Packages E.g

```
src/
 └── com/
    └── example/
       └── utils/
          └── MathUtils.scala
       └── models/
          └── Person.scala
       └── MainApp.scala
```

MathUitls.scala

```
package com.example.utils

object MathUtils {
  def multiply(x: Int, y: Int): Int = x * y
  def divide(x: Int, y: Int): Double = x.toDouble / y
}
```

Person.scala

```
package com.example.models

case class Person(name: String, age: Int)
```

MainApp.scala

```
package com.example

import com.example.utils.MathUtils
import com.example.models.Person

object MainApp {
  def main(args: Array[String]): Unit = {
    val product = MathUtils.multiply(6, 7)
    val quotient = MathUtils.divide(42, 7)

    val person = Person("Alice", 30)

    println(s"Product: $product")
    println(s"Quotient: $quotient")
    println(s"Person: ${person.name}, Age: ${person.age}")
  }
}
```

Currying is a technique in functional programming where a function with multiple arguments is transformed into a series of functions, each taking a single argument.

Instead of taking all arguments at once, the curried function takes one argument and returns a new function that takes the next argument, and so on, until all arguments are provided and the final result is computed.

Key Points about Currying

Transformation: A function of multiple arguments is transformed into a chain of functions each taking a single argument.

Partial Application: Currying allows for partial application, where you can fix a few arguments of a function and get a new function that takes the remaining arguments.

Benefits of Currying

1. **Partial Application**: Currying allows partial application of functions, enabling you to fix some arguments and create new functions.
2. **Function Composition**: It facilitates function composition by allowing you to break down complex functions into simpler ones.
3. **Higher-Order Functions**: Currying works seamlessly with higher-order functions, enhancing modularity and code reuse.

Higher Order Functions

A higher-order function is a function that can take other functions as parameters, return a function as a result, or both

Call by Value

- **Evaluation Strategy**: In call by value, the argument expressions are evaluated before entering the function. The result of this evaluation (the value) is then passed to the function.

- **Behavior**: The expression is evaluated once, and the evaluated value is used wherever the parameter is referenced in the function body.

Call by Name

- **Evaluation Strategy**: In call by name, the argument expressions are not evaluated before entering the function. Instead, the expressions are passed literally to the function and are evaluated each time they are used within the function.

- **Behavior**: The expression is evaluated every time the parameter is referenced in the function body.

First Class Functions

First-class functions are a fundamental concept in functional programming, including in Scala. When we say that functions are "first-class citizens," we mean that functions are treated like any other value. This includes the ability to:

1. **Assign functions to variables.**
2. **Pass functions as arguments to other functions.**
3. **Return functions from other functions.**

Closure

A closure in Scala (and in many other programming languages) is a function that captures the bindings of free variables that are defined in its lexical scope. In simpler terms, a closure is a function that "remembers" the environment in which it was created. This means that it can access variables from its enclosing scope even after that scope has finished execution.

Key Characteristics of Closures

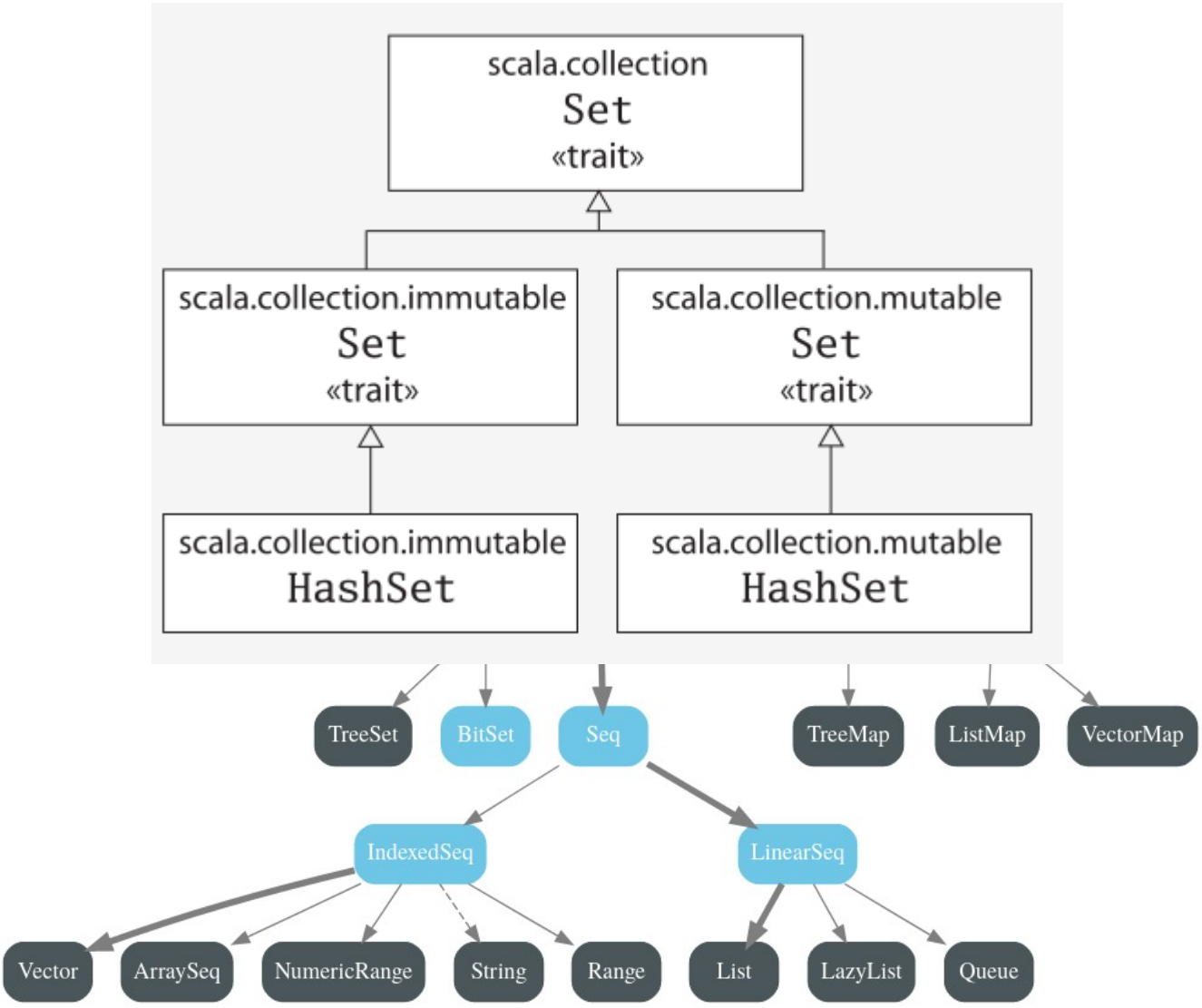
A closure captures the variables from its surrounding scope that are used inside the function.

Remembers the Environment: The function retains access to these variables even when the scope in which they were created has ended.

Three main categories of collections

Looking at Scala collections from a high level, there are three main categories to choose from:

- **Sequences** are a sequential collection of elements and may be *indexed* (like an array) or *linear* (like a linked list)
- **Maps** contain a collection of key/value pairs, like a Java `Map`, Python dictionary, or Ruby `Hash`
- **Sets** are an unordered collection of unique elements



The main collections you'll use on a regular basis are:

Collection Type	Immutable	Mutable	Description
`List`	✓		A linear (linked list), immutable sequence
`Vector`	✓		An indexed, immutable sequence
`LazyList`	✓		A lazy immutable linked list, its elements are computed only when they're needed; Good for large or infinite sequences.
`ArrayBuffer`		✓	The go-to type for a mutable, indexed sequence
`ListBuffer`		✓	Used when you want a mutable `List`; typically converted to a `List`
`Map`	✓	✓	An iterable collection that consists of pairs of keys and values.
`Set`	✓	✓	An iterable collection with no duplicate elements