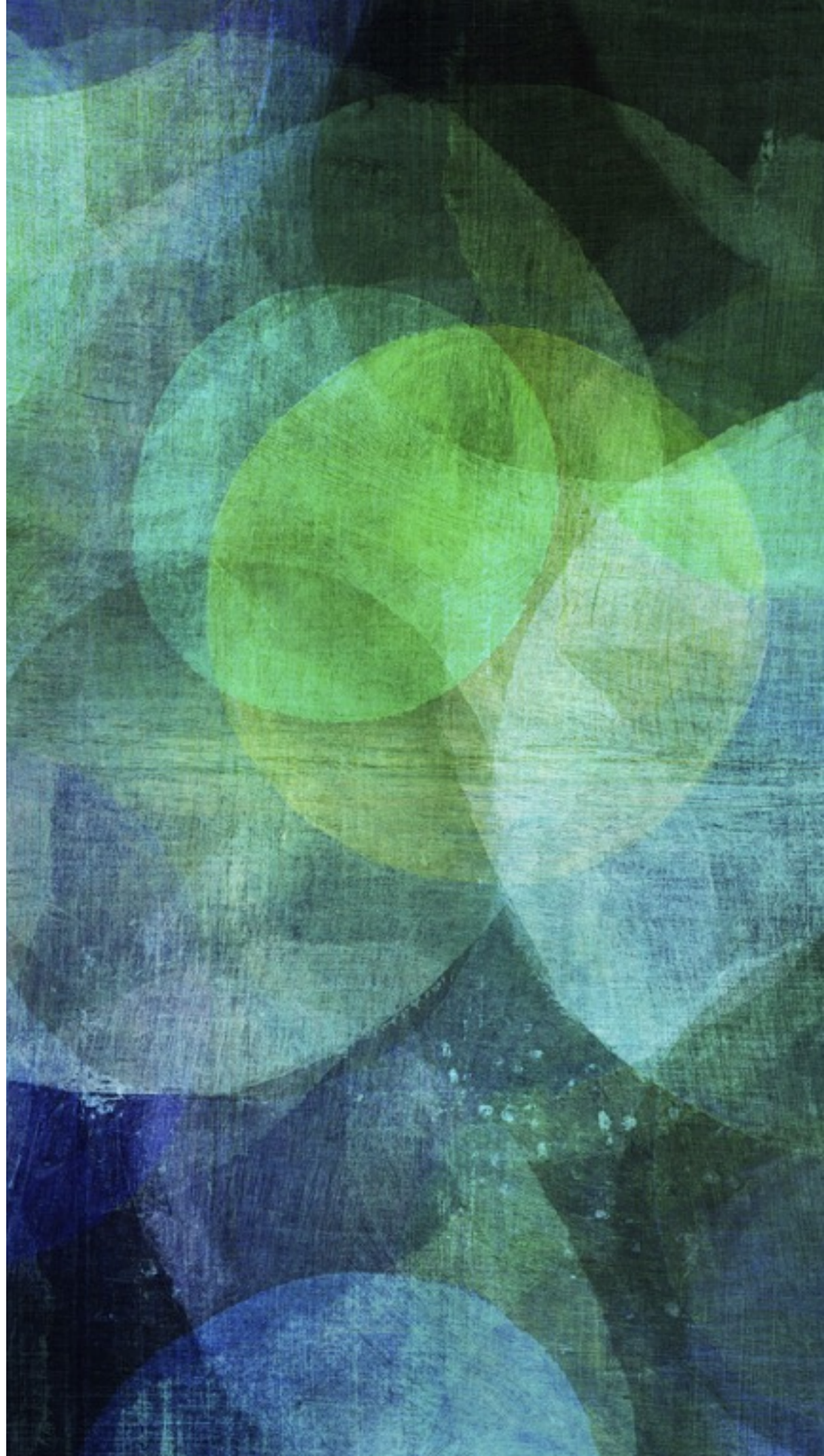
# SCALA
# PART TWO

*Beyond the basics*



# OUTLINE

- apply
- objects
- packages
- pattern matching
- case classes
- try-catch-finally

# APPLY METHODS

- Syntactic sugar for when a class or object has one main use.

```
class Foo {}
object FooMaker {
  def apply() = new Foo
}

val newFoo = FooMaker()
```

# WAIT

- Previous example used `object`

# OBJECTS

- Objects are used to hold ***single*** instances of a class

```
object Timer {
  var count = 0
  def currentCount(): Long = {
    count += 1
    count
  }
}
```

# USING OBJECTS

- Because singleton, no need to instantiate with `new`

```
object Timer {
  var count = 0
  def currentCount(): Long = {
    count += 1
    count
  }
}

Timer.currentCount()
```

# COMPANION OBJECTS

```
class Bar(foo: String) {
  def show() = println("Foo = " + foo)
}
object Bar {
  def apply(foo: String) = new Bar(foo)
}
```

```
Bar("Foo").show()
```

- Note same name
- Trivial example only used to remove need for `new`

# APPLY AND OBJECTS

- Any questions?

- Exercise: Write an object called “Color” with value bindings for hex color codes of red, blue and yellow
- Display the hex value of red such as Color.RED

- Next… packages

# PACKAGES

- You can organize Scala code into “packages” (think folders)

```
package com.datastax

object Example {
  def show() = println("show")
}
```

- com.datastax.Example.show

# PATTERN MATCHING

- Value and type

# PATTERN MATCHING ON VALUE

```
val times = 1
times match {
  case 1 => "one"
  case 2 => "two"
  case _ => "some other number"
}
```

# PATTERN MATCHING ON VALUE.... AND GUARDS

```
val times = 1
times match {
  case i if i == 1 => "one"
  case i if i == 2 => "two"
  case _ => "some other number"
}
```

# PATTERN MATCHING ON TYPE

```
def bigger(o: Any): Any = {
  o match {
    case i: Int if i < 0 => i - 1
    case i: Int => i + 1
    case d: Double if d < 0.0 => d - 0.1
    case d: Double => d + 0.1
    case text: String => text + "s"
  }
}
```

# PATTERN MATCHING EXERCISE

- Write and run an example of pattern matching on value; e.g. create an object with a method that accepts one argument. Display a value based on pattern matching of the value of the argument

- Update your example to use at least one guard

- Bonus: write an example with pattern matching based on type

# CASE CLASSES

- Used to conveniently store and match on the contents of a class
- Similar to object, no need to instantiate with `new`

# CASE CLASS EXAMPLES

```
case class Calculator(brand: String, model: String)
val hp20b = Calculator("HP", "20b")
```

# CASE CLASS EXAMPLES

- case classes automatically have equality and nice `toString` methods based on the constructor arguments.

```
val hp20b = Calculator("HP", "20b")
val hp20B = Calculator("HP", "20b")
```

```
hp20b == hp20B
```

# CASE CLASSES WITH PATTERN MATCHING

- val hp20b = Calculator("HP", "20B")
- val hp30b = Calculator("HP", "30B")
- def calcType(calc: Calculator) = calc match {
- case Calculator("HP", "20B") => "financial"
- case Calculator("HP", "48G") => "scientific"
- case Calculator("HP", "30B") => "business"
- case Calculator(ourBrand, ourModel) => "Calculator: %s %s is of unknown type".format(ourBrand, ourModel)
- }

# CASE CLASSES WITH PATTERN MATCHING... LAST MATCH OPTIONS

- case Calculator(ourBrand, ourModel) => "Calculator: %s %s is of unknown type".format(ourBrand, ourModel)
- could also be
- case Calculator(_, _) => "Calculator of unknown type”
- case _ => "Calculator of unknown type”
- case c@Calculator(_, _) => "Calculator: %s of unknown type".format(c)

# PATTERN MATCHING

- Any questions?

- Next exception handling… with pattern matching

# EXCEPTIONS

- // Trivial example

# EXCEPTION TRY HANDLING MAY BE EXPRESSION

```
try {
  remoteCalculatorService.add(1, 2)
} catch {
  case e: ServerIsDownException => log.error(e, "the
remote calculator service is unavailable")
} finally {
  remoteCalculatorService.close()
}
```

# PART 2 CONCLUSION

- apply
- objects
- packages
- pattern matching
- case classes
- try-catch-finally