    return Arbitraries.integers()
                .between(0, 10000)
                .filter(i -> i % 2 == 0);
}
```

Combining

```
public class Person {
    public Person(String firstName, String lastName) {…}
    public String fullName() {return firstName + " " + lastName;}
}
```

Combining

```
public class Person {
    public Person(String firstName, String lastName) {…}
    public String fullName() {return firstName + " " + lastName;}
}
```

```
@Provide
Arbitrary<Person> validPerson() {
    Arbitrary<Character> initialChar = Arbitraries.chars().between('A', 'Z');
    Arbitrary<String> firstName = Arbitraries.strings()… ;
    Arbitrary<String> lastName = Arbitraries.strings()… ;
    return Combinators.combine(initialChar, firstName, lastName)
            .as((initial, first, last) -> new Person(initial + first, last));
}
```

Exhaustive Value Generation

```
@Property(generation = GenerationMode.EXHAUSTIVE)
void allChessSquares(
      @ForAll @CharRange(from = 'a', to = 'h') char column,
      @ForAll @CharRange(from = '1', to = ,8') char row
) {
   String square = column + "" + row;
   System.out.println(square);
}
```

How to Specify it

Patterns and Strategies of PBT

- Fuzzying
- Postconditions
- Metamorphic Properties
- Black-box Testing
- Inductive Testing
- Test Oracle
- Stateful Testing
- Model-based Properties

How to Specify it

Patterns and Strategies of PBT

- Fuzzying
- Postconditions
- Metamorphic Properties
- Black-box Testing
- Inductive Testing
- Test Oracle
- Stateful Testing
- Model-based Properties

How to Specify it

Patterns and Strategies of PBT

- Fuzzying
- Postconditions
- Metamorphic Properties
- Black-box Testing
- Inductive Testing
- Test Oracle
- Stateful Testing
- Model-based Properties

https://johanneslink.net/how-to-specify-it/

Postconditions

- We can often enumerate postconditions for an operation
- Example: Inserting in Binary Search Tree

  "After insertion of key X with value Y into an arbitrary binary tree,
  we should find value Y when searching for X"

```
@Property
boolean inserted_value_can_be_found(
        @ForAll Integer key, @ForAll Integer value,
        @ForAll BinarySearchTree<Integer, Integer> bst
) {
    Optional<Integer> found = bst.insert(key, value).find(key);
    return found.equals(Optional.of(value));
}
```

Metamorphic Properties

"... even if the expected result of a function call [...] may be difficult to predict, we may still be able to express an expected relationship between this result, and the result of a related call."

John Hughes in *How to Specify It*

https://johanneslink.net/how-to-specify-it

Metamorphic Property: Inject Data

- If I know f(x), then I can derive f(x + a)
- Example: Reversing two concatenated lists

  reverse(list1) = list1', reverse(list2) = list2'
  => reverse(list1 + list2) = list1' + list2'

```
@Property
boolean reverseConcatenatedLists(
        @ForAll List<Integer> first,
        @ForAll List<Integer> second
) {
    List<Integer> firstReversed = reverse(first);
    List<Integer> secondReversed = reverse(second);

    List<Integer> reversed = reverse(concat(first, second));

    return reversed.equals(concat(secondReversed, firstReversed));
}
```

Metamorphic Property: Inverse Functions

- function + inverse function produces the original input
  - Encode / Decode

```java
class InverseFunctions {

    @Property
    void encodeAndDecodeAreInverse(
            @ForAll @StringLength(min = 1, max = 20) String toEncode,
            @ForAll("charset") String charset
    ) throws UnsupportedEncodingException {
        String encoded = URLEncoder.encode(toEncode, charset);
        assertThat(URLDecoder.decode(encoded, charset)).isEqualTo(toEncode);
    }

    @Provide
    Arbitrary<String> charset() {
        Set<String> charsets = Charset.availableCharsets().keySet();
        return Arbitraries.of(charsets.toArray(new String[charsets.size()]));
    }
}
```

```java
class InverseFunctions {

    @Property
    void encodeAndDecodeAreInverse(
            @ForAll @StringLength(min = 1, max = 20) String toEncode,
            @ForAll("charset") String charset
    ) throws UnsupportedEncodingException {
        String encoded = URLEncoder.encode(toEncode, charset);
        assertThat(URLDecoder.decode(encoded, charset)).isEqualTo(toEncode);
    }

    @Provide
    Arbitrary<String> charset() {
        Set<String> charsets = Charset.availableCharsets().keySet();
        return Arbitraries.of(charsets.toArray(new String[charsets.size()]));
    }
}
```

```
sample = ["€", "Big5"]
original-sample = ["鯒阧퉅ꭑ̡찝蚨ڬ⁑냅ꦀ୨۴斃", "x-IBM1098"]

  org.opentest4j.AssertionFailedError:
  Expecting:
   <"?">
  to be equal to:
   <"€">
  but was not.
```

Inductive Testing: Solving a smaller problem first

- Sometimes we can derive a full specification from a set of smaller properties
- Example: A list is sorted, if
  - The first element is smaller than or equal to the second
  - Everything after the first element is also sorted

```java
@Property
boolean sortingAListWorks(@ForAll List<Integer> unsorted) {
   return isSorted(sort(unsorted));
}

private boolean isSorted(List<Integer> sorted) {
   if (sorted.size() <= 1) return true;
   return sorted.get(0) <= sorted.get(1)
           && isSorted(sorted.subList(1, sorted.size()));
}
```

Lessons Learned

- Example-based tests…
  - are often a good starting point
  - can sometimes be translated into a property
- Properties can be weaker than the functional specification and still provide value
- Invest in building domain-specific generators
- Check your generators for expected special cases and reasonable value distribution

Alternative PBT Tools for Java

- JUnit-Quickcheck:
  Tight integration with JUnit 4
- QuickTheories:
  Can work with any test framework
- Vavr: Functional Java library
  with PBT module of its own
- jetCheck: Built by JetBrains to test their IDEs

jqwik on Github:

https://github.com/jlink/jqwik

I'm looking for contributors!

Code:

https://github.com/jlink/pbt-workshop

https://github.com/jlink/jqwik-samples

Slides:

https://johanneslink.net/downloads/PropertyTesting-Online.pdf

Blog:

https://blog.johanneslink.net/2018/03/24/property-based-testing-in-java-introduction/