Scala

Sydney Payne, Genny Hyla, Eliana Kang

Background: The Scala Language

- Creator: Martin Odersky
  - Student of Niklaus Wirth, creator of Pascal and several other languages
  - Co-designer of Generic Java
  - Known as the “father” of the javac compiler

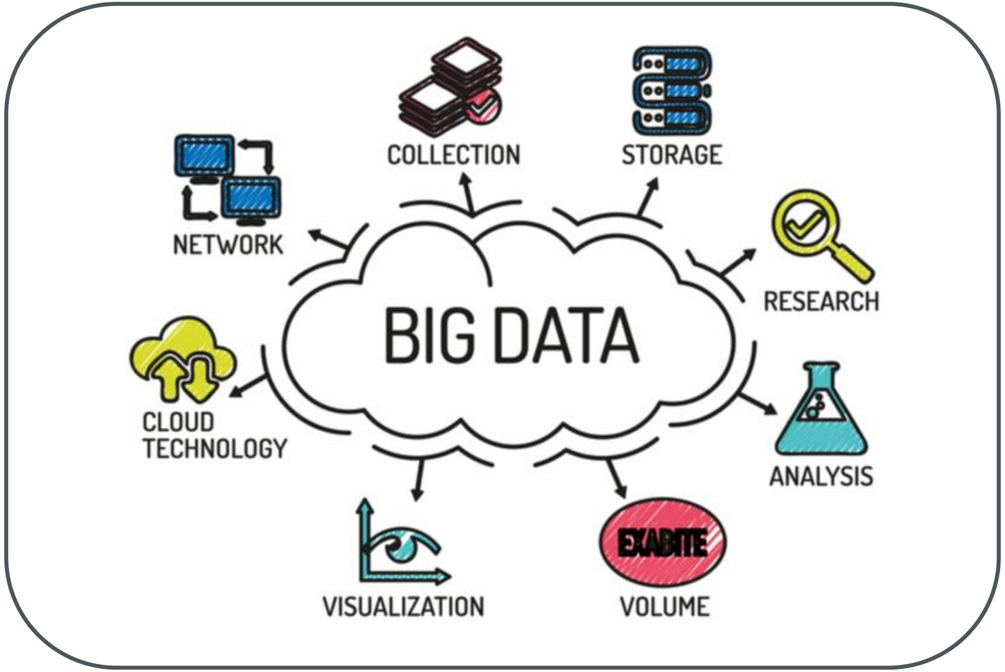
Design Philosophy:

“Scala was designed to show that a fusion of functional and object-oriented programming is possible and practical.”

– Martin Odersky

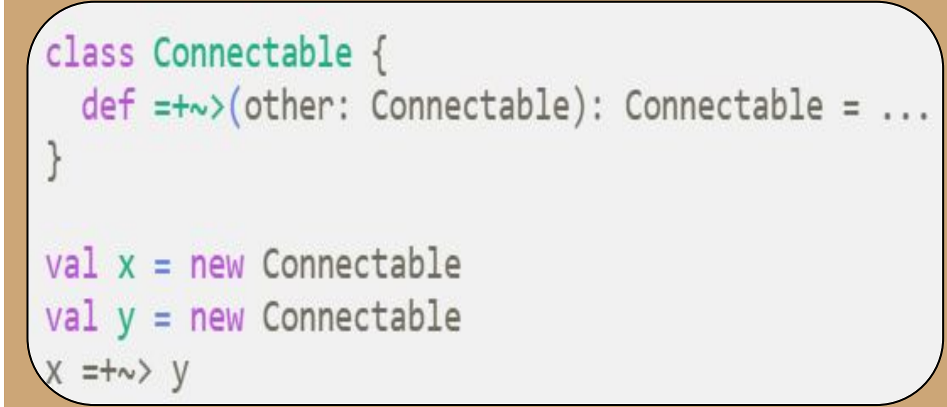
Continued

- Uses in server-side, frontend, data processing, scripting
- Companies: Netflix, Twitter, LinkedIn

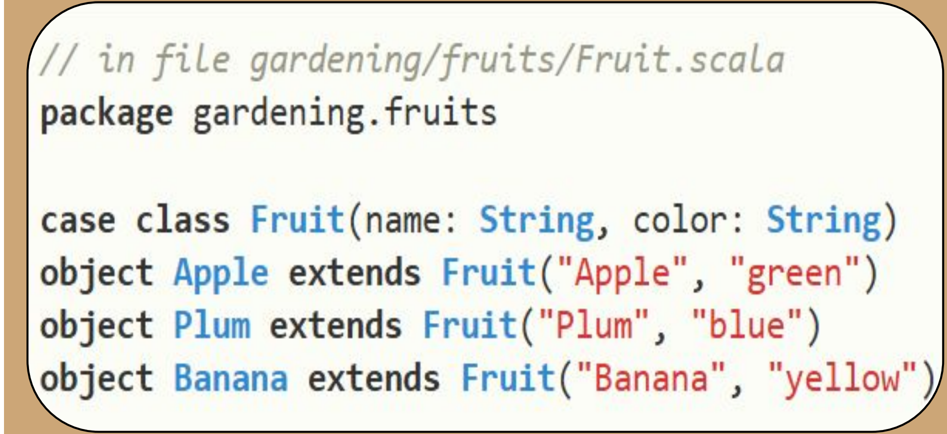


Names, Bindings, and Scopes

- Statically typed and type inference
  - val x : Int = 5 (static typing)
  - val x = 5 (type interference)
- Unique naming conventions
  - Symbolic method names
  - Backticking
- Scopes:
  - Static scoping
  - Local scope
  - No true global scope



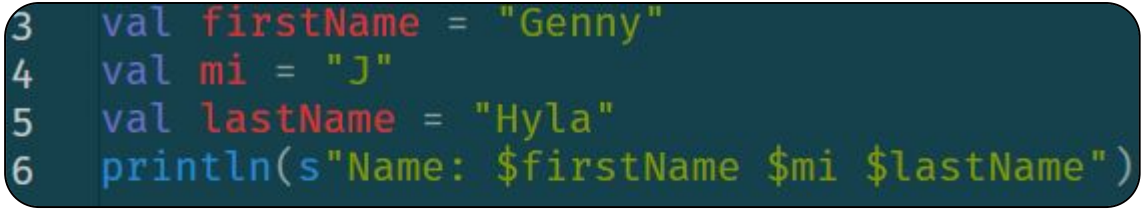
```scala
class Connectable {
  def =+~>(other: Connectable): Connectable = ...
}

val x = new Connectable
val y = new Connectable
x =+~> y
```



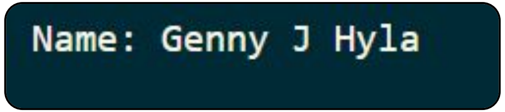
```scala
// in file gardening/fruits/Fruit.scala
package gardening.fruits

case class Fruit(name: String, color: String)
object Apple extends Fruit("Apple", "green")
object Plum extends Fruit("Plum", "blue")
object Banana extends Fruit("Banana", "yellow")
```

Data types

- Scala is able to implicitly or explicitly infer the data type
  - val a = 1 (implicit)
  - Val a: Int = 1 (explicit)
- Similar to Java, except they are full classes, everything is an object so first letter is capitalized
  - Int/Double/Float/Boolean/String/Char/etc
  - String useability
    - Supports interpolation: println(s”Name: $firstName $mi $lastName”)
    - Write multiline strings with “““ ”””

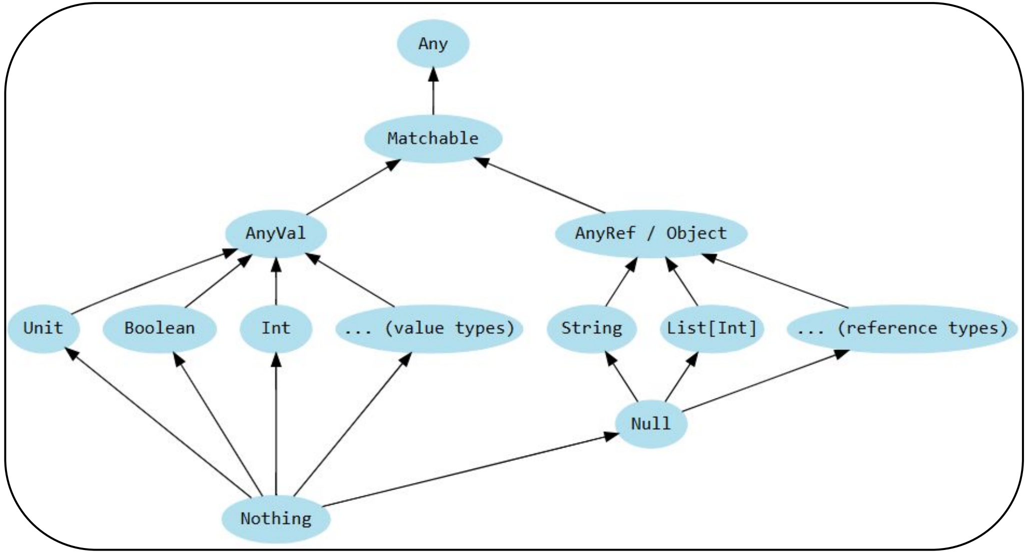


```scala
val firstName = "Genny"
val mi = "J"
val lastName = "Hyla"
println(s"Name: $firstName $mi $lastName")
```



```
Name: Genny J Hyla
```

Continued

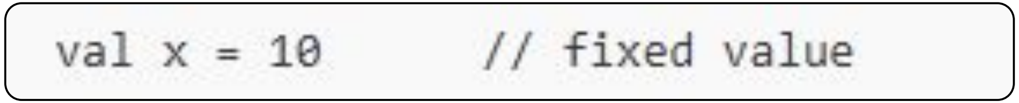


- Void: methods with no return
- Nothing: a subtype of all types, bottom type, there is no value with this type
- Null: subtype of all reference types

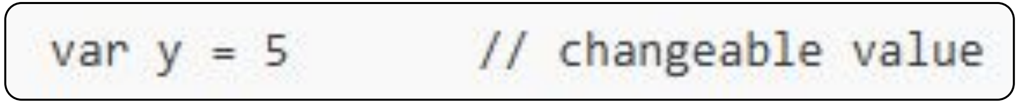
- Any: supertype of all types, top type
- Matchable: subtype of Any, used to mark pattern matching types
- AnyVal: subtype of matchable, used for value types, includes Unit which carries no information ()
- AnyRef: subtype of matchable, used for reference types, all user-defined types are subtypes

Assignment statements

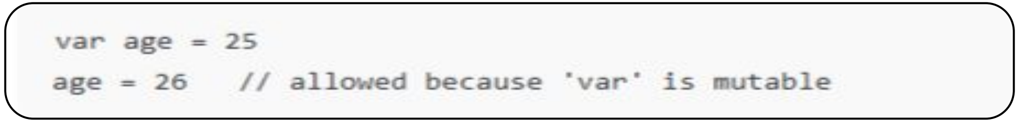
- Assignment gives a name (variable) a value.
- Two ways to assign in Scala:
  - val: can’t change (fixed)
  - var: can change



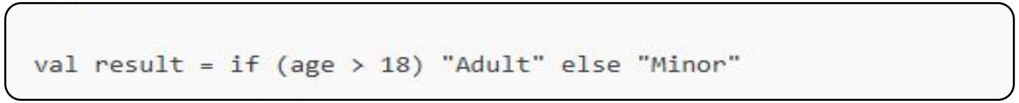
```
val x = 10        // fixed value
```



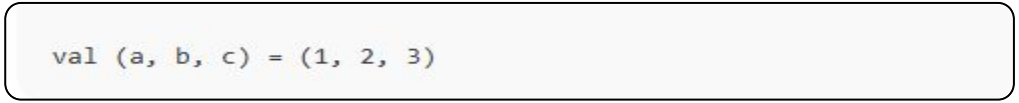
```
var y = 5         // changeable value
```



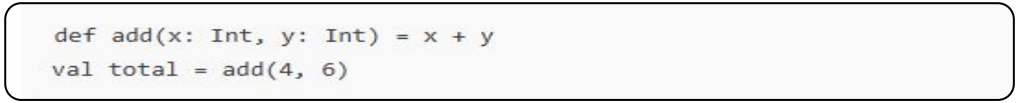
```
var age = 25
age = 26   // allowed because 'var' is mutable
```



```
val result = if (age > 18) "Adult" else "Minor"
```



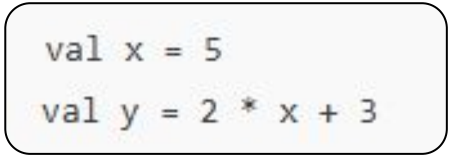
```
val (a, b, c) = (1, 2, 3)
```



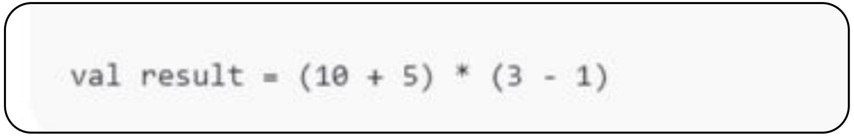
```
def add(x: Int, y: Int) = x + y
val total = add(4, 6)
```

Expressions in Scala

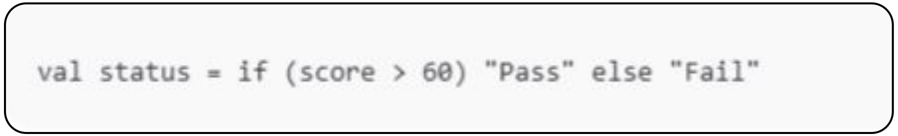
- Evaluation (Execution): Carrying out the instructions in code.
- Expressions are the texts that describes programs.
  - Expanding Expressions
  - Nested (Compound) Expressions
  - Expressive Expressions
- When an expression is evaluated (run), is produces a value.
  - The results stored in the computer’s memory after evaluation.



```
val x = 5
val y = 2 * x + 3
```



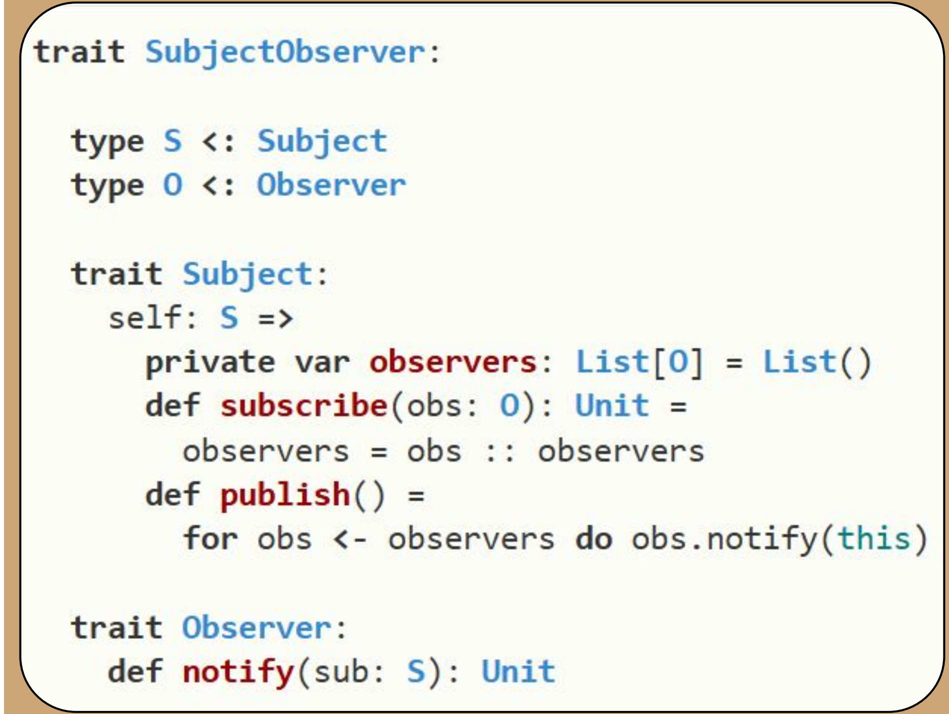
```
val result = (10 + 5) * (3 - 1)
```



```
val status = if (score > 60) "Pass" else "Fail"
```

Object-oriented programming

- Classes: implement interfaces specified by traits
- Traits: specify abstract interfaces or concrete implementations
  - Mixin compositions
- Extends OOP
  - Self-type annotations
  - Abstract type members



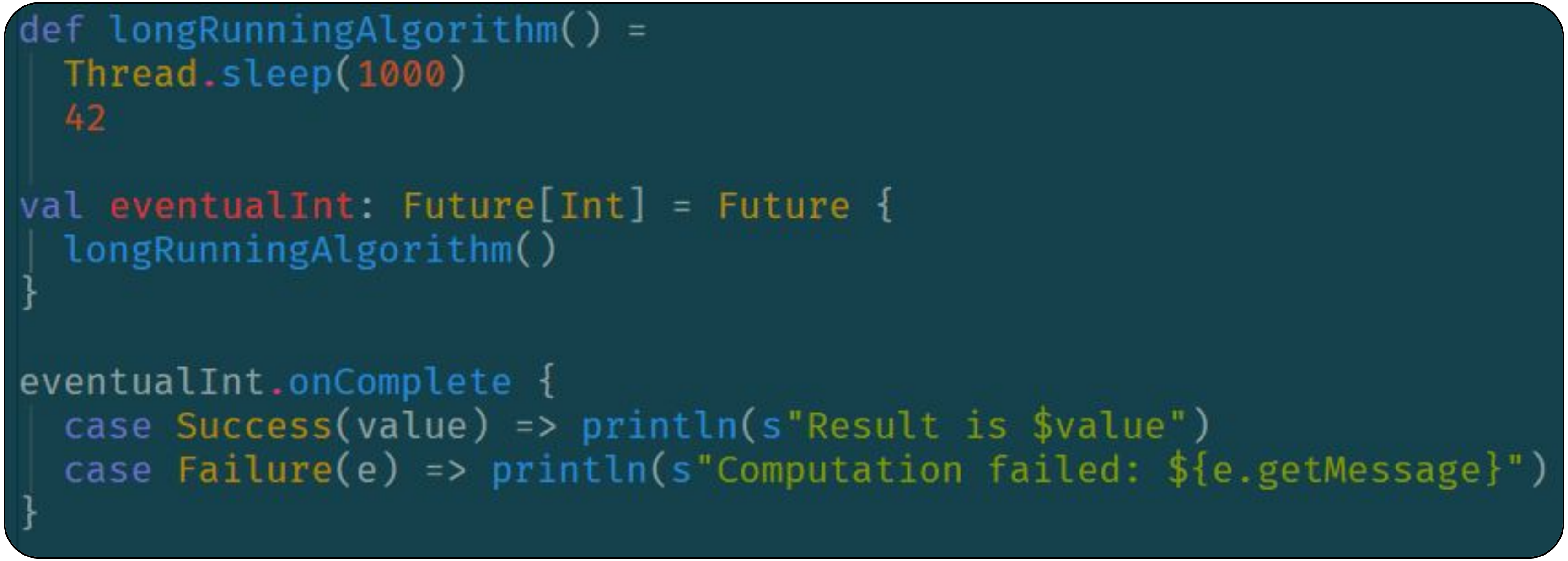
```
trait SubjectObserver:

  type S <: Subject
  type O <: Observer

  trait Subject:
    self: S =>
      private var observers: List[O] = List()
      def subscribe(obs: O): Unit =
        observers = obs :: observers
      def publish() =
        for obs <- observers do obs.notify(this)

  trait Observer:
    def notify(sub: S): Unit
```

Concurrency

- Scala has an interesting feature called ‘Future’
  - Runs tasks off the main thread
  - This represents a value that might not currently be available
  - Used to call algs that run an indeterminate amount of time
  - Will always be an instance of ‘scala.util.Try’ types: Success or Failure
  - Are useful because they work with ‘For’ expressions, can combine threads with futures in the expression

Continued

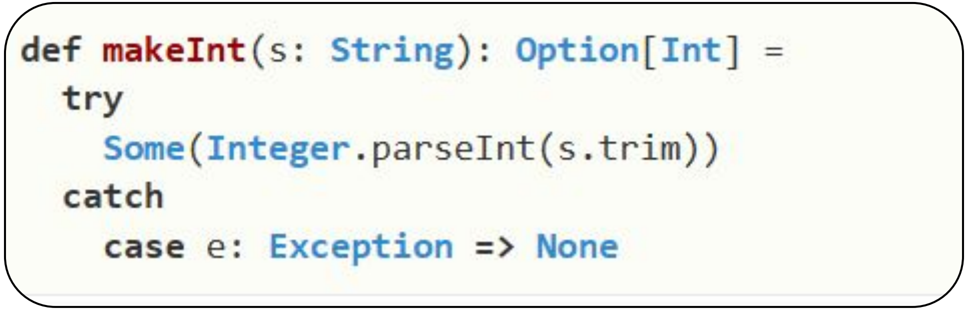
Like Java, threads can be used to run concurrent programs and separate work



```
def longRunningAlgorithm() =
  Thread.sleep(1000)
  42

val eventualInt: Future[Int] = Future {
  longRunningAlgorithm()
}

eventualInt.onComplete {
  case Success(value) => println(s"Result is $value")
  case Failure(e) => println(s"Computation failed: ${e.getMessage}")
}
```

Exception and Event Handling

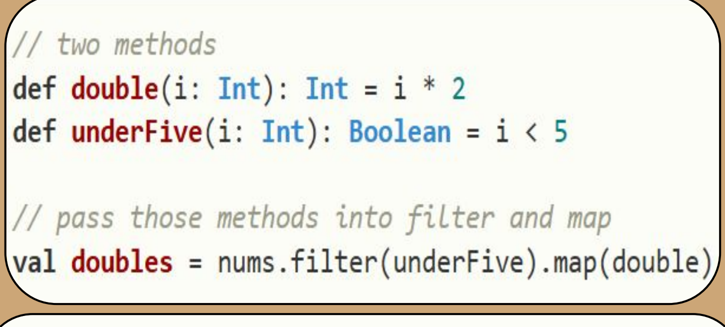
- Scala code is a combination of expressions
  - We use the ‘Option’ type which consists of ‘Some’ and ‘None’, similar to the Java ‘Optional’ class
  - If your code succeeds, then its type is returned inside a Some value
  - If it fails, where it would throw the exception, it now returns a None value
- When working with throw we can still use try/catch/finally blocks to hand the exception
- It is recommended to use the Option method below



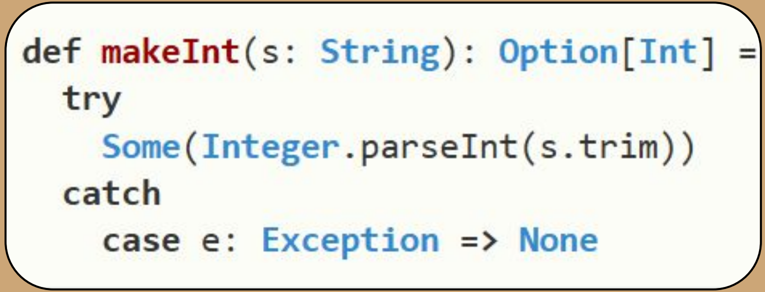
```scala
def makeInt(s: String): Option[Int] =
  try
    Some(Integer.parseInt(s.trim))
  catch
    case e: Exception => None
```

Functional programming

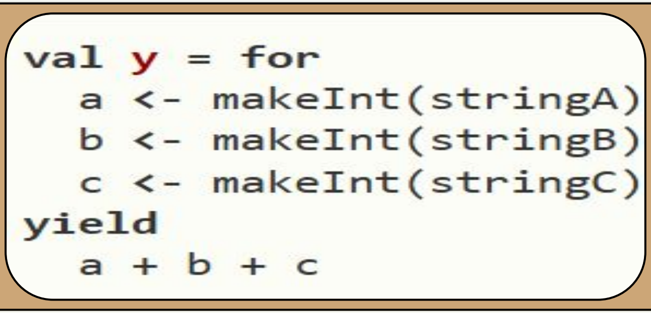
- Immutable values and collections
- Pure functions using method syntax or function syntax
- Functions are values
- Error handling
- Pattern matching



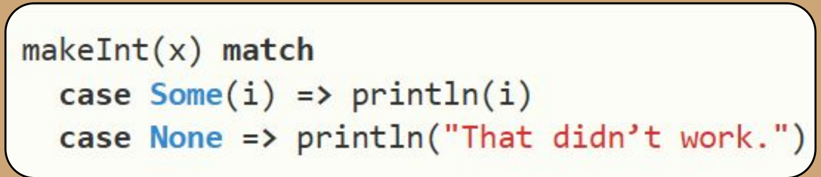
```
// two methods
def double(i: Int): Int = i * 2
def underFive(i: Int): Boolean = i < 5

// pass those methods into filter and map
val doubles = nums.filter(underFive).map(double)
```



```
def makeInt(s: String): Option[Int] =
  try
    Some(Integer.parseInt(s.trim))
  catch
    case e: Exception => None
```



```
val y = for
  a <- makeInt(stringA)
  b <- makeInt(stringB)
  c <- makeInt(stringC)
yield
  a + b + c
```



```
makeInt(x) match
  case Some(i) => println(i)
  case None => println("That didn’t work.")
```

Project proposal

Technology

- Data processing
  - Apache Spark
- Machine learning
  - Smile
  - Vegas
  - Breeze
- Web interface
  - Play Framework
- Data manipulation
  - Spark SQL

Objective

Using student and employee mental health data, create an analytics dashboard to visualize data and make predictions on an individual’s stress levels.

Continued

Features

- Data visualization and relationships between data sets
- Personalized input probability to view mental health in the workforce

Constraints

- Consistency
- Privacy
- Volume of inputs
- User interactions
- Outdated information

Applications

- Personal projections
- Team stress analysis
- Business tasking/burnout reduction

References

Documentation. Scala Documentation. https://docs.scala-lang.org/

Creative Scala: Form and Function

Scala Documentation

An Overview of the Scala Programming Language

Principles and Practice of Programming Languages

https://docs.scala-lang.org/scala3/book/scala-features.html

https://www.scala-lang.org/

https://sysgears.com/articles/how-tech-giants-use-scala/

https://www.scala-lang.org/docu/files/ScalaOverview.pdf

https://www.geeksforgeeks.org/scala/data-types-in-scala/