The Scalactic Way
ScalaDays 2015, San Francisco

Bill Venners
Artima, Inc.
Escalate Software
The Scalactic Way

Use types to achieve quality.
Three prongs of the quality pitchfork

ScalaTest - quality through tests

Scalactic - quality through types

SuperSafe - quality through static analysis
Scalactic Anyvals

// n > 0
PosInt
PosLong
PosFloat
PosDouble

// n >= 0
PosZInt
PosZLong
PosZFloat
PosZDouble
Demo!
case class PropertyCheckConfig(
  minSuccessful: Int = 100,
  maxDiscarded: Int = 500,
  minSize: Int = 0,
  maxSize: Int = 100,
  workers: Int = 1
) {
  require(minSuccessful > 0, "minSuccessful had value \(\) +
          minSuccessful + ", but must be greater than zero")
  require(maxDiscarded >= 0, "maxDiscarded had value \(\) +
          maxDiscarded + ", but must be greater than or equal to zero")
  require(minSize >= 0, "minSize had value \(\) + minSize +
          ", but must be greater than or equal to zero")
  require(maxSize >= 0, "maxSize had value \(\) + maxSize +
          ", but must be greater than or equal to zero")
  require(minSize <= maxSize, "minSize had value \(\) + minSize +
          ", which must be less than or equal to maxSize, which had value \(\) + maxSize)
  require(workers > 0, "workers had value \(\) + workers +
          ", but must be greater than zero")
}
case class PropertyCheckConfiguration(
    minSuccessful: PosInt = 100,
    maxDiscardedFactor: PosZDouble = 5.0, // This changed in ScalaCheck
    minSize: PosZInt = 0,
    maxSize: PosZInt = 100,
    workers: PosInt = 1
) {
    require(minSize <= maxSize, "minSize had value " + minSize + ", which must be less than or equal to maxSize, which had value " + maxSize)
}
case class PropertyCheckConfiguration(
  minSuccessful: PosInt = 100,
  maxDiscardedFactor: PosZDouble = 5,
  minSize: PosZInt = 0,
  sizeRange: PosZInt = 101,
  workers: PosInt = 1
)
Don’t need these tests

```java
intercept[IllegalArgumentException] { 
    PropertyCheckConfiguration(
        minSuccessful = 0 
    )
}
}
intercept[IllegalArgumentException] { 
    PropertyCheckConfiguration(
        minSuccessful = -1 
    )
}
```
final class EvenInt private (val value: Int) extends AnyVal {
  override def toString: String = s"EvenInt($value)"
}

object EvenInt {
  def from(value: Int): Option[EvenInt] =
    if (value % 2 == 1) Some(new EvenInt(value)) else None
  def apply(value: Int): EvenInt = macro EvenIntMacro.apply
}
final class EvenInt private (val value: Int) extends AnyVal {
  override def toString: String = s"EvenInt($value)"
}

object EvenInt {
  def from(value: Int): Option[EvenInt] = 
    if (value % 2 == 0) Some(new EvenInt(value)) else None
  def apply(value: Int): EvenInt = macro EvenIntMacro.apply
}

object EvenIntMacro extends CompileTimeAssertions {
    val notValidMsg = "EvenInt.apply can only be invoked on even Int literals, like EvenInt(3)."
    val notLiteralMsg = "EvenInt.apply can only be invoked on Int literals, like " + "EvenInt(3). Please use EvenInt.from instead."
    ensureValidIntLiteral(c)(value, notValidMsg, notLiteralMsg) { i => i % 2 == 1 }
    c.universe.reify { EvenInt.from(value.splice).get }
  }
}
def half(even: Int): Int = {
  require(even % 2 == 0, s"$even was not even")
  even / 2
}
def half(even: EvenInt): Int = {
    // Why no require here?
    even / 2
}
The Scalactic Way

Use types where practical to focus and reduce the need for tests (both assert and require).
Stories!
Safety, but complexity

// ScalaTest 2.0
scala> equal("one")
res0: MatcherFactory1[Any, Equality] = equal ("one")

// ScalaTest 3.0
scala> equal("one")
res0: MatcherFactory1[Any, EvidenceThat[String]#CanEqual] = equal ("one")
Safety, but complexity

// ScalaTest 2.0
scala> equal ("one")
res0: MatcherFactory1[Either, Equality] = equal ("one")

// ScalaTest 3.0
scala> equal("one")
res0: MatcherFactory1[Either, EvidenceThat[String]#CanEqual] = equal ("one")

// ScalaTest 2.0
scala> Some("one") should (be (defined) and not equal "one")

// ScalaTest 3.0
scala> Some("one") should (be (defined) and not equal "one")
<console>:20: error: could not find implicit value for parameter typeClass2:
  org.scalactic.enablers.EvidenceThat[String]#CanEqual[Some[String]]
  Some("one") should not equal "one"
  ^
Let's graph something!
The Scalactic Way

The type system does not offer the best solution to every problem.
The SuperSafe Way

- Can run all the time
- Doesn’t hurt compile time
- No warnings, only errors
- Not a linter; a “Scala subset policy enforcer”
1. Milano
2. New York
3. Tokyo
4. Amsterdam
5. Paris
6. Brussels
Q => A

artima.com/flyer

SuperSafe, Escalate Stairway to Scala training

50% discount at Artima booth