Difflicious

Readable and Flexible Diffs

Jacob Wang

Scala Love 2022

Hello

- Software Developer at 

- @jatcwang
- I like types, libraries and tools :)

It’s Friday afternoon…

It’s Friday afternoon…

It’s Friday afternoon…

“If it compiles, it works”

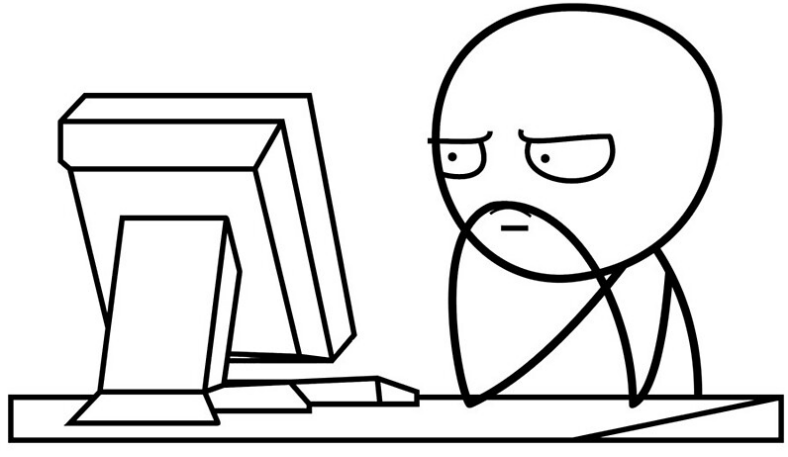
But the tests are failing…

But the tests are failing…

```
[info]   Complex(Map(0 -> Simple(0,zero), 1 -> Simple(1,ten)),List(Simple(1,first), Simple(2
,third)),Set(Simple(1,first), Simple(2,third))) was not equal to Complex(Map(0 -> Simple(0,z
ero), 2 -> Simple(2,two)),List(Simple(1,first), Simple(2,second), Simple(3,third)),Set(Simpl
e(1,first), Simple(2,second))) (Ex3_DataStructures.scala:51)
```

But the tests are failing…

```
[info]   Complex(Map(0 -> Simple(0,zero), 1 -> Simple(1,ten)),List(Simple(1,first), Simple(2
,third)),Set(Simple(1,first), Simple(2,third))) was not equal to Complex(Map(0 -> Simple(0,z
ero), 2 -> Simple(2,two)),List(Simple(1,first), Simple(2,second), Simple(3,third)),Set(Simpl
e(1,first), Simple(2,second))) (Ex3_DataStructures.scala:51)
```



That’s enough for the day

That’s enough for the day

A career in goat farming looks great all of a sudden…

Tests should be nicer

Tests should be nicer

- Tell what’s wrong at a glance

Tests should be nicer

- Tell what’s wrong at a glance
- Need more flexibility in our comparisons

Tests should be nicer

- Tell what's wrong at a glance
- Need more flexibility in our comparisons
- Difflicious to the rescue!

DIFFLICIOUS STEP-BY-STEP

DIFFLICIOUS STEP-BY-STEP

- Add **difflicious** to your test dependencies
  - difflicious-munit
  - difflicious-scalatest
  - Scala 2.13 & 3

DIFFLICIOUS STEP-BY-STEP

- Add **difflicious** to your test dependencies
  - difflicious-munit
  - difflicious-scalatest
  - Scala 2.13 & 3
- Create `Differ`s (derive + configure)

DIFFLICIOUS STEP-BY-STEP

- Add **difflicious** to your test dependencies
  - difflicious-munit
  - difflicious-scalatest
  - Scala 2.13 & 3
- Create `Differ`s (derive + configure)
  - Differ.apply (e.g. `Differ[List[Int]]`) to “summon” an instance

DIFFLICIOUS STEP-BY-STEP

- Add **difflicious** to your test dependencies
  - difflicious-munit
  - difflicious-scalatest
  - Scala 2.13 & 3
- Create `Differ`s (derive + configure)
  - Differ.apply (e.g. `Differ[List[Int]]`) to “summon” an instance
- Use `Differ`s to diff values

A SIMPLE EXAMPLE

```scala
import difflicious.Differ
import difflicious.implicits.*
import difflicious.munit.MUnitDiff.*

case class Foo(i: Int, s: String)

// Create the differ
given differ: Differ[Foo] = Differ.derived

val expected = Foo(1, "a")
val actual = Foo(1, "b")

// Assert no difference between two values
differ.assertNoDiff(actual, expected)
```

A SIMPLE EXAMPLE

```scala
import difflicious.Differ
import difflicious.implicits.*
import difflicious.munit.MUnitDiff.*

case class Foo(i: Int, s: String)

// Create the differ
given differ: Differ[Foo] = Differ.derived

val expected = Foo(1, "a")
val actual = Foo(1, "b")

// Assert no difference between two values
differ.assertNoDiff(actual, expected)
```

```
Foo(
  i: 1,
  s: "b" -> "a"
)
```

DATA STRUCTURES DIFFS

DATA STRUCTURES DIFFS

- **Seq**, **Map** and **Set** are supported by default

DATA STRUCTURES DIFFS

- **Seq**, **Map** and **Set** are supported by default
- **Seq** pair by index when diffing

DATA STRUCTURES DIFFS

- **Seq**, **Map** and **Set** are supported by default
- **Seq** pair by index when diffing
- **Map** pair by key when diffing

DATA STRUCTURES DIFFS

- **Seq**, **Map** and **Set** are supported by default
- **Seq** pair by index when diffing
- **Map** pair by key when diffing
- **Set** uses `==`

DATA STRUCTURES DIFFS

- **Seq**, **Map** and **Set** are supported by default
- **Seq** pair by index when diffing
- **Map** pair by key when diffing
- **Set** uses `==`

```
val expected = Vector(
  Map(
    "Bono" -> Set("Plant"),
    "Sven" -> Set("Plates")
  )
)
val actual = Vector(
  Map(
    "Bono" -> Set("Plant", "Plates")
    // Sven?
  )
)

// "Summon" an instance using Differ.apply
val differ = Differ[Vector[Map[String, Set[String]]]]

differ.assertNoDiff(actual, expected)
```

```
Vector(
  Map(
    "Bono" -> Set(
        "Plant",
        "Plates"
      ),
    "Sven" -> Set(
        "Plates",
      )
  ),
)
```

From the diff we see that:

- Bono unexpectedly showed up with Plates
- Sven is missing

PAIRING THINGS UP

- Sometimes, The default diffing behaviour of **Seq** and **Set** isn't what we want

PAIRING THINGS UP

- Sometimes, The default diffing behaviour of **Seq** and **Set** isn't what we want

```
val expectedCats = List(
  Cat("Lucy", 14),
  // Sven?
  Cat("Bono", 8)
)

val actualCats = List(
  Cat("Bono", 7),
  Cat("Sven", 3)
  // Lucy?
)

Differ[List[Cat]].assertNoDiff(actualCats, expectedCats)
```

PAIRING THINGS UP

- Sometimes, The default diffing behaviour of **Seq** and **Set** isn't what we want

```
val expectedCats = List(
  Cat("Lucy", 14),
  // Sven?
  Cat("Bono", 8)
)

val actualCats = List(
  Cat("Bono", 7),
  Cat("Sven", 3)
  // Lucy?
)

Differ[List[Cat]].assertNoDiff(actualCats, expectedCats)
```

```
List(
  Cat(
    name: "Bono" -> "Lucy",
    age: 7 -> 14,
  ),
  Cat(
    name: "Sven" -> "Bono",
    age: 3 -> 8,
  ),
)
```

PAIRING THINGS UP (2)

- We can use `pairBy` to get more meaningful diffs!

PAIRING THINGS UP (2)

- We can use `pairBy` to get more meaningful diffs!

```
val catsDiffer = Differ[List[Cat]].pairBy(_.name)

catsDiffer.assertNoDiff(actualCats, expectedCats)
```

PAIRING THINGS UP (2)

- We can use `pairBy` to get more meaningful diffs!

```
val catsDiffer = Differ[List[Cat]].pairBy(_.name)

catsDiffer.assertNoDiff(actualCats, expectedCats)
```

```
List(
  Cat(
    name: "Bono",
    age: 7 -> 8
  ),
  Cat(
    name: "Sven",
    age: 3
  ),
  Cat(
    name: "Lucy",
    age: 14
  )
)
```

Ignoring fields

Ignoring fields

- Sometimes, we **can't** or **don't** want to compare certain fields

Ignoring fields

- Sometimes, we **can't** or **don't** want to compare certain fields
- e.g. Externally generated IDs, out-of-context data for current test

Ignoring fields

- Sometimes, we **can't** or **don't** want to compare certain fields
- e.g. Externally generated IDs, out-of-context data for current test

```
// Example business logic:
// Register dogs, returning dogs with their ID
def registerDogs(dogData: List[DogData]): List[Dog] =

  val dogIds = writeToDatabaseReturningIds(dogData)

  fetchDogsById(dogIds) // We cannot predict the IDs generated!
```

```
case class DogData(name: String)
case class Dog(id: Int, name: String)
```

Ignoring fields (2)

Let’s ignore dog IDs from comparison

```
given Differ[Dog] = Differ.derived

val expectedDogs = List(Dog(0, "Wolfy"), Dog(0, "Bamboo"))

val dogData = List(DogData("Wolfy"), DogData("Bamboa"))
val actualDogs = registerDogs(dogData)
```

Ignoring fields (2)

Let’s ignore dog IDs from comparison

```
given Differ[Dog] = Differ.derived

val expectedDogs = List(Dog(0, "Wolfy"), Dog(0, "Bamboo"))

val dogData = List(DogData("Wolfy"), DogData("Bamboa"))
val actualDogs = registerDogs(dogData)
```

```
// Setup differ with field ignores
val dogDiffer: Differ[List[Dog]] = Differ[List[Dog]].ignoreAt(_.each.id)

dogDiffer.assertNoDiff(actualDogs, expectedDogs)
```

Ignoring fields (2)

Let’s ignore dog IDs from comparison

```
given Differ[Dog] = Differ.derived

val expectedDogs = List(Dog(0, "Wolfy"), Dog(0, "Bamboo"))

val dogData = List(DogData("Wolfy"), DogData("Bamboa"))
val actualDogs = registerDogs(dogData)
```

```
// Setup differ with field ignores
val dogDiffer: Differ[List[Dog]] = Differ[List[Dog]].ignoreAt(_.each.id)

dogDiffer.assertNoDiff(actualDogs, expectedDogs)
```

```
List(
  Dog(
    id: [IGNORED],
    name: "Wolfy",
  ),
  Dog(
    id: [IGNORED],
    name: "Bamboa" -> "Bamboo",
  ),
)
```

Deep configuration!

Deep configuration!

- `.ignoreAt(_.each.id)`

Deep configuration!

- `.ignoreAt(_.each.id)`
  - `_.each.id` is the “path expression”

Deep configuration!

- `.ignoreAt(_.each.id)`
  - `_.each.id` is the “path expression”
- Allow us to easily tweak behaviour for the current test

Deep configuration!

- `.ignoreAt(_.each.id)`
  - `_.each.id` is the “path expression”
- Allow us to easily tweak behaviour for the current test

There are other useful configuration methods like `.configure` and `.replace`

```
given Differ[Employee] = Differ.derived
given Differ[DogZoo] = Differ.derived

val newDogsDiffer: Differ[List[Dog]] = // a heavily configured Differ[List[Dog]]

val configuredDogZooDiffer = Differ[DogZoo]
  // .configure allows you to "focus" on a differ inside to make multiple tweaks
 to it
  .configure(_.employees)
 (_.ignoreAt(_.each.age).ignoreAt(_.each.hoursWorked).pairBy(_.name))
  // .replace will replace the Differ at the given path
  .replace(_.dogs)(newDogsDiffer)
```

Path Expressions

Differ Type	Allowed Paths	Explanation
Seq	.each	Traverse down to the Differ used to compare the elements
Set	.each	Traverse down to the Differ used to compare the elements
Map	.each	Traverse down to the Differ used to compare the values of the Map
Case Class	(any case class field)	Traverse down to the Differ of the field
Sealed Trait / Enum	.subType[SomeSubType]	Traverse down to the Differ for the specified sub type

Putting it all together

Let’s track office capacity and that everyone is dressed correctly :)

```scala
sealed trait Person:
  def name: String

object Person:
  case class Contractor(name: String) extends Person
  case class Employee(name: String, attire: String) extends Person

case class OfficeCapacity(
  home: Set[Person],
  office: Map[Int, Person]
)
```

```scala
// Derive default differs
given Differ[Person] = Differ.derived
given Differ[OfficeCapacity] = Differ.derived

val officeCapacityDiffer = Differ[OfficeCapacity]
  // Pants optional when WFH ;)
  .ignoreAt(_.home.each.subType[Employee].attire)
  // Pair people by name when comparing
  .configure(_.home)(_.pairBy(_.name))
```

What a diff output might look like:

```
OfficeCapacity(
  home: Set(
    Employee(
      name: "Sarah",
      attire: [IGNORED],
    ),
    Contractor != Employee
    === Obtained ===
    Contractor(
      name: "Paolo",
    )
    === Expected ===
    Employee(
      name: "Paolo",
      attire: [IGNORED],
    ),
  ),
  office: Map(
    1 -> Employee(
        name: "Percy",
        attire: "casual" -> "suit",
      ),
  ),
)
```

Final tips

Final tips

- Use `Differ.useEquals` if just want to compare a type by `==`

Final tips

- Use `Differ.useEquals` if just want to compare a type by `==`
- **IntelliJ**: need to adjust some settings to not make all test failure color red
  - Editor | Color Scheme | Console Colors | Console | Error Output, uncheck the red foreground color

Thank you!

- SoftwareMill
  - Many inspirations from **diffx**
- EPFL & all other Scala 3 contributors
  - *Givens* save keystrokes
  - Macros are tremendous fun