An introduction to property-based testing

Vincent Pradeilles - Worldline

Classic unit tests

“Given, When, Then”

Classic unit tests

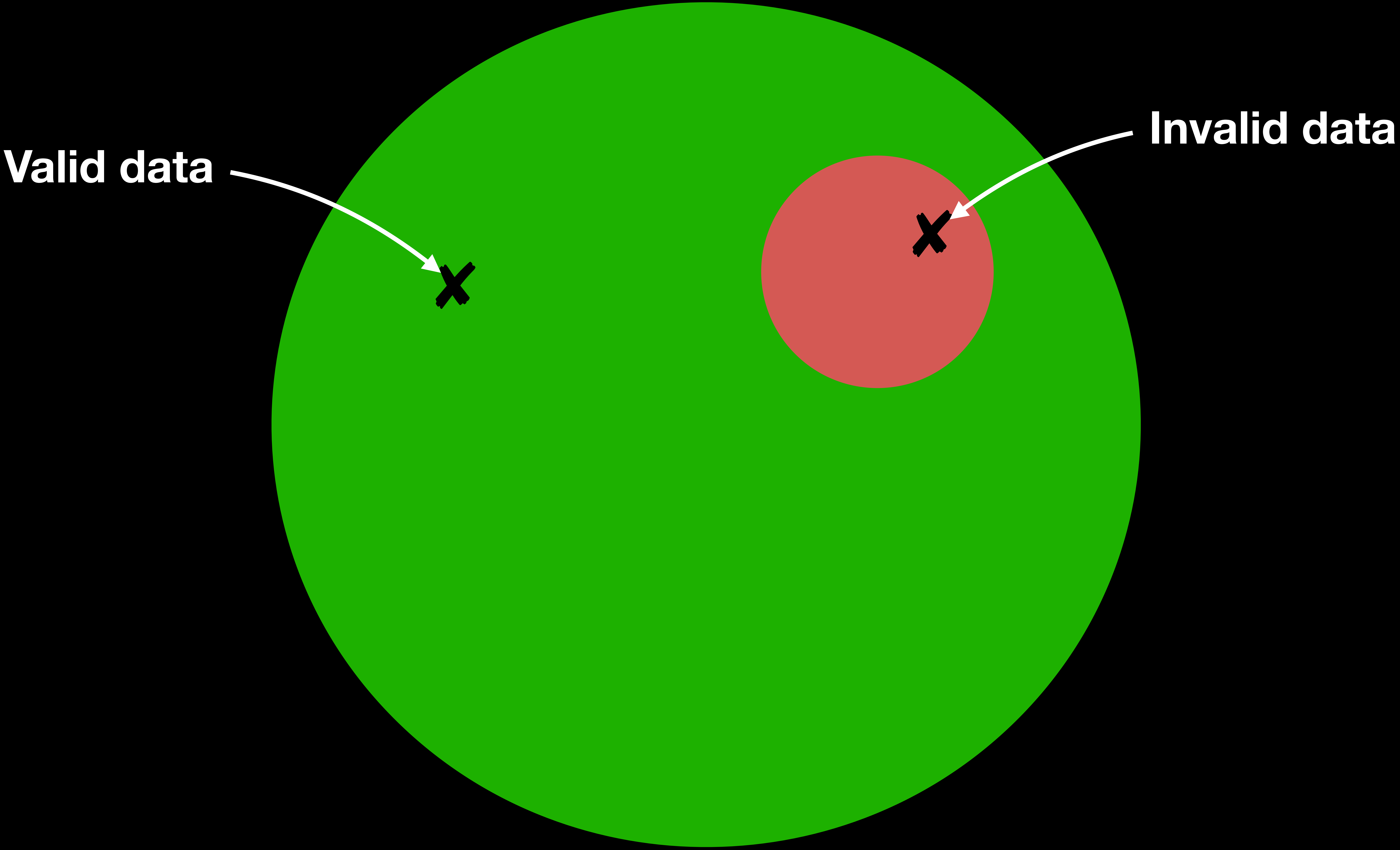
Given some state
When I perform this action
Then I expect that result

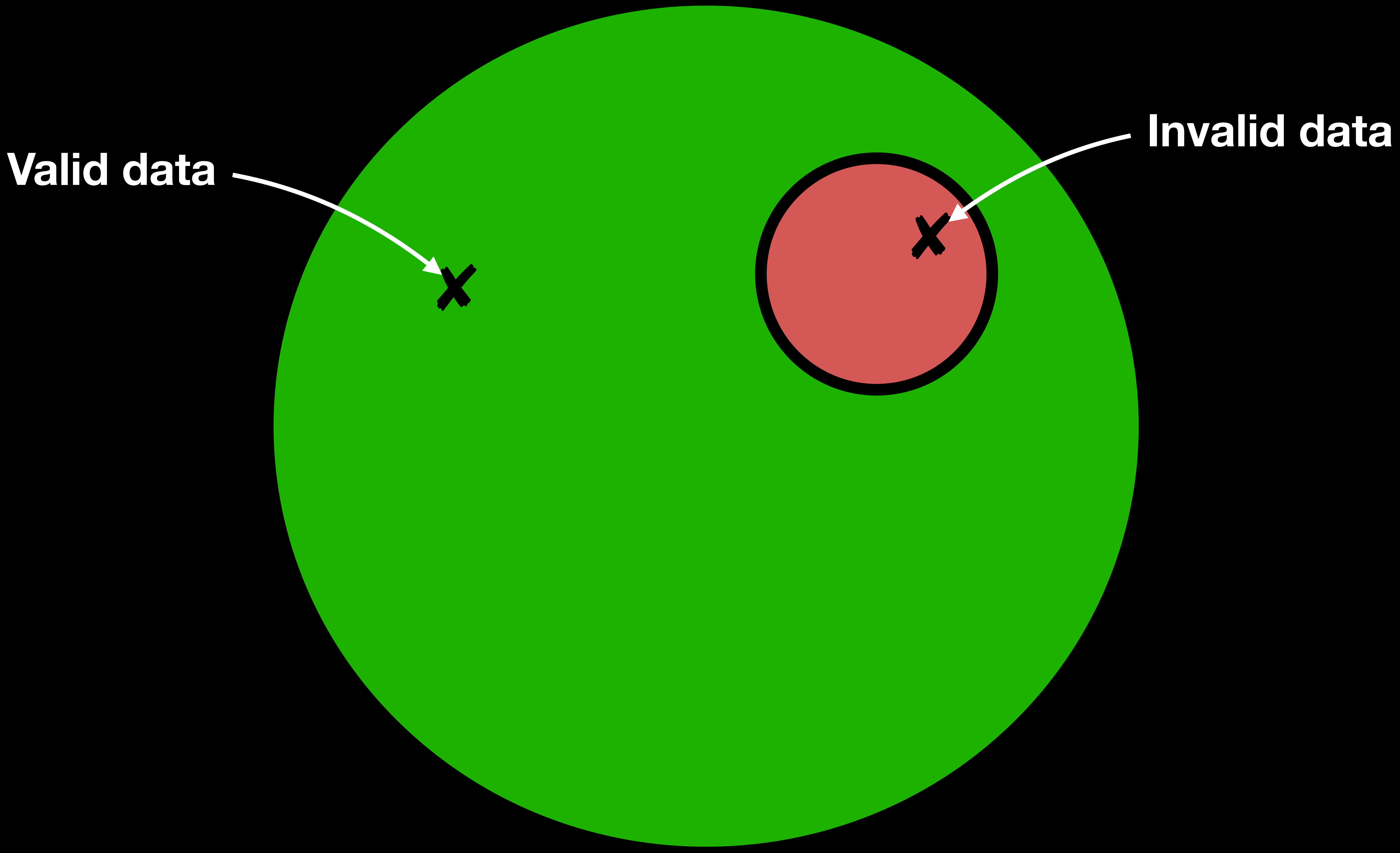
Classic unit tests

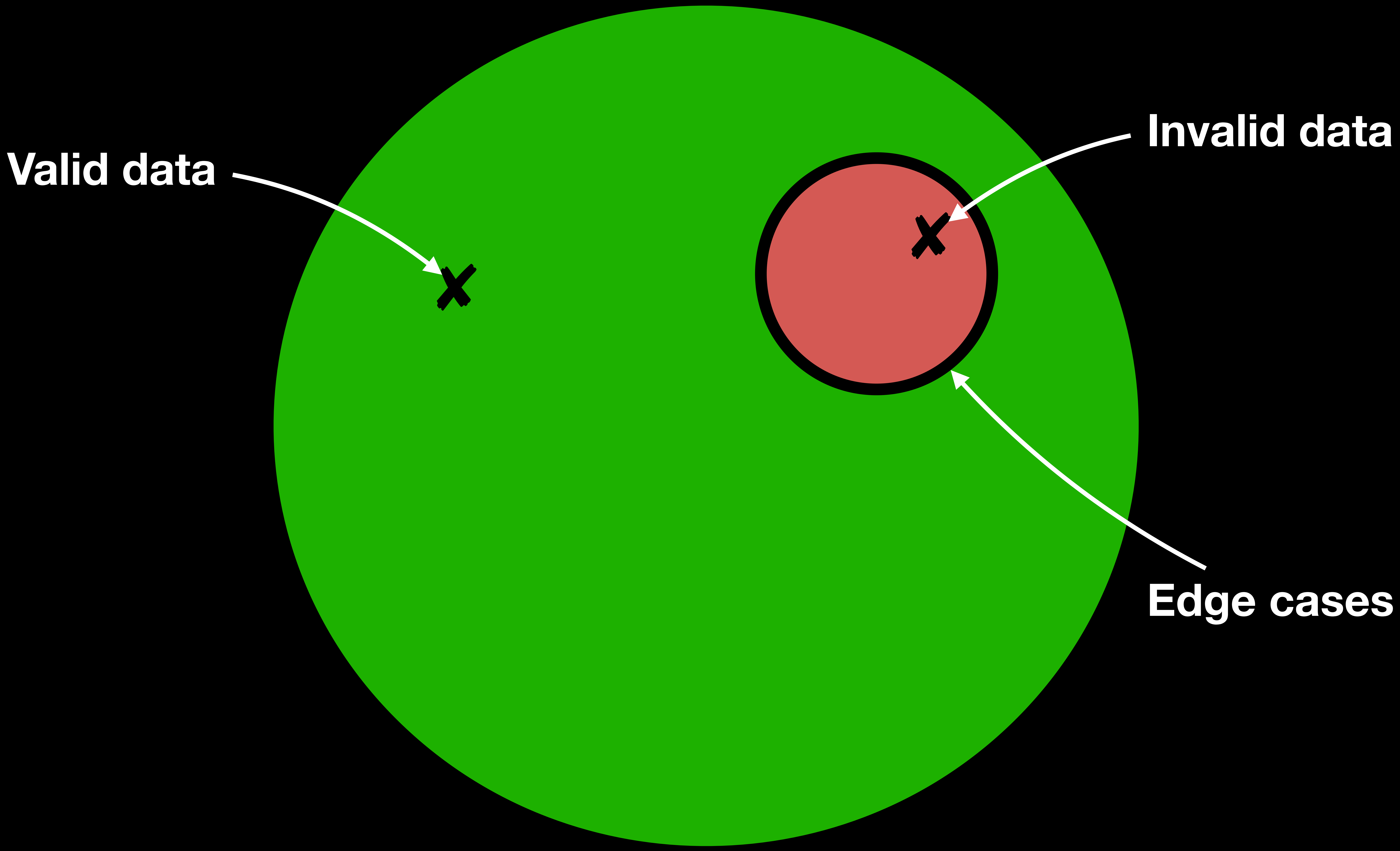
```
func testForValidData() {
    // Given
    let validData = MyData.valid()
    // When
    let result = myBusinessLogic(data)
    // Then
    XCTAssert(/* assert that result meets expectations */)
}
```

Classic unit tests

```
func testForInvalidData() {
    // Given
    let invalidData = MyData.invalid()
    // When
    let result = myBusinessLogic(invalidData)
    // Then
    XCTAssert(/* assert that result handles error */)
}
```


Valid data
Invalid data


Valid data
Invalid data


Valid data
Invalid data
Edge cases

Classic unit tests

- 👍 Great at catching regressions
- 👎 Don’t allow to learn anything new

Could tests discover edge cases ?

🤔

$$\text{Data} \xrightarrow{\hspace{3em}} \text{Properties}$$

$$\text{Properties} \longrightarrow \text{Data}$$

Let’s look at an example

Let’s look at an example

```swift
extension Array {
    public func reversed() -> Array<Element>
}
```

How do we define a property?

Using natural language

“If an array is reversed twice, then the result is equal to the input”

Using natural language

~~“If an array is reversed twice, then the result is equal to the input”~~

“Any array reversed twice is equal to itself”

Using first order logic

$$\forall\ a \in \text{Array},\ \mathit{reverse}(\mathit{reverse}(a)) = a$$

Using functional programming

```
func checkArrayReverse(_ array: Array) -> Bool {
    return array.reversed().reversed() == array
}
```

Using functional programming

$$\forall\ arr \in \text{Array},\ \textit{reverse}(\textit{reverse}(arr)) = arr$$

```swift
func checkArrayReverse(_ array: Array) -> Bool {
    return array.reversed().reversed() == array
}
```

Using functional programming

$\forall\ arr \in Array,\ reverse(reverse(arr)) = arr$

```
func checkArrayReverse(_ array: Array) -> Bool {
    return array.reversed().reversed() == array
}
```

Using functional programming

$\forall\ arr \in Array,\ \mathit{reverse}(\mathit{reverse}(arr)) = arr$

```
func checkArrayReverse(_ array: Array) -> Bool {
    return array.reversed().reversed() == array
}
```

Using functional programming

$$\forall \, arr \in Array, \; \mathit{reverse}(\mathit{reverse}(arr)) = arr$$

```
func checkArrayReverse(_ array: Array) -> Bool {
    return array.reversed().reversed() == array
}
```

Property-based testing in a nutshell

Property-based testing in a nutshell

For any input values (x,y, z, …)
Such that precondition (x, y, z, …) is satisfied
Property (x, y, z, …) must be true

Property-based testing in a nutshell

For any input value (array)
Property (array.reversed().reversed == array)
must be true

How do we implement such tests?

Introducing SwiftCheck

SwiftCheck

SwiftCheck is a framework that let us write and run property-based tests.

It works by following a simple set of rules:

A. Test the property using random input values

B. If the property fails, shrink the responsible input until the property no longer fails

C. Return either OK or the smallest counterexample

SwiftCheck

```
public protocol Arbitrary {
    public static var arbitrary: Gen<Self> { get }
    public static func shrink(_: Self) -> [Self]
}
```

Implementing a first property

```
property("Simple test of array reversal") <- forAll({ (array: [Int]) -> Testable in
    return array.reversed().reversed() == array
})
```

Now let’s define a more complex property 💪

A more complex property

We want to test that the elements of the array are correctly reversed.

For any input values (array, i)
Such that array.isEmpty == false, 0 ≤ i < (array.count - 1)
Property (array.reversed()[i] == array[(array.count - 1) - i])
must be true

A more complex property

```
property("Elements are correctly reversed") <- forAll({ (array: [Int]) -> Testable in
    return (!array.isEmpty ==> {
        let arrayIndices = Gen.fromElements(of: array.indices)

        return forAll(arrayIndices) { (index: Int) -> Testable in
            return array.reversed()[index] == array[(array.count - 1) - index]
        }
    })
})
```

A more complex property

```
property("Elements are correctly reversed") <- forAll({ (array: [Int]) -> Testable in
    return (!array.isEmpty ==> {
        let arrayIndices = Gen.fromElements(of: array.indices)

        return forAll(arrayIndices) { (index: Int) -> Testable in
            return array.reversed()[index] == array[(array.count - 1) - index]
        }
    })
})
```

A more complex property

```
property("Elements are correctly reversed") <- forAll({ (array: [Int]) -> Testable in
    return (!array.isEmpty ==> {
        let arrayIndices = Gen.fromElements(of: array.indices)

        return forAll(arrayIndices) { (index: Int) -> Testable in
            return array.reversed()[index] == array[(array.count - 1) - index]
        }
    })
})
```

A more complex property

```
property("Elements are correctly reversed") <- forAll({ (array: [Int]) -> Testable in
    return (!array.isEmpty ==> {
        let arrayIndices = Gen.fromElements(of: array.indices)

        return forAll(arrayIndices) { (index: Int) -> Testable in
            return array.reversed()[index] == array[(array.count - 1) - index]
        }
    })
})
```

A more complex property

```
property("Elements are correctly reversed") <- forAll({ (array: [Int]) -> Testable in
    return (!array.isEmpty ==> {
        let arrayIndices = Gen.fromElements(of: array.indices)

        return forAll(arrayIndices) { (index: Int) -> Testable in
            return array.reversed()[index] == array[(array.count - 1) - index]
        }
    })
})
```

A more complex property

```
property("Elements are correctly reversed") <- forAll({ (array: [Int]) -> Testable in
    return (!array.isEmpty ==> {
        let arrayIndices = Gen.fromElements(of: array.indices)

        return forAll(arrayIndices) { (index: Int) -> Testable in
            return array.reversed()[index] == array[(array.count - 1) - index]
        }
    })
})
```

Time to catch a bug 🐛

Time to catch a bug 🐛

Requirement: we need to validate email addresses.

We asked Google, and got this nice regex:

```
let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}"
```

How could we test this regex?

Time to catch a bug 🐛

```swift
func isValidEmail(_ candidateEmail: String) -> Bool {
    let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}"

    let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
    return emailTest.evaluate(with: candidateEmail)
}
```

Create a generator for emails

```
let upper: Gen<Character> = Gen<Character>.fromElements(in: "A"..."Z")
let lower: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")
let numeric: Gen<Character> = Gen<Character>.fromElements(in: "0"..."9")
let special: Gen<Character> = Gen<Character>.fromElements(of: ["!", "#", "$", "%", "&",
"'", "*", "+", "-", "/", "=", "?", "^", "_", "`", "{", "|", "}", "~", "."])

let localEmail = Gen<Character>.one(of: [
                upper,
                lower,
                numeric,
                special,
                ]).proliferateNonEmpty
                .suchThat({ $0[($0.endIndex - 1)] != "." })
                .map(String.init(_:))
```

Create a generator for emails

```swift
let upper: Gen<Character> = Gen<Character>.fromElements(in: "A"..."Z")
let lower: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")
let numeric: Gen<Character> = Gen<Character>.fromElements(in: "0"..."9")
let special: Gen<Character> = Gen<Character>.fromElements(of: ["!", "#", "$", "%", "&",
"'", "*", "+", "-", "/", "=", "?", "^", "_", "`", "{", "|", "}", "~", "."])

let localEmail = Gen<Character>.one(of: [
                upper,
                lower,
                numeric,
                special,
                ]).proliferateNonEmpty
                .suchThat({ $0[($0.endIndex - 1)] != "." })
                .map(String.init(_:))
```

Create a generator for emails

```swift
let upper: Gen<Character> = Gen<Character>.fromElements(in: "A"..."Z")
let lower: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")
let numeric: Gen<Character> = Gen<Character>.fromElements(in: "0"..."9")
let special: Gen<Character> = Gen<Character>.fromElements(of: ["!", "#", "$", "%", "&",
"'", "*", "+", "-", "/", "=", "?", "^", "_", "`", "{", "|", "}", "~", "."])

let localEmail = Gen<Character>.one(of: [
                upper,
                lower,
                numeric,
                special,
                ]).proliferateNonEmpty
                .suchThat({ $0[($0.endIndex - 1)] != "." })
                .map(String.init(_:))
```

Create a generator for emails

```swift
let upper: Gen<Character> = Gen<Character>.fromElements(in: "A"..."Z")
let lower: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")
let numeric: Gen<Character> = Gen<Character>.fromElements(in: "0"..."9")
let special: Gen<Character> = Gen<Character>.fromElements(of: ["!", "#", "$", "%", "&",
"'", "*", "+", "-", "/", "=", "?", "^", "_", "`", "{", "|", "}", "~", "."])

let localEmail = Gen<Character>.one(of: [
                upper,
                lower,
                numeric,
                special,
                ]).proliferateNonEmpty
                .suchThat({ $0[($0.endIndex - 1)] != "." })
                .map(String.init(_:))
```

Create a generator for emails

```
let upper: Gen<Character> = Gen<Character>.fromElements(in: "A"..."Z")
let lower: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")
let numeric: Gen<Character> = Gen<Character>.fromElements(in: "0"..."9")
let special: Gen<Character> = Gen<Character>.fromElements(of: ["!", "#", "$", "%", "&",
"'", "*", "+", "-", "/", "=", "?", "^", "_", "`", "{", "|", "}", "~", "."])

let localEmail = Gen<Character>.one(of: [
                upper,
                lower,
                numeric,
                special,
                ]).proliferateNonEmpty
                .suchThat({ $0[($0.endIndex - 1)] != "." })
                .map(String.init(_:))
```

Create a generator for emails

```swift
let upper: Gen<Character> = Gen<Character>.fromElements(in: "A"..."Z")
let lower: Gen<Character> = Gen<Character>.fromElements(in: "a"..."z")
let numeric: Gen<Character> = Gen<Character>.fromElements(in: "0"..."9")
let special: Gen<Character> = Gen<Character>.fromElements(of: ["!", "#", "$", "%", "&",
"'", "*", "+", "-", "/", "=", "?", "^", "_", "`", "{", "|", "}", "~", "."])

let localEmail = Gen<Character>.one(of: [
                upper,
                lower,
                numeric,
                special,
                ]).proliferateNonEmpty
                .suchThat({ $0[($0.endIndex - 1)] != "." })
                .map(String.init(_:))
```

Write the property

```
let emailGen = glue([localEmail, Gen.pure("@"), hostname, Gen.pure("."), tld])

let args = CheckerArguments(maxTestCaseSize: 10)

property("Email passes validation", arguments: args) <- forAll(emailGen) { (email: String) in
    return isValidEmail(email)
}.noShrinking
```

Write the property

```swift
let emailGen = glue([localEmail, Gen.pure("@"), hostname, Gen.pure("."), tld])

let args = CheckerArguments(maxTestCaseSize: 10)

property("Email passes validation", arguments: args) <- forAll(emailGen) { (email: String) in
    return isValidEmail(email)
}.noShrinking
```

Write the property

```
let emailGen = glue([localEmail, Gen.pure("@"), hostname, Gen.pure("."), tld])

let args = CheckerArguments(maxTestCaseSize: 10)

property("Email passes validation", arguments: args) <- forAll(emailGen) { (email: String) in
    return isValidEmail(email)
}.noShrinking
```

Write the property

```
let emailGen = glue([localEmail, Gen.pure("@"), hostname, Gen.pure("."), tld])

let args = CheckerArguments(maxTestCaseSize: 10)

property("Email passes validation", arguments: args) <- forAll(emailGen) { (email: String) in
    return isValidEmail(email)
}.noShrinking
```

Write the property

```
let emailGen = glue([localEmail, Gen.pure("@"), hostname, Gen.pure("."), tld])

let args = CheckerArguments(maxTestCaseSize: 10)

property("Email passes validation", arguments: args) <- forAll(emailGen) { (email: String) in
    return isValidEmail(email)
}.noShrinking
```

```
*** Failed! Proposition: Email passes validation
Falsifiable (after 2 tests):
|@o.az
```

Write the property

```
*** Failed! Proposition: Email passes validation
Falsifiable (after 2 tests):
|@o.az
```

We managed to catch a bug! 🐛

Can we test for invalid e-mails?

Can we test for invalid e-mails?

It’s actually impossible 😭 Why ?

Generating invalid e-mails would require a precondition to tell valid and invalid e-mails appart…

…which is exactly what we need to test!

In such a case, good old data based testing is the only option 🤷🏻‍♂️

Mixing both approaches

Mixing both approaches

```swift
extension Array where Element == Double {
    func average() -> Double
}
```

Mixing both approaches

```
// Data based testing
XCTAssertEqual([1, 5, 10, 50].average(), 16.5)
```

Mixing both approaches

```
// Data based testing
XCTAssertEqual([1, 5, 10, 50].average(), 16.5)

// Property based testing
property("average of products equals product of average") <- forAllNoShrink(genScalar,
                                                                               genValues) { (a: Double, x: [Double]) in
    let averageOfProducts = (a * x).average()
    let productOfAverages = a * x.average()

    return areEqual(averageOfProducts, productOfAverages)
}
```

Mixing both approaches

```
// Data based testing
XCTAssertEqual([1, 5, 10, 50].average(), 16.5)

// Property based testing
property("average of products equals product of average") <- forAllNoShrink(genScalar,
                                                                            genValues) { (a: Double, x: [Double]) in
    let averageOfProducts = (a * x).average()
    let productOfAverages = a * x.average()

    return areEqual(averageOfProducts, productOfAverages)
}

property("average of sums equals sum of averages") <- forAllNoShrink(genValues,
                                                                     genValues) { (x: [Double], y: [Double]) in

    makeSameSize(&x, &y)

    let averageOfSums = (x + y).average()
    let sumOfAverages = x.average() + y.average()

    return areEqual(averageOfSums, sumOfAverages)
}
```

How do I use this in my app? 🤔

How do I use this in my app?

Properties are “easy” to define for the lower levels of an app.

But as we get closer to business requirements, things get messy pretty fast.

However, consider the following property:

```
property("Trivial") <- forAll({ (input: Input) -> Testable in
    myBusinessLogic(input)

    return true
})
```

How do I use this in my app?

This property might – rightfully – seem trivial.

However, it actually performs two useful assertions:

- The function “myBusinessLogic” does return
- The function “myBusinessLogic” doesn’t crash

Pretty useful, for something so trivial 👌

You could even have a code-generation tool write it for you 🎅

How do I use this in my app?

“Trivial” properties also make sense in other contexts!

Consider UI testing: in most cases, it makes sense to test that views don’t overlap.

That’s quite easy to implement using property-based testing:

- Generate a random model
- Use it to fill up your view
- Iterate over its subviews and check that no two have frames that overlap

(Thank you to Pierre Felgines for this cool use case!)

When is this approach relevant?

When is this approach relevant?

- External data (user input, web services, etc.)
- Encoding / Decoding
- Regular expressions
- Custom sorting algorithm
- Timezone and Date arithmetic
- SDK development

Is there a lighter approach?

Is there a lighter approach?

```
let faker = Faker(locale: "fr_FR")

let fakeName = faker.name.firstName()
let fakeText = faker.lorem.sentence()
```

Is there a lighter approach?

```swift
let faker = Faker(locale: "fr_FR")

let fakeName = faker.name.firstName()
let fakeText = faker.lorem.sentence()


extension XCTestCase {
   func fuzz(_ times: Int = 100, test: () throws -> Void) rethrows {
        for _ in 1...times {
            try test()
        }
    }
}
```

Is there a lighter approach?

```
class TestCase: XCTestCase {
   func test() {
        fuzz {
            // the test gets run 100 times
        }
    }
}
```

A FILM BY CHRISTOPHER NOLAN
INTERSTELLAR
LEGENDARY
SYNCOPY
WARNER BROS. PICTURES

Murphy's Law doesn't mean
that something bad will happen.

What it means is
whatever can happen will happen.