OBJECT-ORIENTED MEETS FUNCTIONAL

SCALA

Java

```
public class User {
   private String firstName;
   private String lastName;

   public User(String firstName, String lastName) {
     this.firstName = firstName;
     this.lastName  = lastName;
   }
   public String getFirstName() {
     return firstName;
   }
   public void setFirstName(String firstName) {
     this.firstName = firstName;
   }
   public String getLastName() {
     return lastName;
   }
   public void setLastName(String lastName) {
     this.lastName = lastName;
   }

   public String toString() { … }
   public int hashCode() { … }
   public boolean equals(Object obj) { … }
}
```

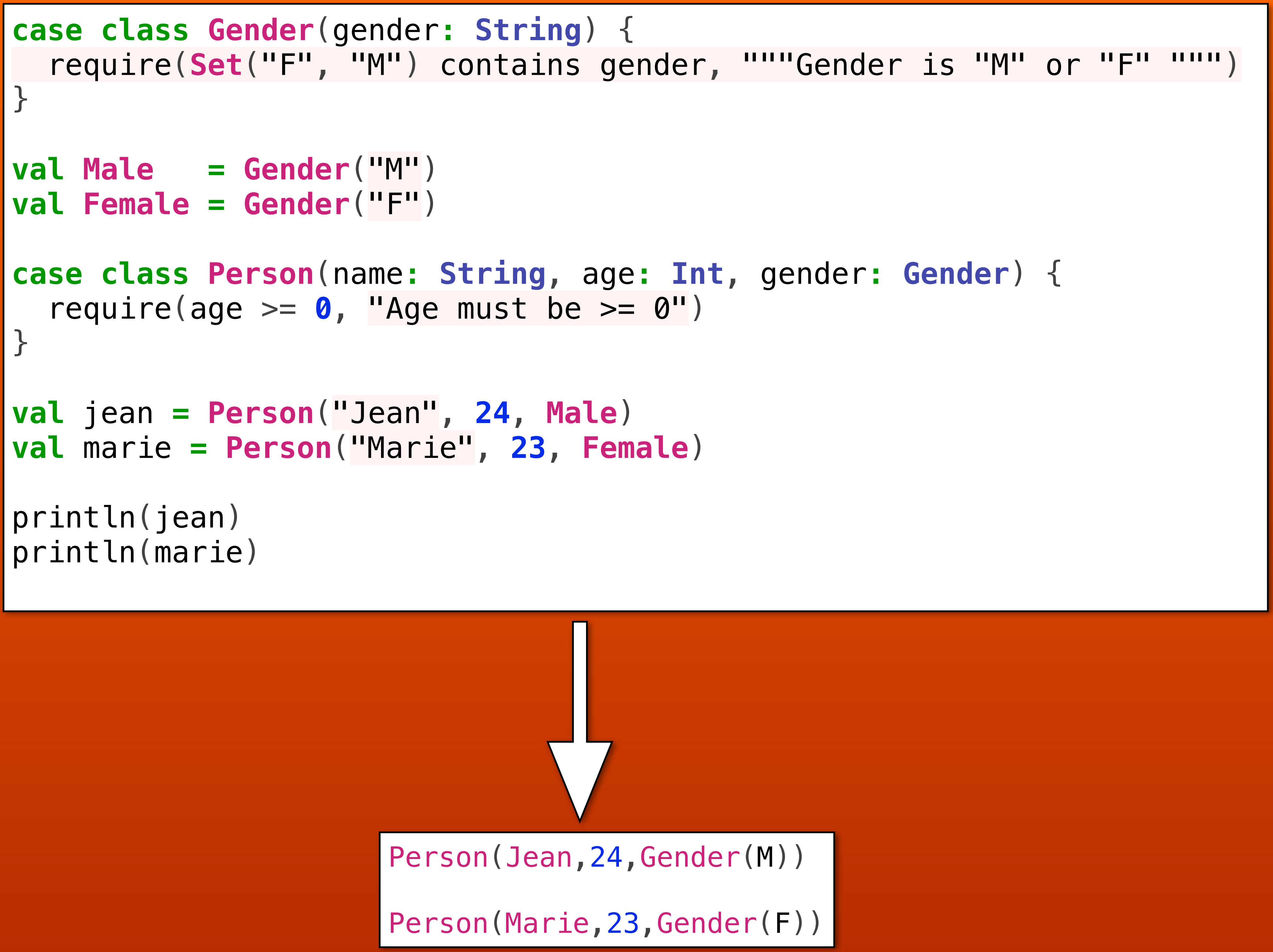
Scala

```
case class User (
    firstname: String,
    lastname: String
)
```

Case Classes

A class definition which includes a number of useful methods generated by the compiler

- toString() - print a nice presentation of case class instance
- getter (and setter, if applicable) methods
- factory method
- equals method from testing structural equality



```scala
case class Gender(gender: String) {
  require(Set("F", "M") contains gender, """Gender is "M" or "F" """)
}

val Male   = Gender("M")
val Female = Gender("F")

case class Person(name: String, age: Int, gender: Gender) {
  require(age >= 0, "Age must be >= 0")
}

val jean = Person("Jean", 24, Male)
val marie = Person("Marie", 23, Female)

println(jean)
println(marie)
```

```
Person(Jean,24,Gender(M))

Person(Marie,23,Gender(F))
```

Case Classes - some examples

Scala Collections

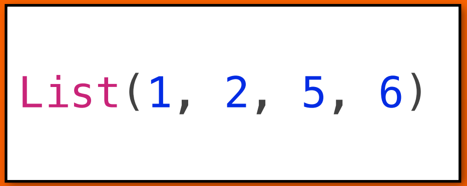
Scala languages has a rich *Collection* library

- Set, List, Vector, Map, …

Combinators are defined on *Collections*

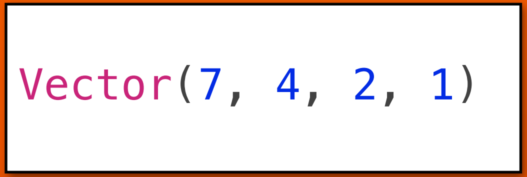
- *map, flatMap, filter, zip, groupBy, min, mean, …*

Scala Collections - Examples



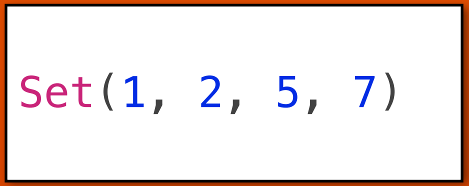
```
List(1, 2, 5, 6)
```

An (ordered) list of Integers - optimized for sequential access



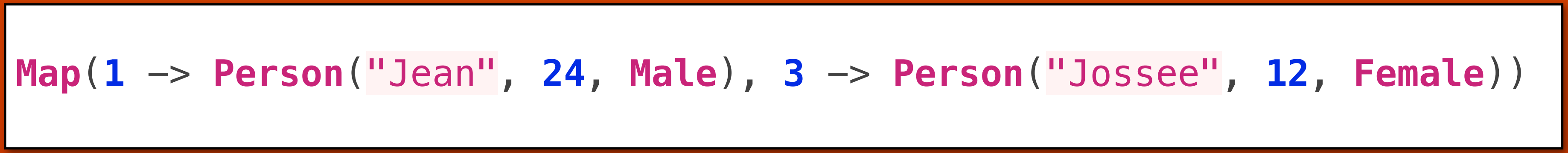
```
Vector(7, 4, 2, 1)
```

An (ordered) vector of Integers - optimized random access



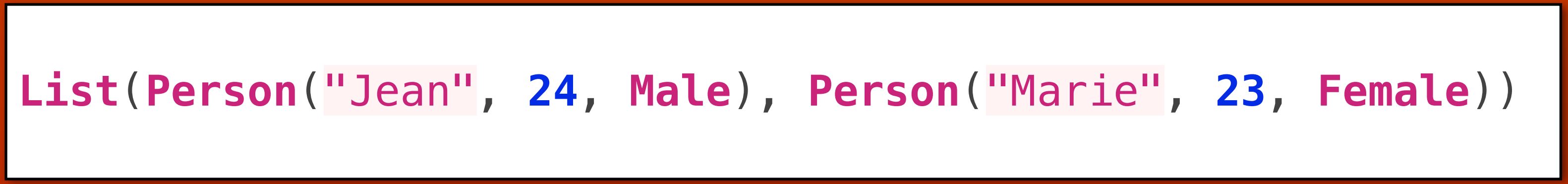
```
Set(1, 2, 5, 7)
```

An (unordered) set of Integers



```
Map(1 -> Person("Jean", 24, Male), 3 -> Person("Jossee", 12, Female))
```

Map of
Integer ➞ Person



```
List(Person("Jean", 24, Male), Person("Marie", 23, Female))
```

A list of ‘Person’
case class instances

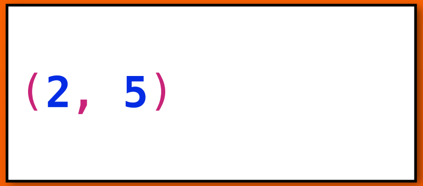
Tuples

An ordered series of 2 or more elements of the same or different type

Syntax: tuple elements separated by a comma and enclosed in parentheses:

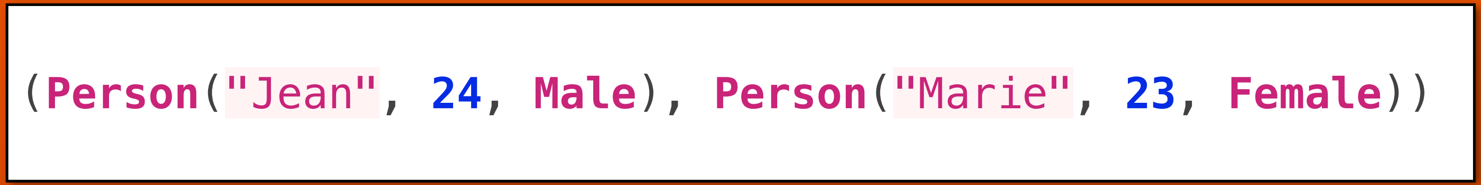
(elem1, elem2, elem3, …)

Tuples - Examples



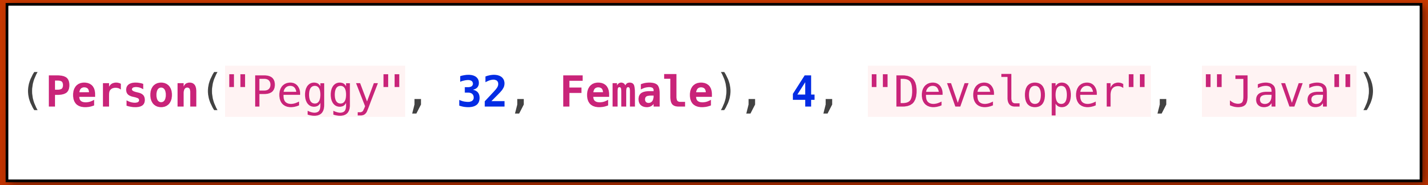
```
(2, 5)
```

A tuple of *two* Integers



```
(Person("Jean", 24, Male), Person("Marie", 23, Female))
```

A tuple of *two* Persons

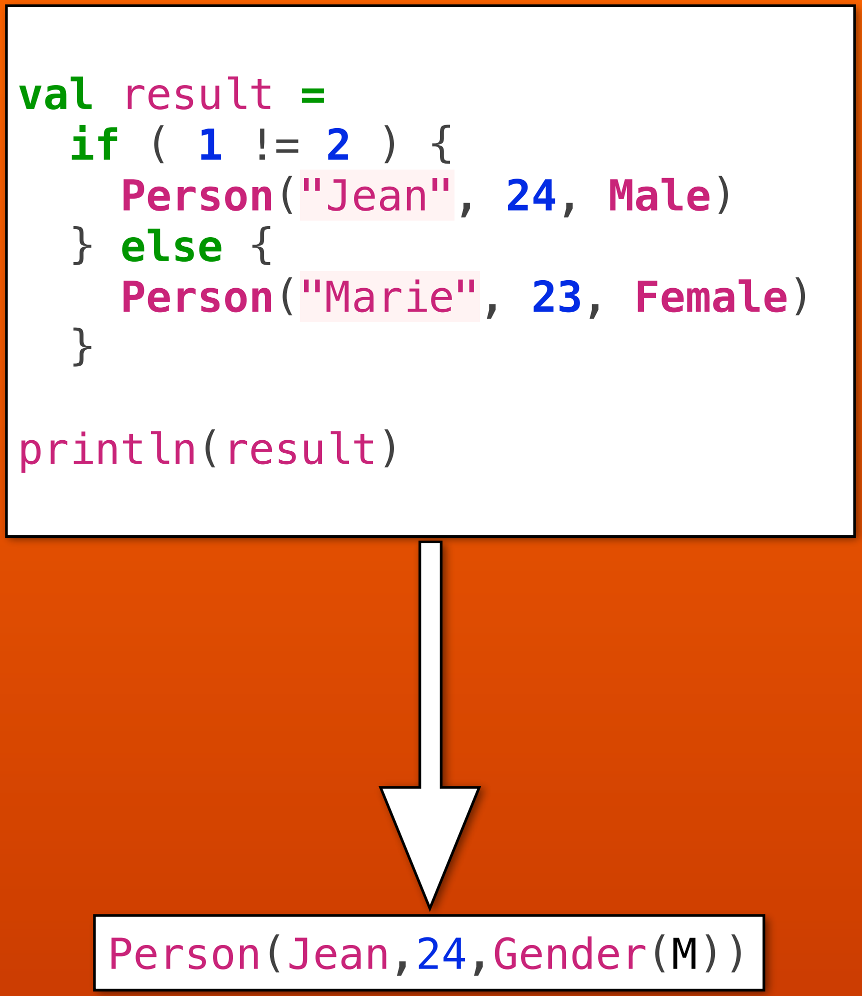


```
(Person("Peggy", 32, Female), 4, "Developer", "Java")
```

A tuple of a Person,
one Integer and
two Strings

Everything is an Expression

- In scala, everything is an expression…
  - No return statement in functions
  - Crucial to composition
- BTW - and completely out of context:
  - no need for semicolons in Scala (unless you really like them…)

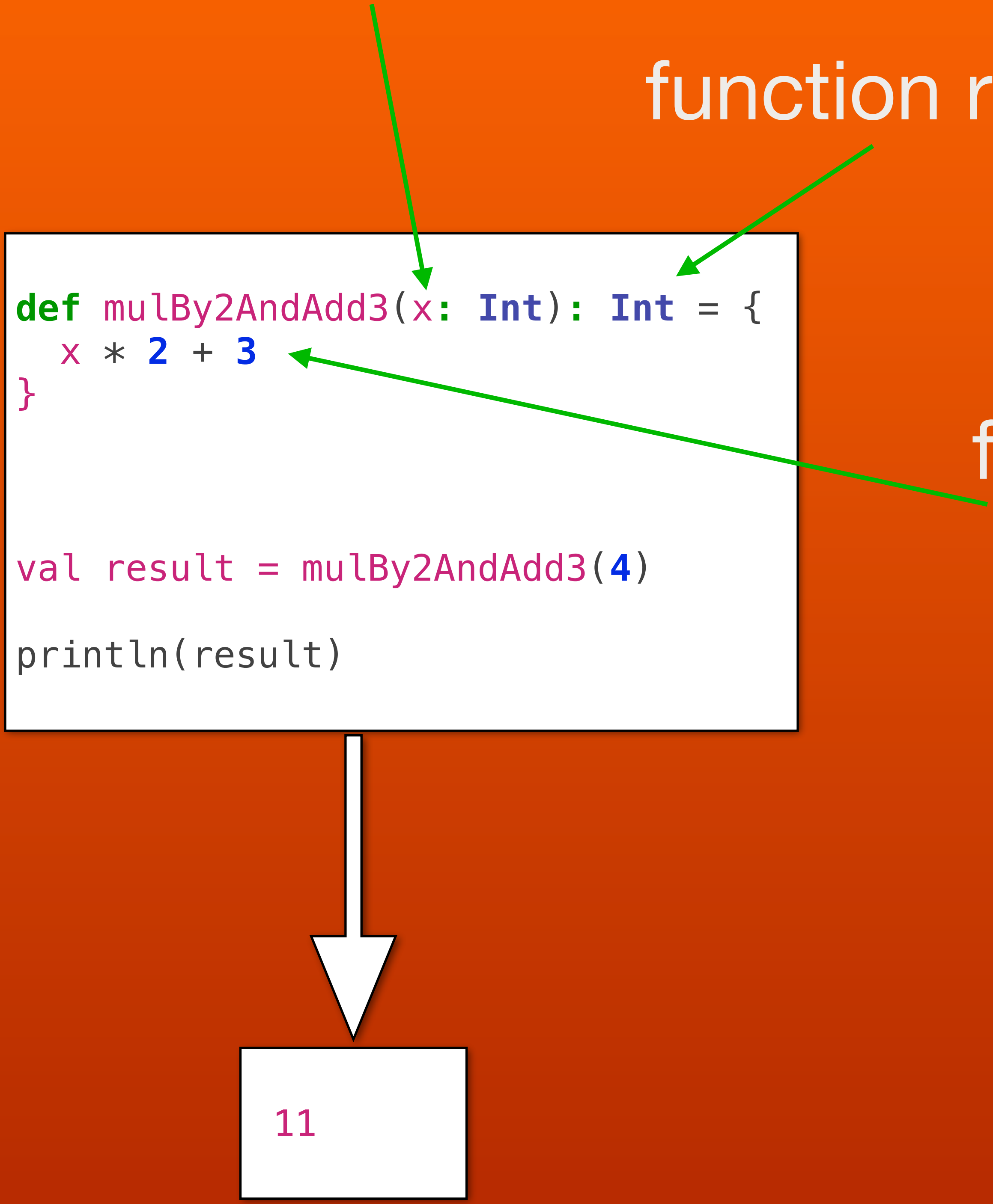


```
val result =
  if ( 1 != 2 ) {
    Person("Jean", 24, Male)
  } else {
    Person("Marie", 23, Female)
  }

println(result)
```

```
Person(Jean,24,Gender(M))
```

Everything is an Expression - Example

(function) definitions - Example



function argument 'x' of type Integer

function return type Integer

function returns value of this expression

```
def mulBy2AndAdd3(x: Int): Int = {
  x * 2 + 3
}



val result = mulBy2AndAdd3(4)

println(result)
```

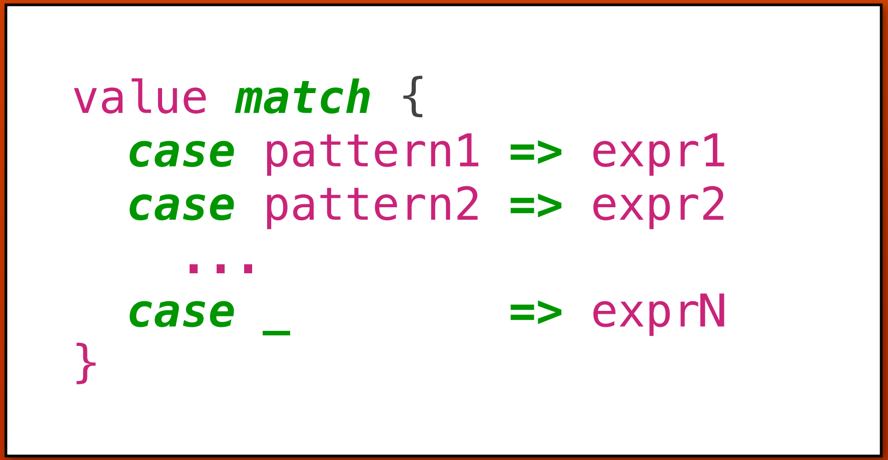
```
11
```

Scala’s ‘switch statement on steroids’

No fall through though

Extremely powerful

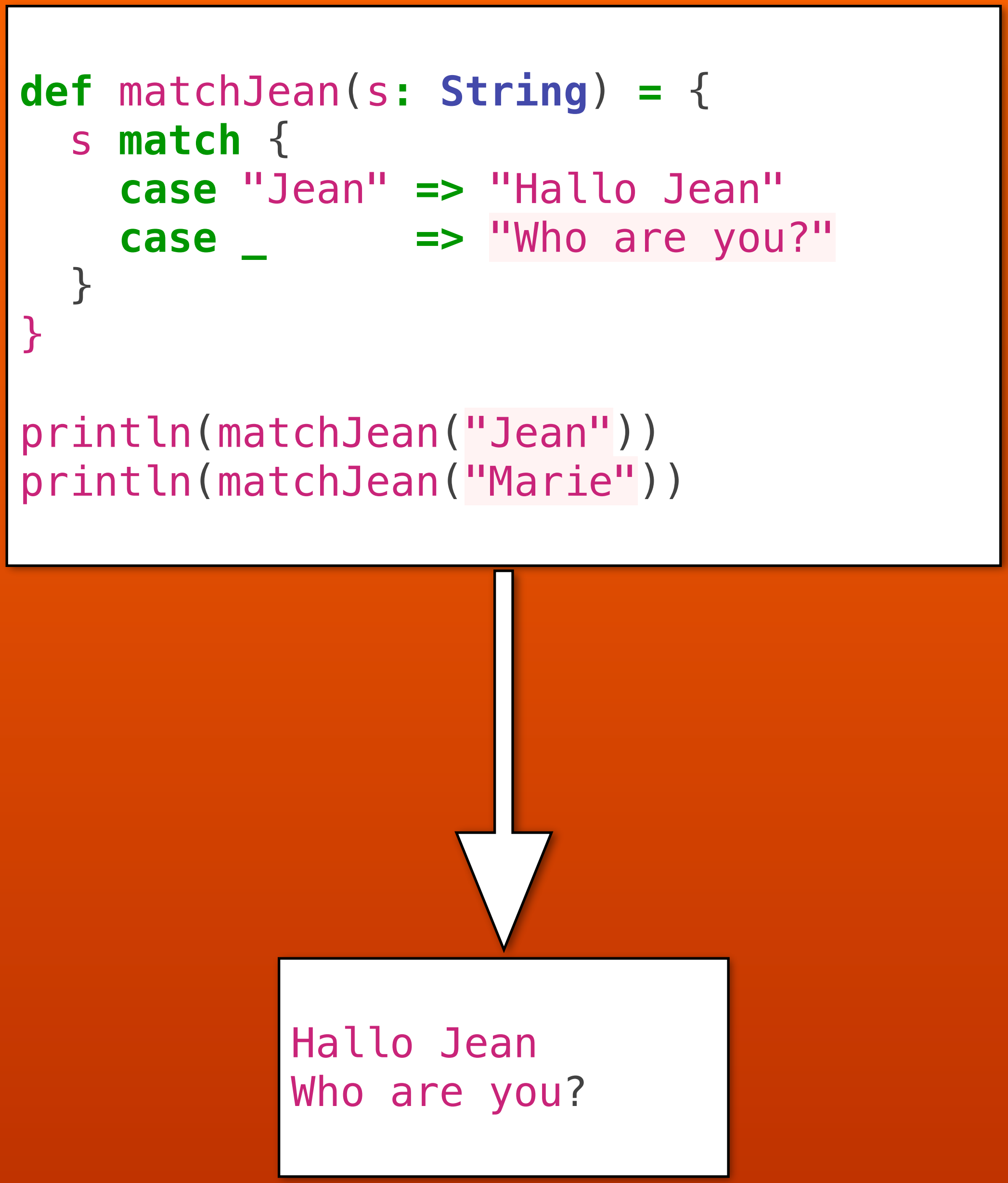
Syntax



```
value match {
  case pattern1 => expr1
  case pattern2 => expr2
    ...
  case _        => exprN
}
```

Pattern Matching

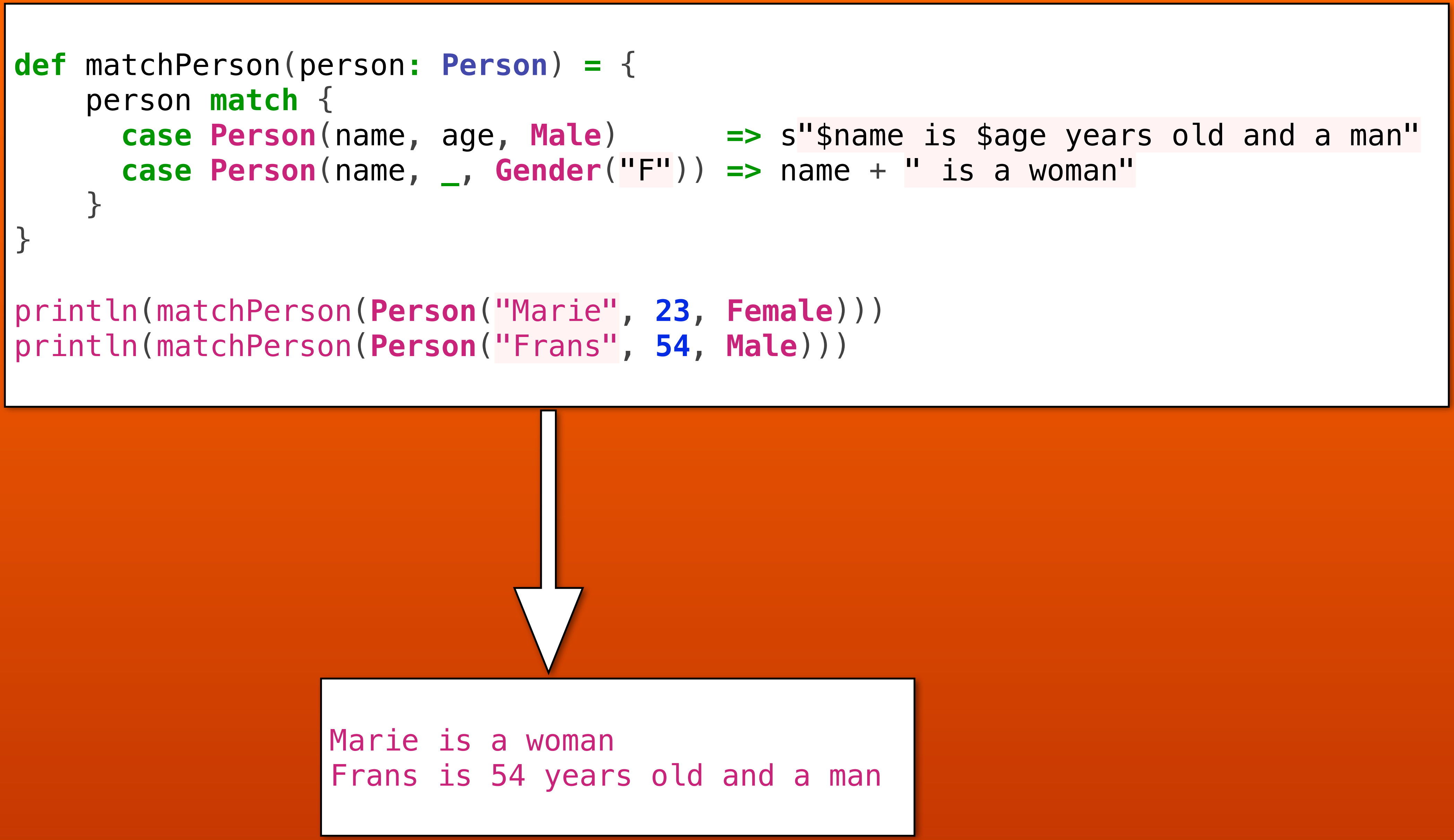
Pattern Matching - Example 1



```
def matchJean(s: String) = {
  s match {
    case "Jean" => "Hallo Jean"
    case _      => "Who are you?"
  }
}

println(matchJean("Jean"))
println(matchJean("Marie"))
```

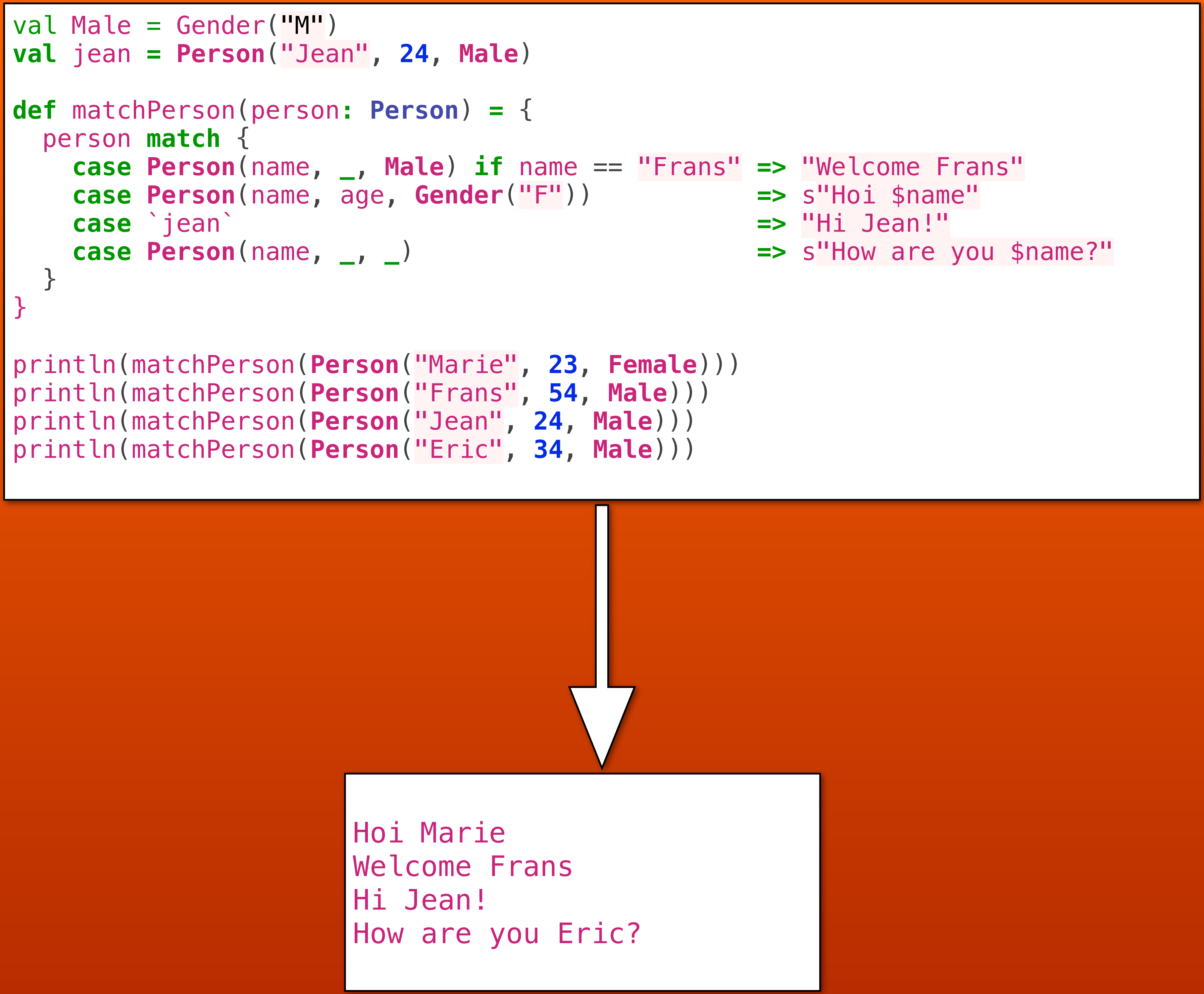
```
Hallo Jean
Who are you?
```

Pattern Matching - Example 2



```
def matchPerson(person: Person) = {
    person match {
      case Person(name, age, Male)      => s"$name is $age years old and a man"
      case Person(name, _, Gender("F")) => name + " is a woman"
    }
}

println(matchPerson(Person("Marie", 23, Female)))
println(matchPerson(Person("Frans", 54, Male)))
```

```
Marie is a woman
Frans is 54 years old and a man
```



```
val Male = Gender("M")
val jean = Person("Jean", 24, Male)

def matchPerson(person: Person) = {
  person match {
    case Person(name, _, Male) if name == "Frans" => "Welcome Frans"
    case Person(name, age, Gender("F"))           => s"Hoi $name"
    case `jean`                                   => "Hi Jean!"
    case Person(name, _, _)                       => s"How are you $name?"
  }
}

println(matchPerson(Person("Marie", 23, Female)))
println(matchPerson(Person("Frans", 54, Male)))
println(matchPerson(Person("Jean", 24, Male)))
println(matchPerson(Person("Eric", 34, Male)))
```

```
Hoi Marie
Welcome Frans
Hi Jean!
How are you Eric?
```

Pattern Matching - Example 3

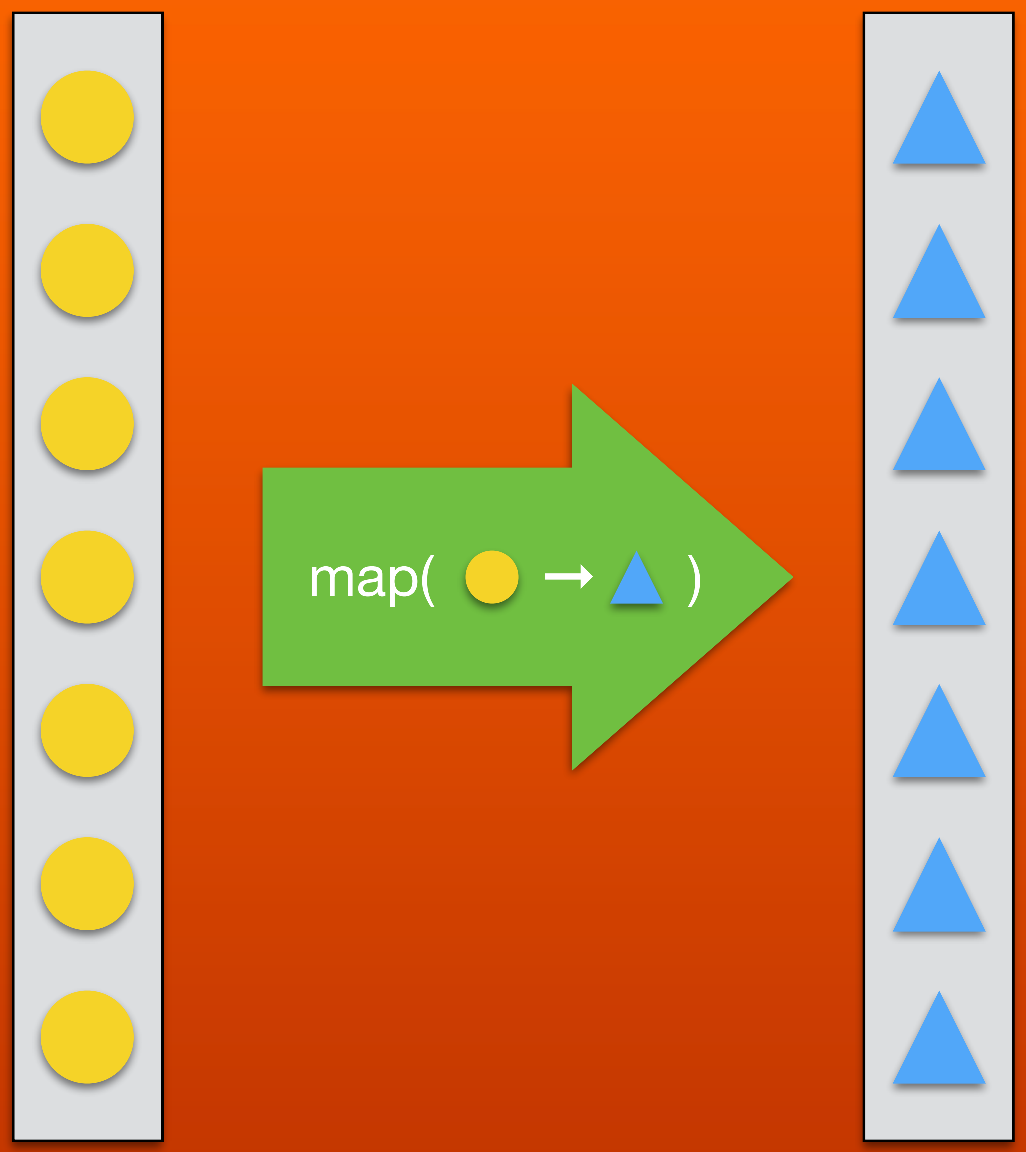
Coming back to Scala collections combinators:

Transform a collection of some type into:

- a collection of the same type
- a collection of a different type
- in general an object of some type…

Let’s have a look at map, flatMap, filter and groupBy

Collection Combinators



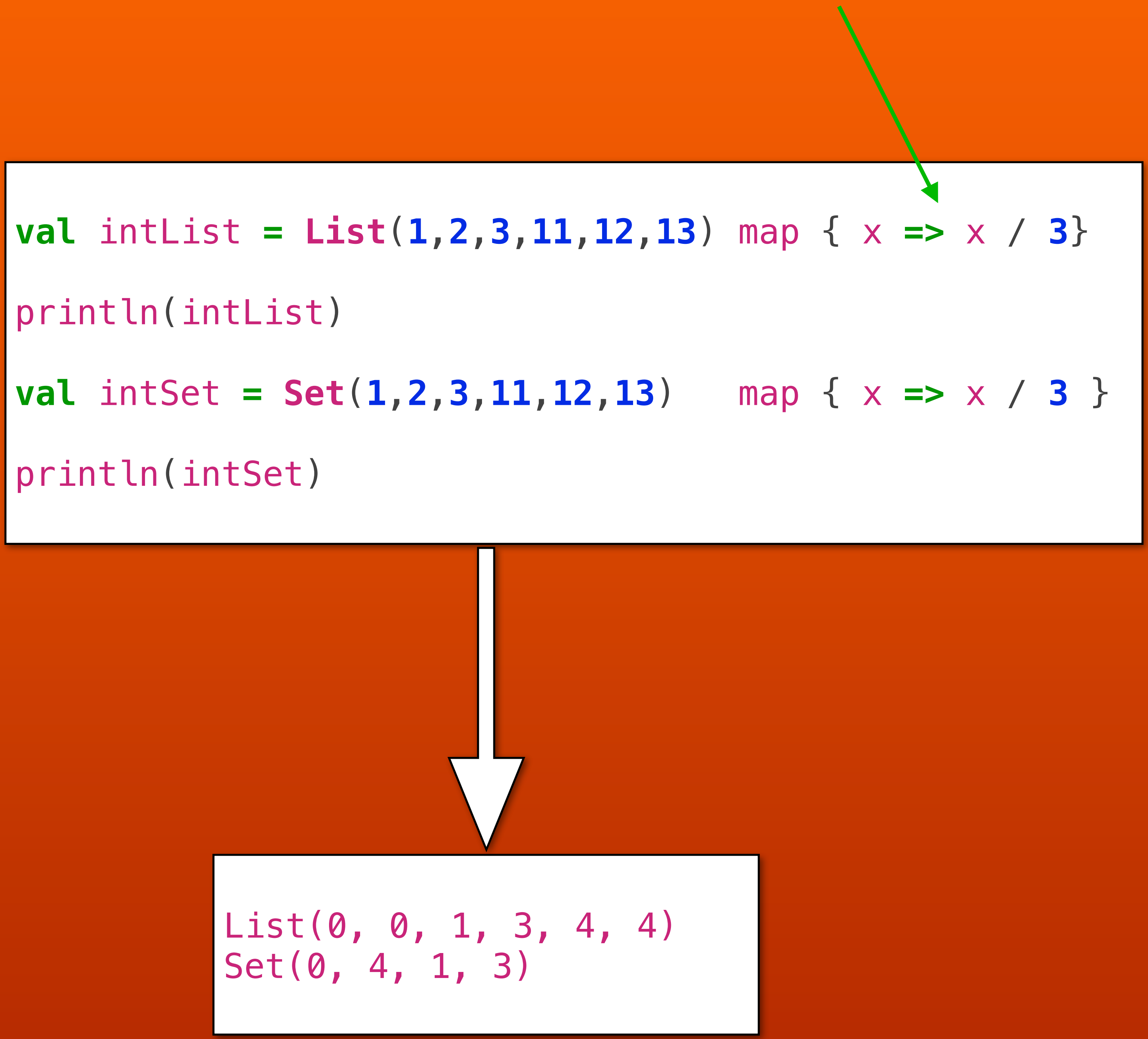
So…

map’s argument is a function that transforms a ● into a ▲

Collection Combinators - map

Collection Combinators - map - Example 1

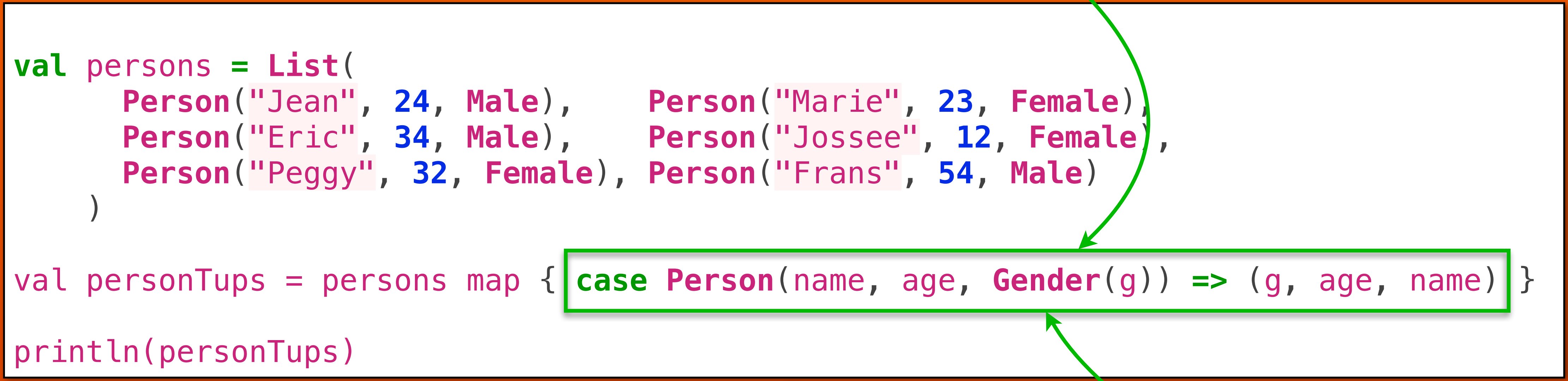
map function (*lambda*) maps Integer → Integer



```
val intList = List(1,2,3,11,12,13) map { x => x / 3}

println(intList)

val intSet = Set(1,2,3,11,12,13)   map { x => x / 3 }

println(intSet)
```

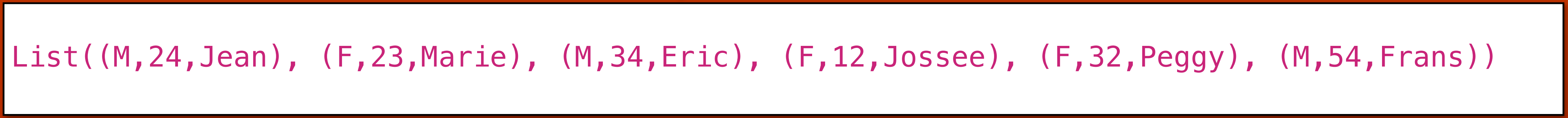
```
List(0, 0, 1, 3, 4, 4)
Set(0, 4, 1, 3)
```

map function (*lambda*) maps
Person → Tuple(String, Int, String)



```
val persons = List(
      Person("Jean", 24, Male),    Person("Marie", 23, Female),
      Person("Eric", 34, Male),    Person("Jossee", 12, Female),
      Person("Peggy", 32, Female), Person("Frans", 54, Male)
    )

val personTups = persons map { case Person(name, age, Gender(g)) => (g, age, name) }

println(personTups)
```

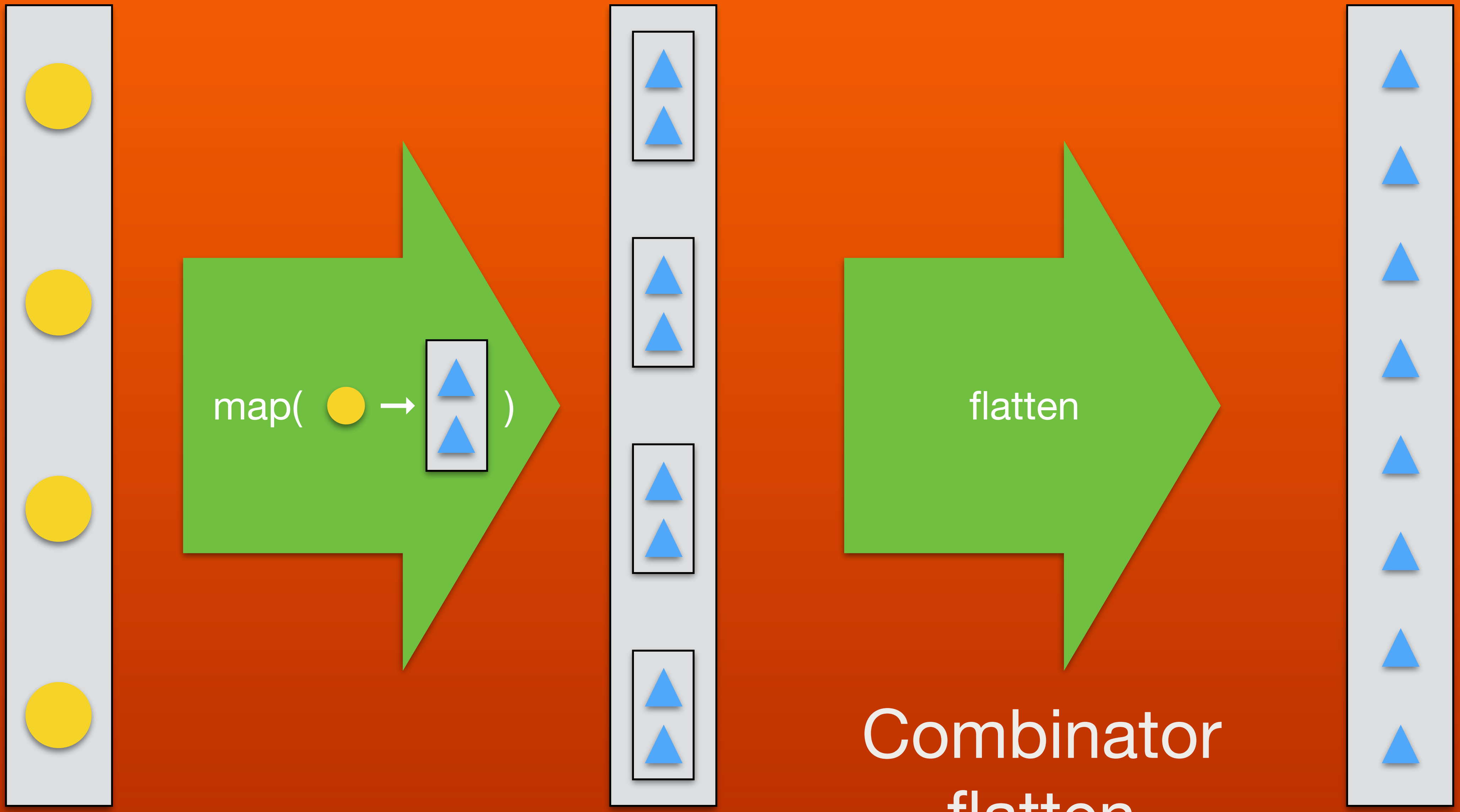
By the way, in Scala,this is what is called a *Partial Function*



```
List((M,24,Jean), (F,23,Marie), (M,34,Eric), (F,12,Jossee), (F,32,Peggy), (M,54,Frans))
```

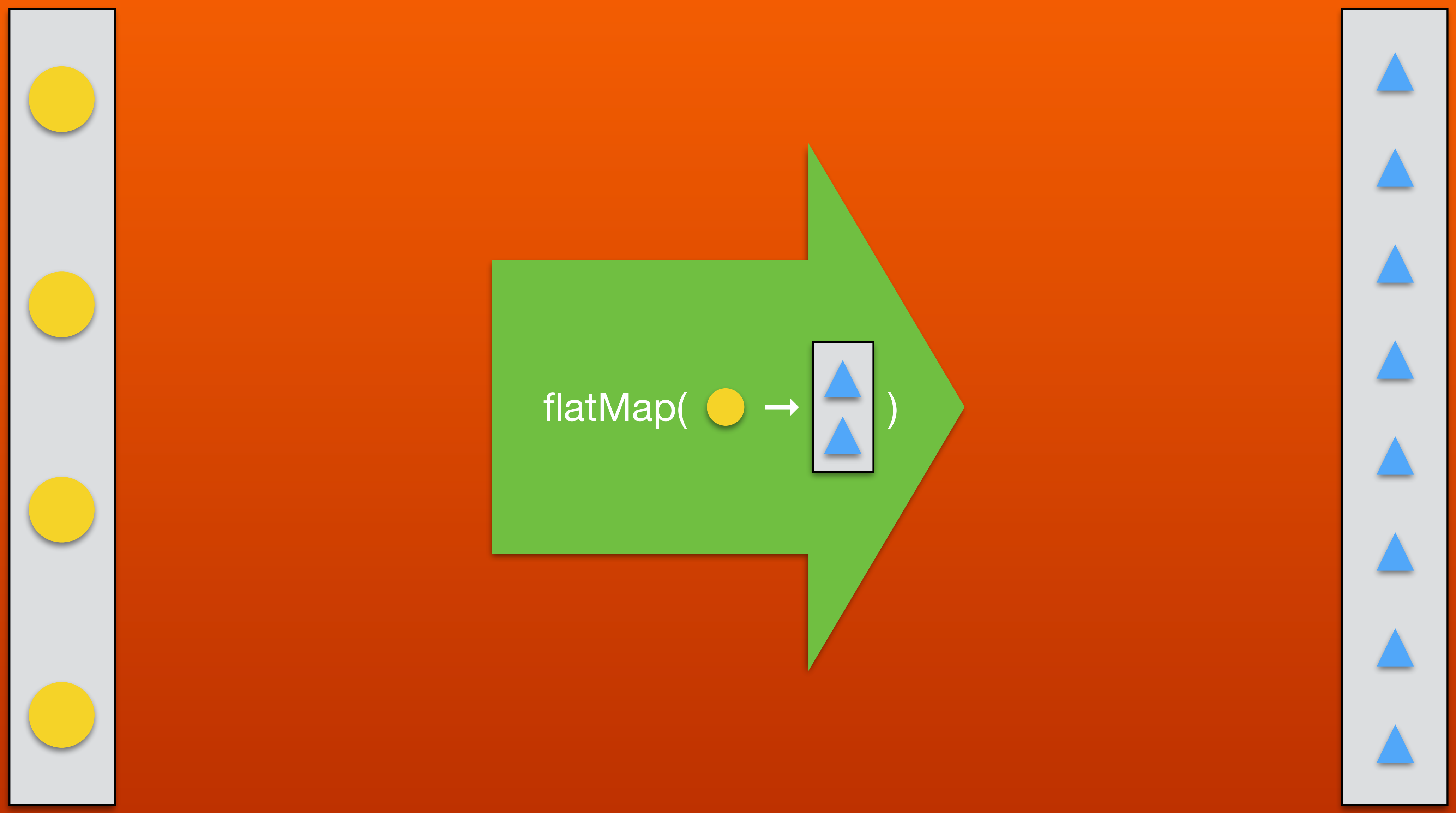
Collection Combinators - map - Example 2

What if map function maps an element into a collection ?



Combinator
flatten
is what
we need

Do this with a single combinator



flatMap(● → ▲ ▲)

Collection Combinators - flatMap

Give a list of Strings, generate a list of all words in all Strings

Split string with sequence of non-word characters as word separator

```
val text = List(
  "Once upon a time in the West,",
  "there was a lonely cowboy"
  )

val words1 = text map     { str => str.split(“""\W+""").toList }
val words2 = text flatMap { str => str.split(“""\W+""").toList }

println(words1)
println(words2)
```

```
List(List(Once, upon, a, time, in, the, West), List(there, was, a, lonely, cowboy))

List(Once, upon, a, time, in, the, West, there, was, a, lonely, cowboy)
```

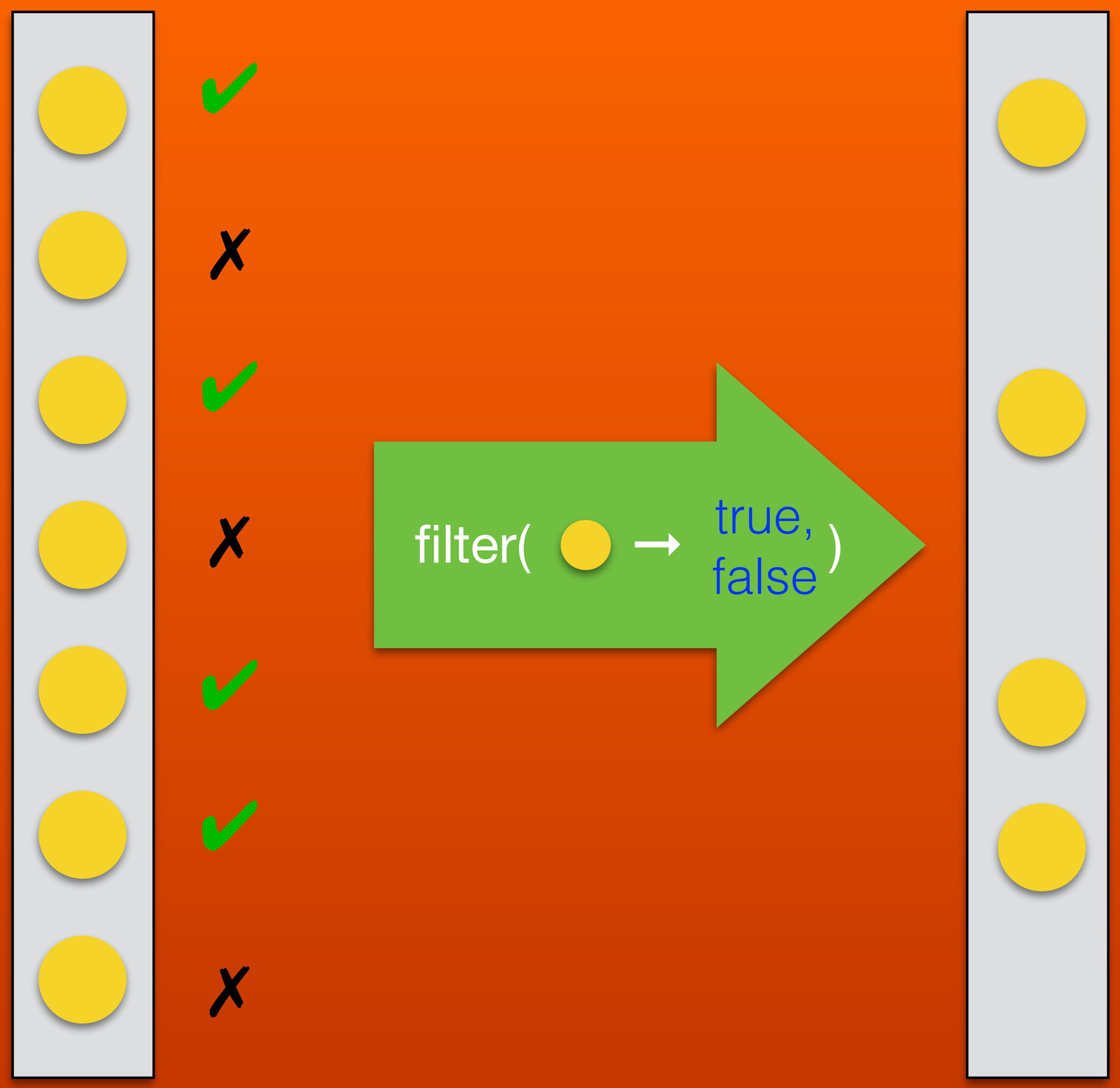
Collection Combinators - flatMap - Example

Filtering of elements in a collection based on a predicate

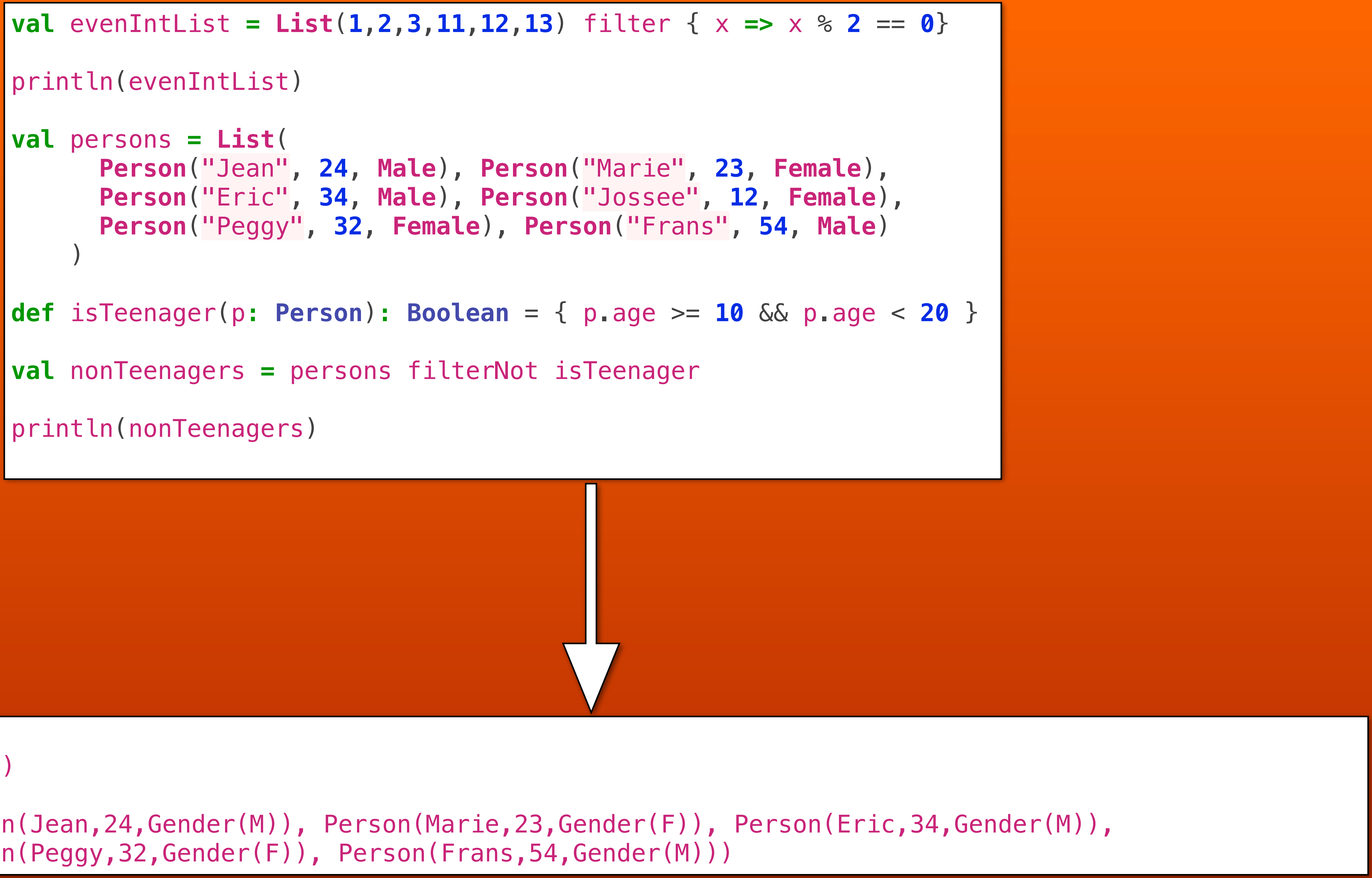
Predicate is a function that maps a collection element values to a true of false

Two forms exist: *filter* & *filterNot*

Collection Combinators - filter



Collection Combinators - filter



```scala
val evenIntList = List(1,2,3,11,12,13) filter { x => x % 2 == 0}

println(evenIntList)

val persons = List(
      Person("Jean", 24, Male), Person("Marie", 23, Female),
      Person("Eric", 34, Male), Person("Jossee", 12, Female),
      Person("Peggy", 32, Female), Person("Frans", 54, Male)
    )

def isTeenager(p: Person): Boolean = { p.age >= 10 && p.age < 20 }

val nonTeenagers = persons filterNot isTeenager

println(nonTeenagers)
```

```
List(2, 12)

List(Person(Jean,24,Gender(M)), Person(Marie,23,Gender(F)), Person(Eric,34,Gender(M)),
     Person(Peggy,32,Gender(F)), Person(Frans,54,Gender(M)))
```

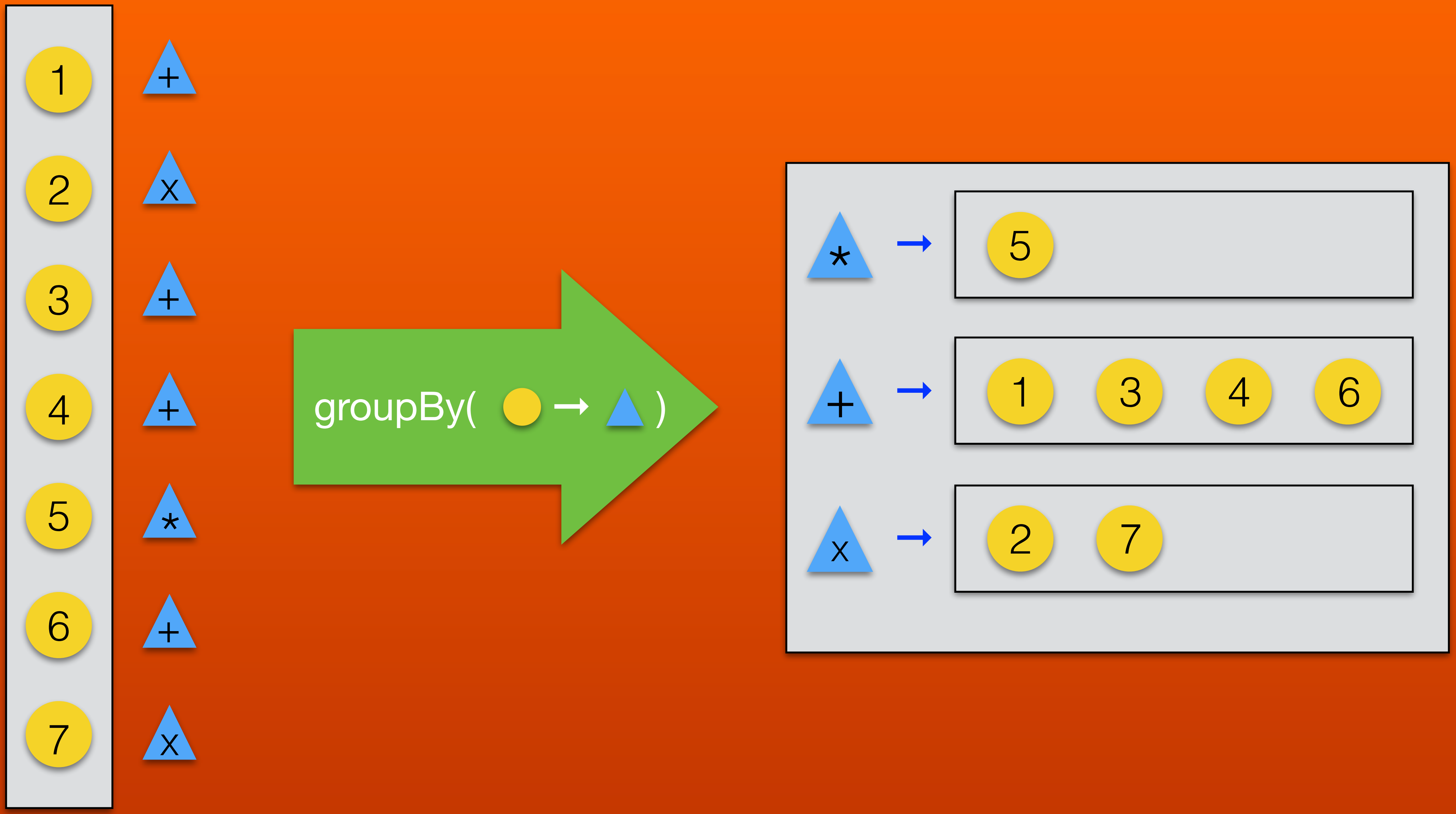
Collection Combinators - filter - Example 1

Groups elements by key

Result is a Map which maps a key to the corresponding elements in the source collection

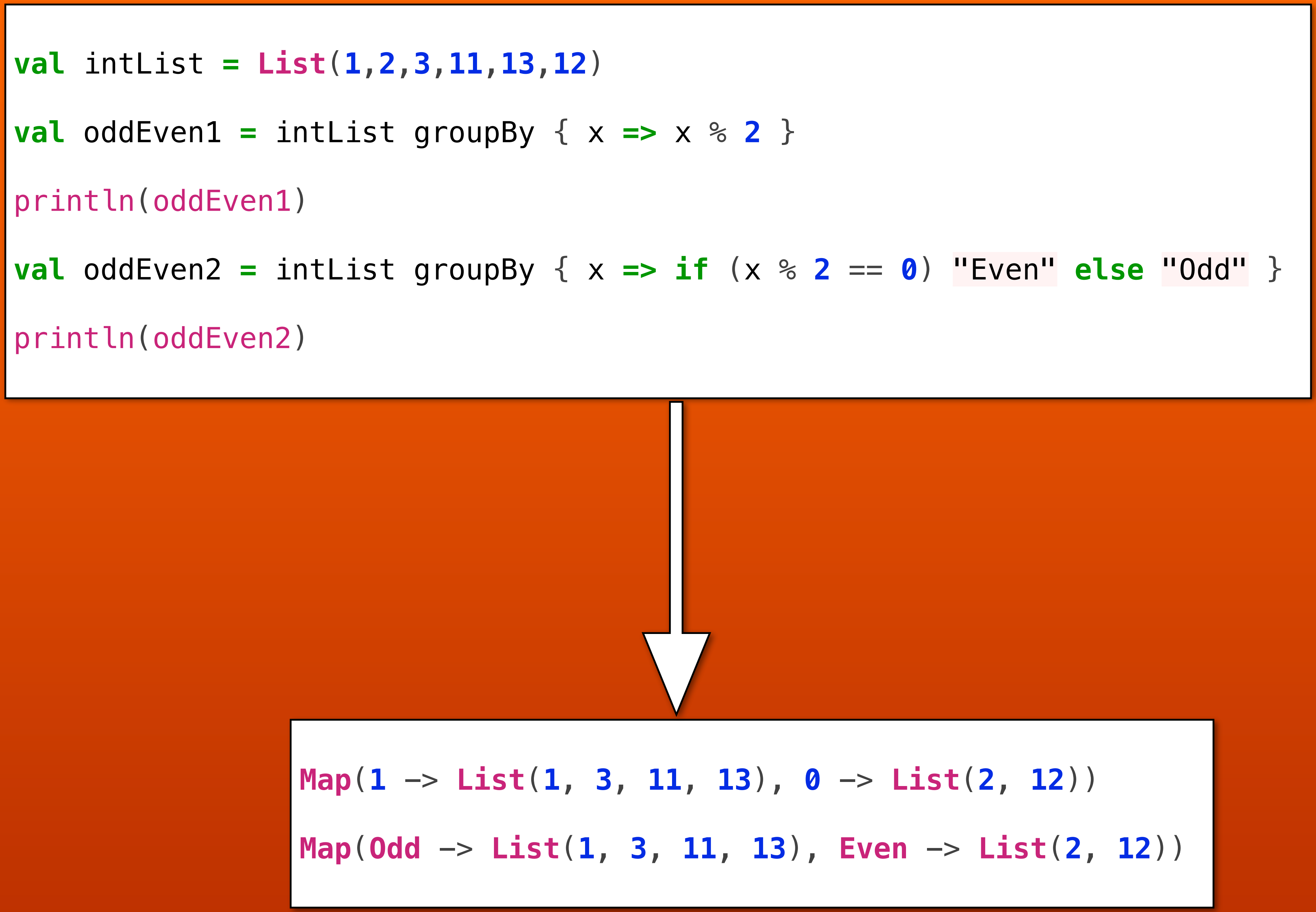
What is needed is a function that computes the key from the elements in the source collection

Collection Combinators - groupBy



groupBy's argument is a function that transforms a value ● into a key ▲

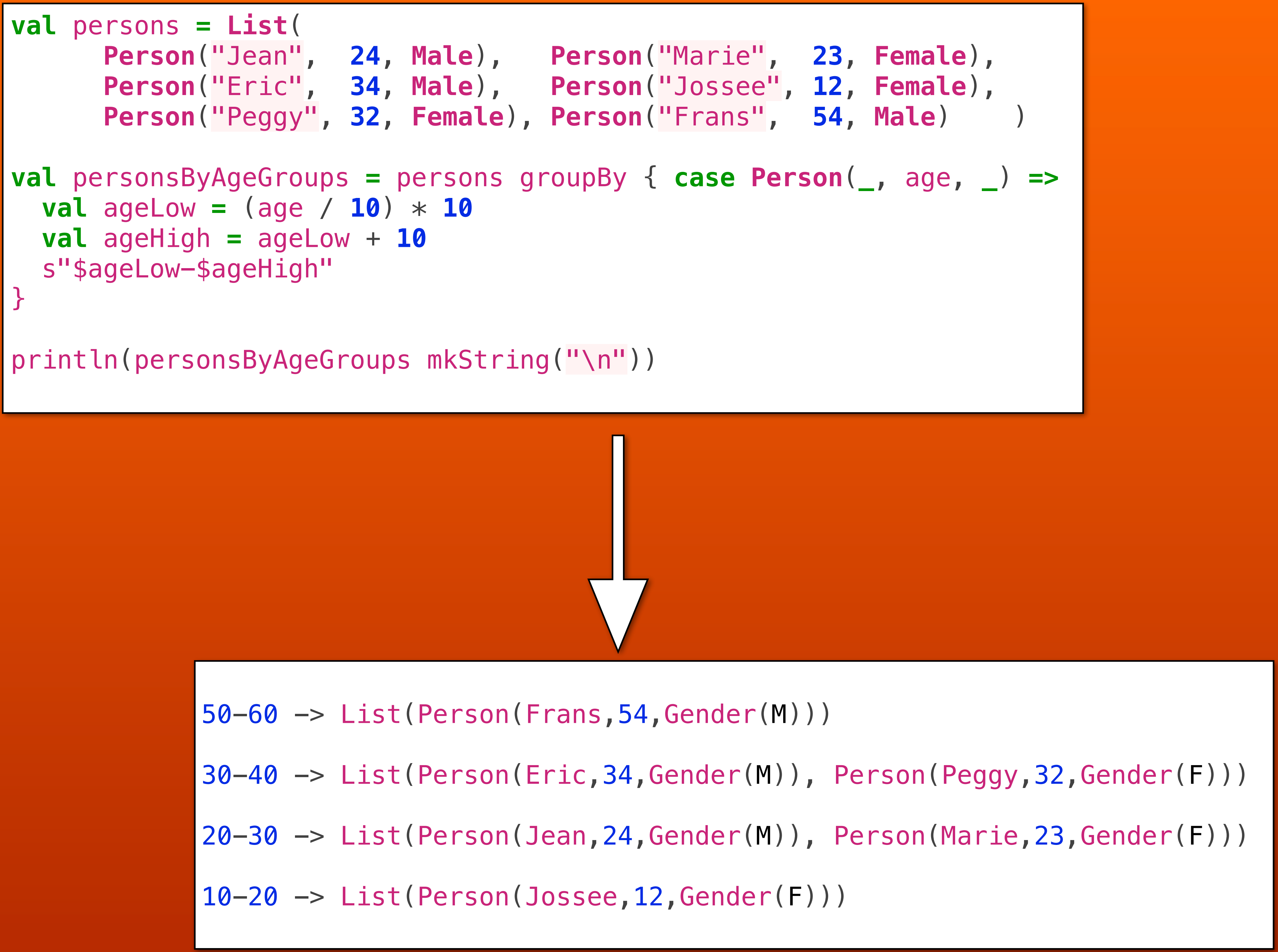
Collection Combinators - groupBy



```scala
val intList = List(1,2,3,11,13,12)

val oddEven1 = intList groupBy { x => x % 2 }

println(oddEven1)

val oddEven2 = intList groupBy { x => if (x % 2 == 0) "Even" else "Odd" }

println(oddEven2)
```

```
Map(1 -> List(1, 3, 11, 13), 0 -> List(2, 12))

Map(Odd -> List(1, 3, 11, 13), Even -> List(2, 12))
```

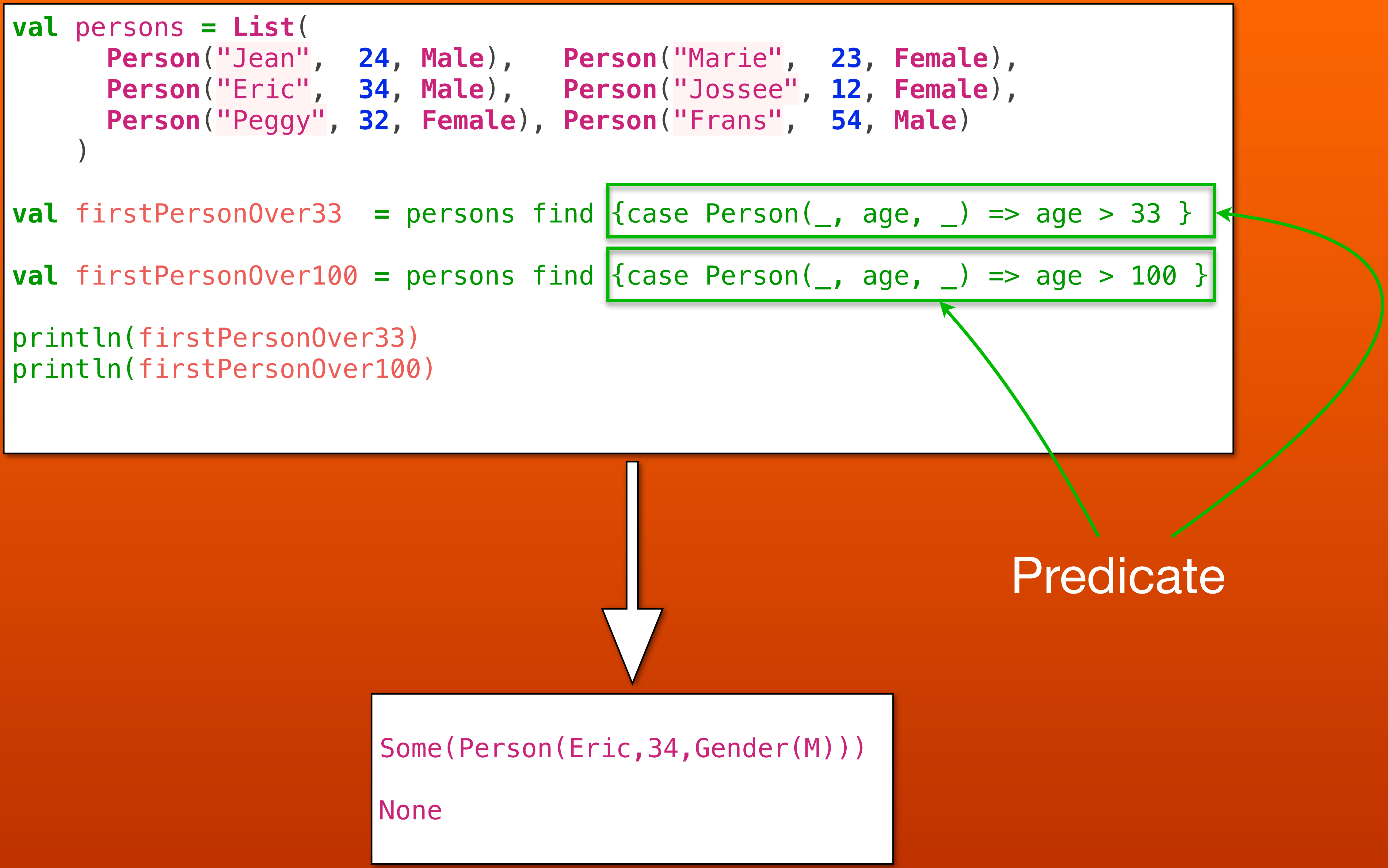
Collection Combinators - groupBy - Example (1)



```scala
val persons = List(
      Person("Jean",  24, Male),   Person("Marie",  23, Female),
      Person("Eric",  34, Male),   Person("Jossee", 12, Female),
      Person("Peggy", 32, Female), Person("Frans",  54, Male)    )

val personsByAgeGroups = persons groupBy { case Person(_, age, _) =>
  val ageLow = (age / 10) * 10
  val ageHigh = ageLow + 10
  s"$ageLow-$ageHigh"
}

println(personsByAgeGroups mkString("\n"))
```

```
50-60 -> List(Person(Frans,54,Gender(M)))

30-40 -> List(Person(Eric,34,Gender(M)), Person(Peggy,32,Gender(F)))

20-30 -> List(Person(Jean,24,Gender(M)), Person(Marie,23,Gender(F)))

10-20 -> List(Person(Jossee,12,Gender(F)))
```

Collection Combinators - groupBy - Example (2)



```
val persons = List(
      Person("Jean",   24, Male),    Person("Marie",   23, Female),
      Person("Eric",   34, Male),    Person("Jossee", 12, Female),
      Person("Peggy", 32, Female), Person("Frans",   54, Male)
    )

val firstPersonOver33  = persons find {case Person(_, age, _) => age > 33 }

val firstPersonOver100 = persons find {case Person(_, age, _) => age > 100 }

println(firstPersonOver33)
println(firstPersonOver100)
```

```
Some(Person(Eric,34,Gender(M)))

None
```

find scans collection *until* it finds an item that meets predicate

Collections - find

Collections - exists

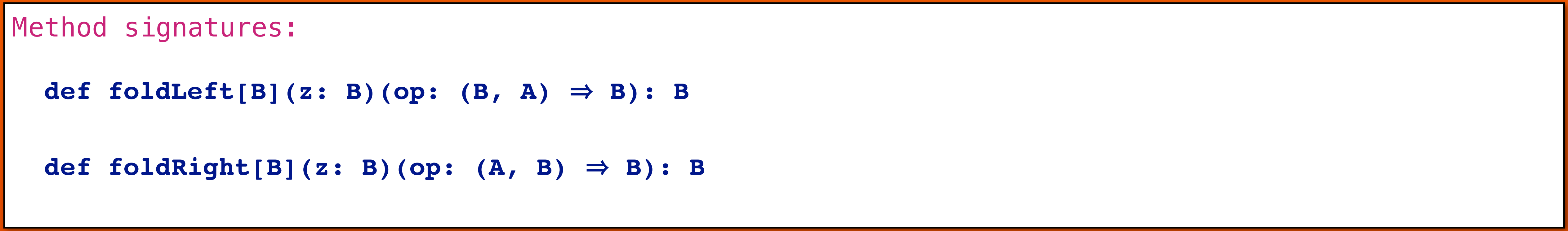
```scala
val persons = List(
      Person("Jean",   24, Male),    Person("Marie",   23, Female),
      Person("Eric",   34, Male),    Person("Jossee", 12, Female),
      Person("Peggy", 32, Female), Person("Frans",   54, Male)
    )

val containsPersonOver33  = persons exists {case Person(_, age, _) => age > 33 }

val containsPersonOver100 = persons exists {case Person(_, age, _) => age > 100 }

println(containsPersonOver33)
println(containsPersonOver100)
```

```
true

false
```

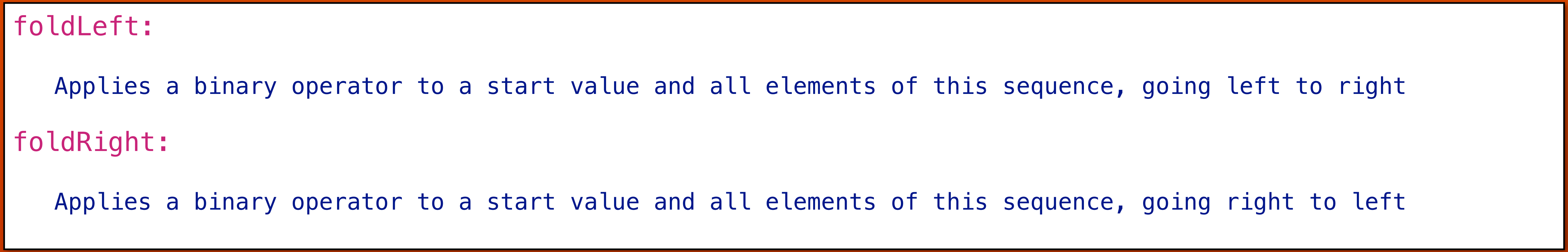
Collections - folding: foldLeft/foldRight

Folding:

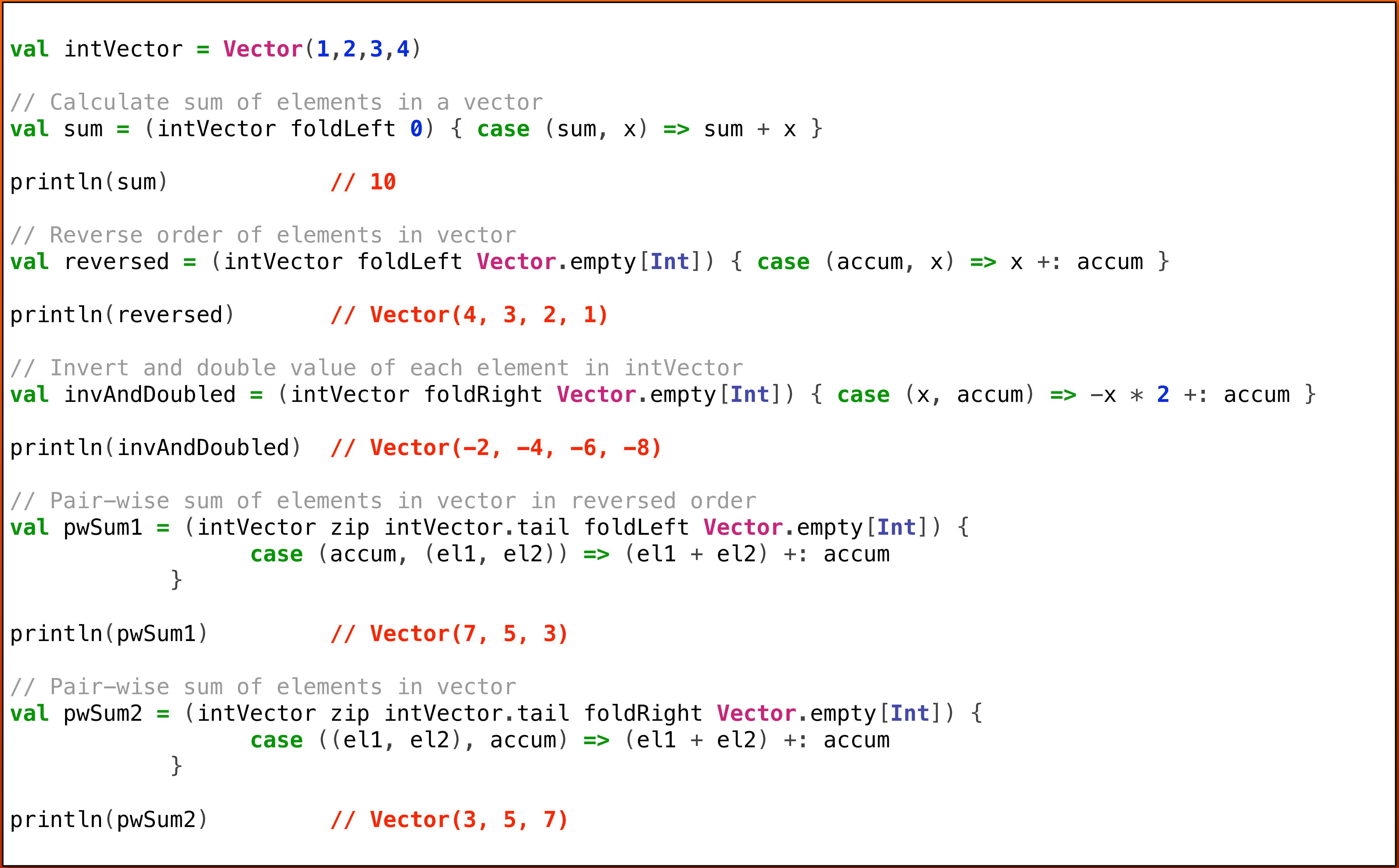
- The Swiss army knife of functional programming
- Combines elements in a collection
- Build a result by accumulating result in an accumulator



```
Method signatures:

  def foldLeft[B](z: B)(op: (B, A) ⇒ B): B

  def foldRight[B](z: B)(op: (A, B) ⇒ B): B
```



```
foldLeft:

   Applies a binary operator to a start value and all elements of this sequence, going left to right

foldRight:

   Applies a binary operator to a start value and all elements of this sequence, going right to left
```



```scala
val intVector = Vector(1,2,3,4)

// Calculate sum of elements in a vector
val sum = (intVector foldLeft 0) { case (sum, x) => sum + x }

println(sum)            // 10

// Reverse order of elements in vector
val reversed = (intVector foldLeft Vector.empty[Int]) { case (accum, x) => x +: accum }

println(reversed)       // Vector(4, 3, 2, 1)

// Invert and double value of each element in intVector
val invAndDoubled = (intVector foldRight Vector.empty[Int]) { case (x, accum) => -x * 2 +: accum }

println(invAndDoubled)  // Vector(-2, -4, -6, -8)

// Pair-wise sum of elements in vector in reversed order
val pwSum1 = (intVector zip intVector.tail foldLeft Vector.empty[Int]) {
                  case (accum, (el1, el2)) => (el1 + el2) +: accum
            }

println(pwSum1)         // Vector(7, 5, 3)

// Pair-wise sum of elements in vector
val pwSum2 = (intVector zip intVector.tail foldRight Vector.empty[Int]) {
                  case ((el1, el2), accum) => (el1 + el2) +: accum
            }

println(pwSum2)         // Vector(3, 5, 7)
```

Collections - folding: foldLeft/foldRight examples

Recursion

- If possible try to write a tail-recursive implementation
- Example below: *fact2* is tail-recursive, *fact1* isn’t

```
def fact1(n: BigInt, acc: BigInt = 1): BigInt = {
  if (n == 1) acc else n * fact1(n - 1)
}

val facts1 = Vector(1 to 10:_*) map (n => fact1(n))

println(facts1)        // Vector(1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800)

println(fact1(20000))  // java.lang.StackOverflowError
```

```
def fact2(n: BigInt, acc: BigInt = 1): BigInt = {
  if (n == 1) acc else fact(n - 1, n * acc)
}

val facts2 = Vector(1 to 10:_*) map (n => fact2(n))

println(facts2)       // Vector(1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800)

println(fact2(20000)) // 181920632023034513482764175686645876607160990147875264891806221863456..........
```

Stream

- Implements a lazy lists where elements are evaluated only when they are needed
- Example: stream of positive, odd integers starting at 1

```scala
def nextInt(n: Long): Stream[Long] = n #:: nextInt(n + 2)

lazy val intsBy2: Stream[Long] = nextInt(1)

println(intsBy2)                             // Stream(1, ?)
println(intsBy2.head)                        // 1
println(intsBy2.tail)                        // Stream(3, ?)
println(intsBy2.take(10).mkString(", “))  // 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
```