Functions

Applications of First-Order Functions

Custom Sorting

- We can sort any type with any comparator
- But what if we want to sort points by their distance from a reference point
  - In general: what if the comparator needs more parameters than just the two elements?
- We can dynamically create a new function with the additional parameters "built-in"

Returning Functions

- We can write a function/method that takes all the needed parameters and returns a function that fits the signature of a comparator

- The distanceComparator method returns a comparator that compares the distance to a reference point

```
def distance(v1: PhysicsVector, v2: PhysicsVector): Double = {
  Math.sqrt(Math.pow(v1.x - v2.x, 2.0) + Math.pow(v1.y - v2.y, 2.0) + Math.pow(v1.z - v2.z, 2.0))
}

def distanceComparator(referencePoint: PhysicsVector): (PhysicsVector, PhysicsVector) => Boolean = {
  (v1: PhysicsVector, v2: PhysicsVector) => {
    distance(v1, referencePoint) < distance(v2, referencePoint)
  }
}
```

Returning Functions

- Use distanceComparator to create a comparator function when needed
- Can create different comparators with different reference points
    - Global state would only allow one comparator at a time

```
val referencePoint = new PhysicsVector(0.5, 0.5, 0.0)
val sortedPoints = MergeSort.mergeSort(points, distanceComparator(referencePoint))
```

```
def distance(v1: PhysicsVector, v2: PhysicsVector): Double = {
  Math.sqrt(Math.pow(v1.x - v2.x, 2.0) + Math.pow(v1.y - v2.y, 2.0) + Math.pow(v1.z - v2.z, 2.0))
}

def distanceComparator(referencePoint: PhysicsVector): (PhysicsVector, PhysicsVector) => Boolean = {
  (v1: PhysicsVector, v2: PhysicsVector) => {
    distance(v1, referencePoint) < distance(v2, referencePoint)
  }
}
```

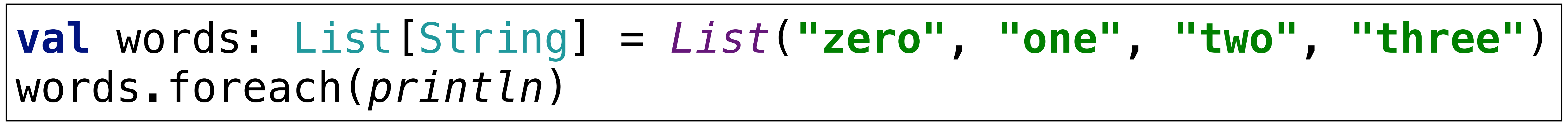
Collection Methods

- We can apply first-order functions to compress our code when working with data structures
  - *and avoid using variables

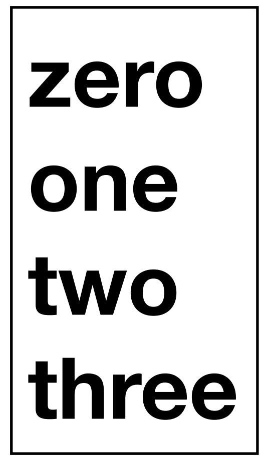
- We'll see a variety of methods that take functions as parameters to help us work with data structures

For Each

- Call a function on each elements of a List
- Only use for the side-effects
    - Ie. Not too useful when embracing immutability (no var)



```scala
val words: List[String] = List("zero", "one", "two", "three")
words.foreach(println)
```



```
zero
one
two
three
```

Filter

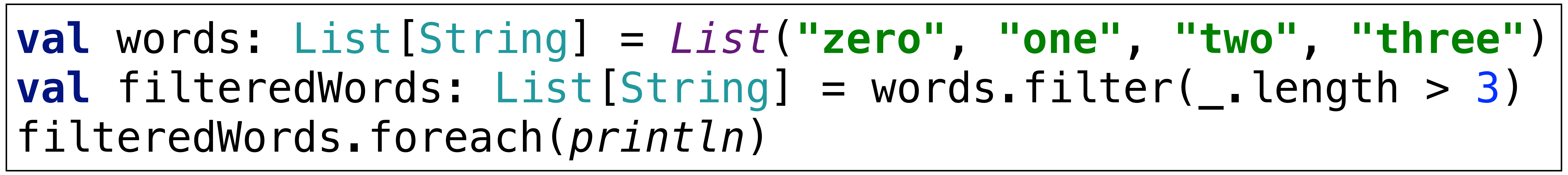
- Takes a function that returns a Boolean
- Returns a new List containing only the elements for which the function returns true

```
val words: List[String] = List("zero", "one", "two", "three")
val filteredWords: List[String] = words.filter(_.length > 3)
filteredWords.foreach(println)
```

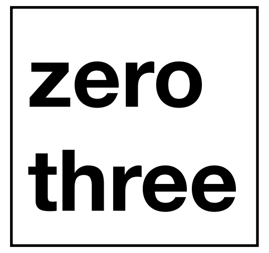
```
zero
three
```

Filter

- The _ shorthand will be used throughout this lecture
- Recall that _ is used in place of the parameter
- No need to create a parameter list
  - "_.length > 3" is equivalent to "(paramName: String) => paramName.length > 3"
- Cannot use this shorthand if a parameter is used more than once!



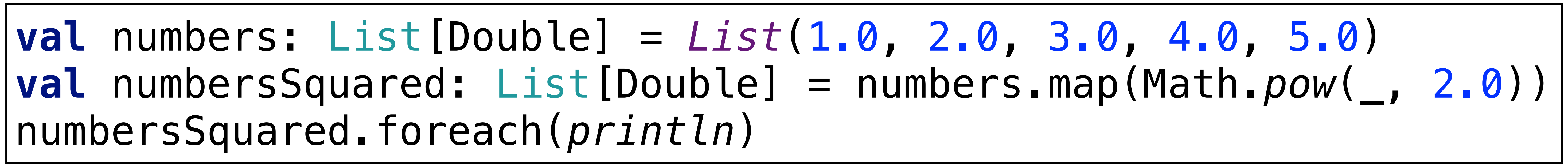
```
val words: List[String] = List("zero", "one", "two", "three")
val filteredWords: List[String] = words.filter(_.length > 3)
filteredWords.foreach(println)
```



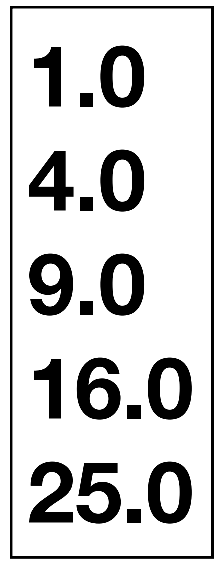
```
zero
three
```

Map

- Takes a function of the data type of the List and returns another data type
- Returns a new List containing the return values of the function with each element as an input



```scala
val numbers: List[Double] = List(1.0, 2.0, 3.0, 4.0, 5.0)
val numbersSquared: List[Double] = numbers.map(Math.pow(_, 2.0))
numbersSquared.foreach(println)
```



```
1.0
4.0
9.0
16.0
25.0
```

Map

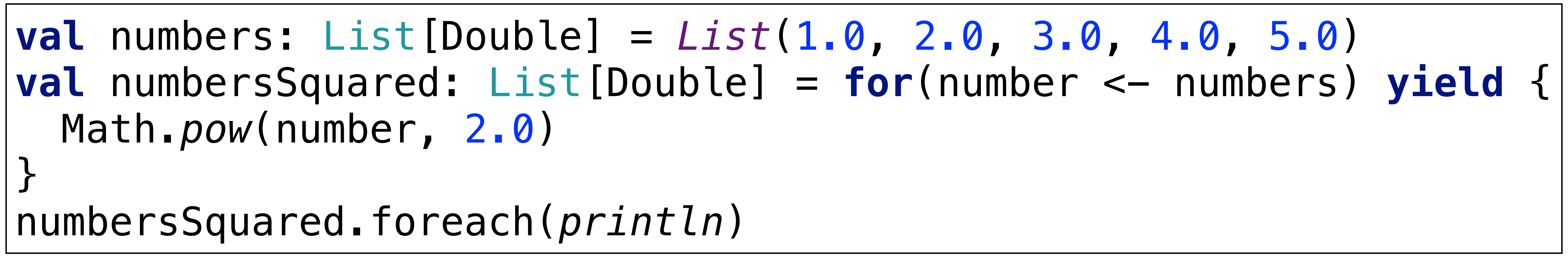
- The map method takes 2 type parameters
- We can provide a function that "maps" the elements to a different type
    - The types can be inferred by the the types of the provided function

```
val words: List[String] = List("zero", "one", "two", "three")
val wordLengths: List[Int] = words.map(_.length)
wordLengths.foreach(println)
```

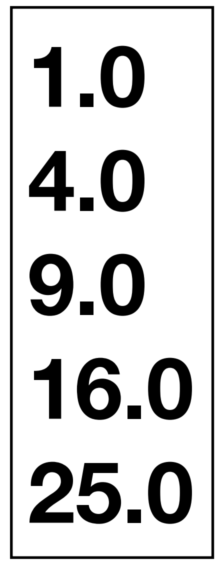
```
4
3
3
5
```

Yield

- As alternate syntax to map, we can use the yield keyword
- Add the keyword yield before the body of a loop
- The last expression of the loop body will be "collected" at each iteration



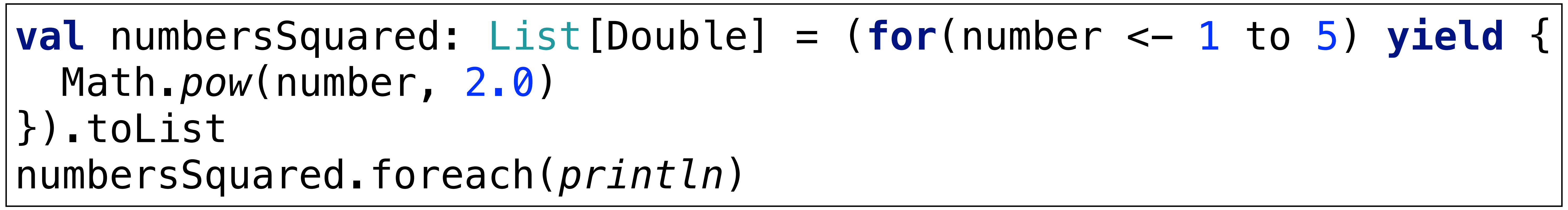
```
val numbers: List[Double] = List(1.0, 2.0, 3.0, 4.0, 5.0)
val numbersSquared: List[Double] = for(number <- numbers) yield {
  Math.pow(number, 2.0)
}
numbersSquared.foreach(println)
```



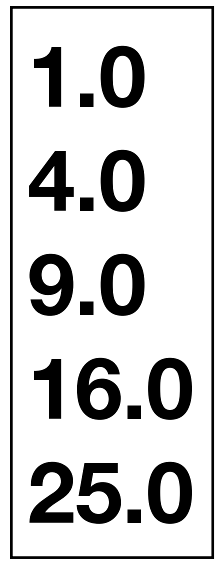
```
1.0
4.0
9.0
16.0
25.0
```

Yield

- Using yield will create a data structure of the same type as the one being iterated over
- It's not always possible to match the type exactly
- Scala will default to a certain data structure
    - Use toList to convert the default type to a List



```scala
val numbersSquared: List[Double] = (for(number <- 1 to 5) yield {
  Math.pow(number, 2.0)
}).toList
numbersSquared.foreach(println)
```



```
1.0
4.0
9.0
16.0
25.0
```

Reduce

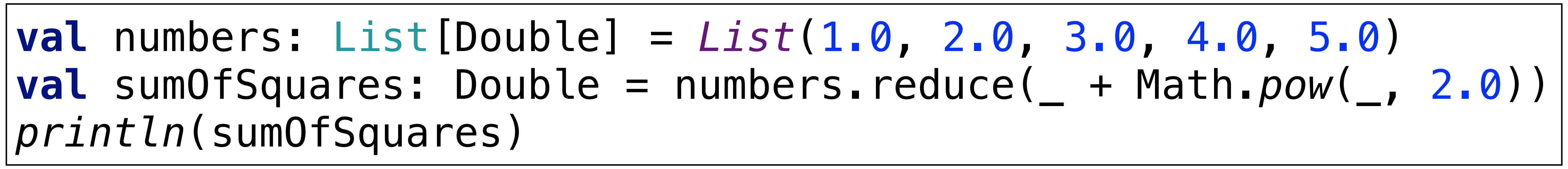
- Takes a function that combines two values of the data type into a single value of that type
- Calls this function on all elements
  - Combines the data into a single value
- The first parameter of the function is the accumulator
  - Stores the total value accumulated so far
  - Initialized as the first element (Note: This example breaks if 1.0 is not the first element

```scala
val numbers: List[Double] = List(1.0, 2.0, 3.0, 4.0, 5.0)
val sumSquares: (Double, Double) => Double = (acc: Double, b: Double) => acc + Math.pow(b, 2.0)
val sumOfSquares: Double = numbers.reduce(sumSquares)
println(sumOfSquares)
```

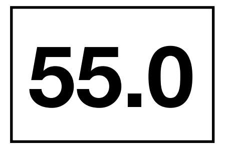
```
55.0
```

Reduce

- We can use the _ shorthand with two parameters
  - The order of appearance of the _'s is the parameter order
- Cannot use _ shorthand if you need to use an input twice



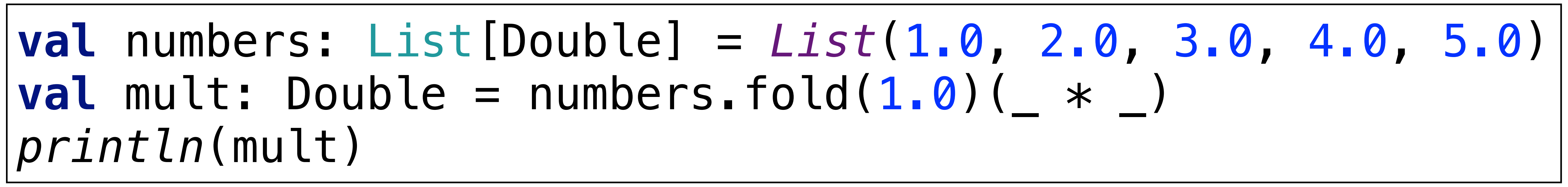
```
val numbers: List[Double] = List(1.0, 2.0, 3.0, 4.0, 5.0)
val sumOfSquares: Double = numbers.reduce(_ + Math.pow(_, 2.0))
println(sumOfSquares)
```



```
55.0
```

Fold

- Similar to reduce
- Use fold if you need to initialize your accumulator
- Use fold if your list might be empty (Return value is the initialized value)
    - Reduce will throw an error if the input list is empty

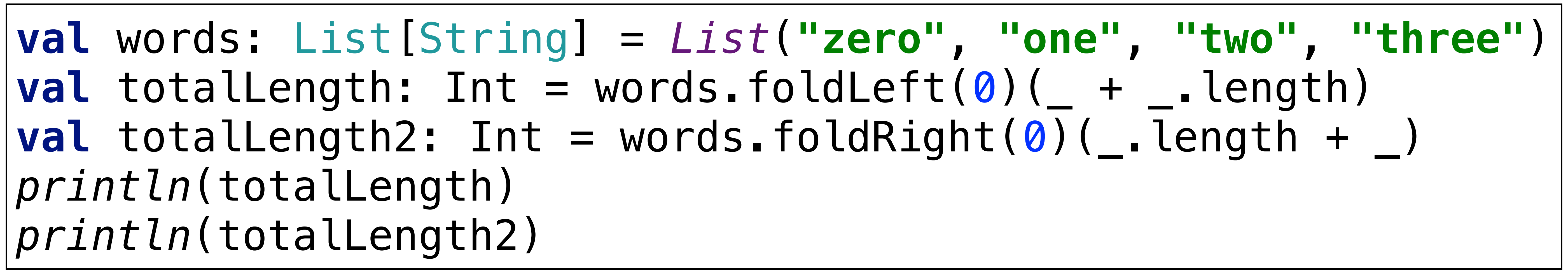


```
val numbers: List[Double] = List(1.0, 2.0, 3.0, 4.0, 5.0)
val mult: Double = numbers.fold(1.0)(_ * _)
println(mult)
```

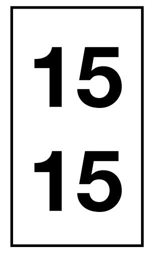
```
120.0
```

Fold

- To accumulate to a type different than the data type
  - Use the left/right version of fold (or reduce)
- Initial value and function return type determine the accumulator type



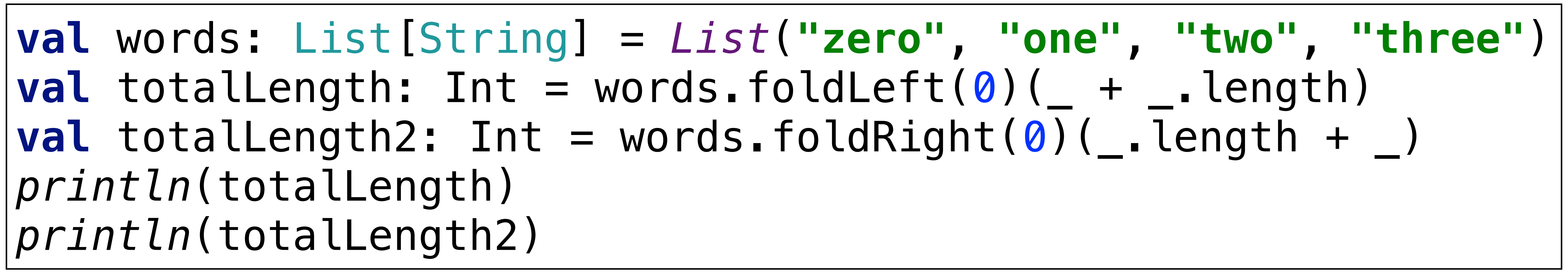
```
val words: List[String] = List("zero", "one", "two", "three")
val totalLength: Int = words.foldLeft(0)(_ + _.length)
val totalLength2: Int = words.foldRight(0)(_.length + _)
println(totalLength)
println(totalLength2)
```



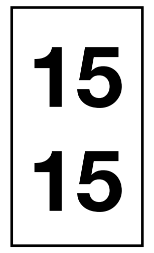
```
15
15
```

Fold

- Using fold/reduce defaults to foldLeft/reduceLeft
  - Start with the first (left-most) element
- To accumulate from the end of the List use foldRight/reduceRight
  - Must reverse the parameter order when using foldRight/reduceRight
    - Accumulator is second parameter, data is first element



```
val words: List[String] = List("zero", "one", "two", "three")
val totalLength: Int = words.foldLeft(0)(_ + _.length)
val totalLength2: Int = words.foldRight(0)(_.length + _)
println(totalLength)
println(totalLength2)
```



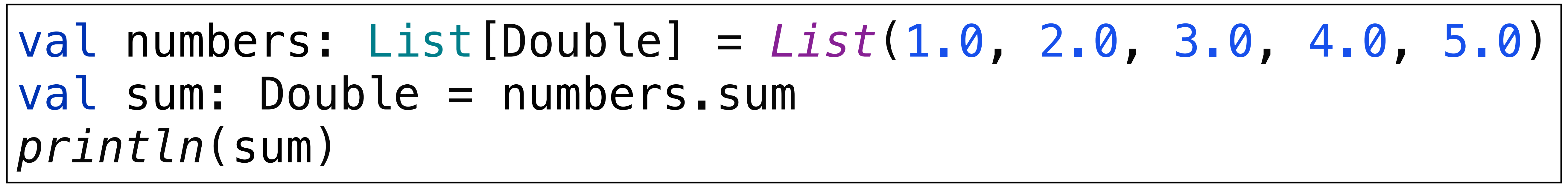
```
15
15
```

Even More List Methods

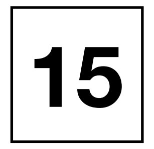
- The next few methods do not use first-order functions, but you may find them useful as you write code without variables

Sum

- Returns the sum of all the elements in the list
- Can only be used with lists of numbers
- Shorthand for calling reduce with the addition function



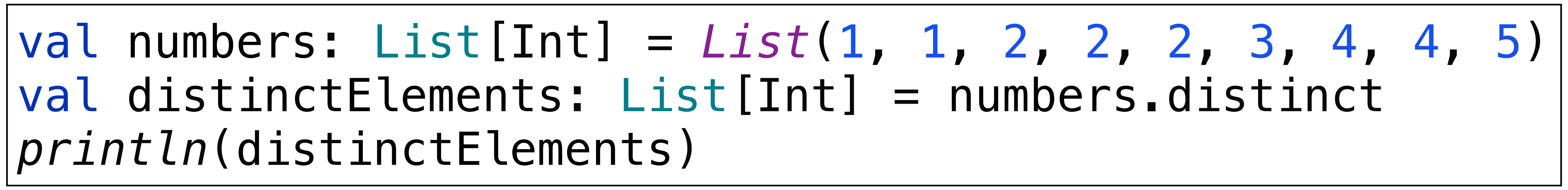
```
val numbers: List[Double] = List(1.0, 2.0, 3.0, 4.0, 5.0)
val sum: Double = numbers.sum
println(sum)
```



```
15
```

Distinct

- Removes all duplicate values from a list
- Returns a new list containing only the distinct elements
- Useful for detecting and removing duplicates

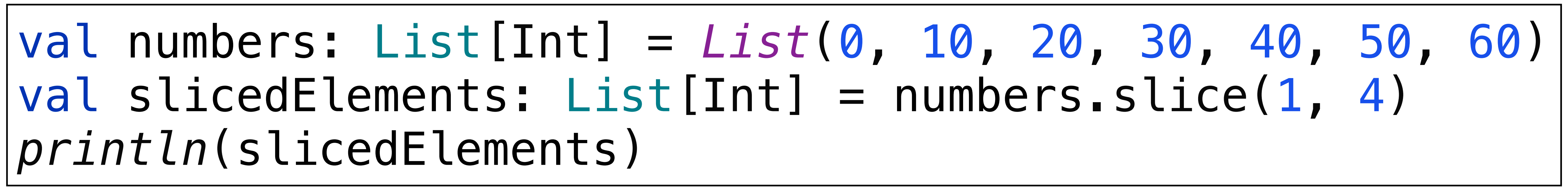


```
val numbers: List[Int] = List(1, 1, 2, 2, 2, 3, 4, 4, 5)
val distinctElements: List[Int] = numbers.distinct
println(distinctElements)
```

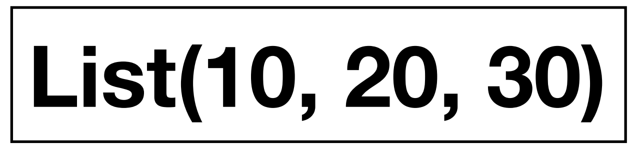
```
List(1, 2, 3, 4, 5)
```

Slice

- Creates a sub-list of a list
- Takes 2 arguments
    - The index of the first element of the slice (Inclusive)
    - The index of the last elements of the slice (Exclusive)
- slice(1, 4) returns a list of the elements from index 1 to 3



```
val numbers: List[Int] = List(0, 10, 20, 30, 40, 50, 60)
val slicedElements: List[Int] = numbers.slice(1, 4)
println(slicedElements)
```



```
List(10, 20, 30)
```

Reverse

- Returns a the elements of the list in reverse order

```
val numbers: List[Int] = List(0, 10, 20, 30, 40, 50, 60)
val reversed: List[Int] = numbers.reverse
println(reversed)
```

```
List(60, 50, 40, 30, 20, 10, 0)
```