Slide 1:

- Hello! My name is Valeria Saldana, and today I am going to be walking through key concepts of Currying, Collections and Callbacks, and Abstract Data Types

Slide 2:

- Here's an outline of what I'll cover today:
- First, I'll discuss Currying, understanding how and why it's used.
- Then, I'll go through Collections and Callbacks, focusing on operations like map, flatMap, and folds.
- Lastly, I'll dive into Abstract Data Types, including Map, Set, and parallel and distributed collections.

Slide 3:

- Let's start with currying.

Slide 4:

- Currying is the process of transforming a function with multiple arguments into a chain of functions, each taking a single argument.
- This is incredibly useful for partial application.

Slide 5:

- Here's an example.
- When we call addCurried(3), it returns a new function (y: Int) => 3 + y. Finally, calling this with 4 gives us 7.

Slide 6:

- The key benefit of currying is partial application, where we can fix some arguments and reuse the resulting function.
- For example, we fix 3 in addCurried(3) and reuse the function addThree.
- Now, let's look at an exercise to solidify the concept.

Slide 7:

- In this task, we'll create a curried function that processes lists of numbers, fixing a prefix and multiplier first.

Slide 8:

- To do this, first, we define the curried function processList.

- Then, using partial application, we fix the prefix as 'Result: ' and the multiplier as 2.
- Finally, we apply this partially applied function to different lists of numbers and print the results.

Slide 9:

- Now, let's move on to collections and callbacks. These are powerful tools in functional programming for transforming and aggregating data.
- We'll focus on key operations like map, flatMap, foldRight, and reduce, and understand when to use each.

Slide 10:

- First up is map. It transforms each element of a collection using a callback function.
- For example, given a list of numbers, we can use map to double each value, as shown here.

Slide 11:

- Next is flatMap, which is similar to map but flattens the resulting collections into a single list.
- In this example, each number is mapped to a list of two elements, and flatMap combines them into one list.

Slide 12 & 13:

- foldRight and foldLeft are very similar but foldRight starts from the rightmost element, while foldLeft starts from the left. Folding operations aggregate elements of a collection.
- In these examples, foldRight and foldLeft both calculate the sum of a list, but the order of processing is different.

Slide 14:

- reduce is similar to folds but doesn't require an initial value.
- It directly combines elements of a collection using a binary operation. However, it assumes the collection is non-empty.

Slide 15:

- So, when do we use each one?

- Maps should be used when transforming elements directly.
- Flatmaps when each element results in a collection.
- foldLeft or foldRight for accumulation with an initial value.
- reduce for accumulation when no initial value is needed.

Slide 16:

- Let’s apply these concepts with an exercise. You’re given a list of students and their exam scores.
- The tasks include calculating average scores using map, flattening all scores using flatMap, finding the total sum with foldLeft, and identifying the top student with reduce.

Slide 17:

- First of all, we have to define the data and case class.
- Then, we use map to get the name and the average scores of each student.
- Then we use flatmap to get a list of all of the scores

Slide 18:

- We use foldLeft to get a total sum of all of the scores,
- And lastly, we use reduce to find the student with the highest average score.

Slide 19:

- Finally, let’s discuss Abstract Data Types.
- We’ll cover Map, Set, and parallel or distributed collections.

Slide 20:

- A Set is a collection of unique elements. It’s often used for membership testing and operations like union, intersection, and difference.
- In this example, we demonstrate these operations on two sets, producing their union, intersection, and difference.

Slide 21:

- Parallel and distributed collections allow efficient processing of large datasets by dividing work across CPU cores or distributed systems.
- In Scala, you can use .par to convert a collection into a parallel collection, enabling operations like map and reduce to run concurrently.

Slide 22:

- Here's an exercise to tie everything together. You're given a dataset of employees.
- Each Employee object contains the following fields: id, name, department, salary, and the projects they are involved in.
- Our goal is to process this dataset using Abstract Data Types and higher-order functions.

Slide 23:

- First, we use the filter operation to extract employees from the Engineering department who earn more than $80,000.
- filter takes a predicate function, which is applied to each employee in the list. If the predicate returns true, the employee is included in the resulting list.

- Next, we use map to transform the filtered list into a collection of tuples, where each tuple contains the employee's name and the number of projects they are involved in.
- map applies a function to each element in the collection and returns a new collection of transformed elements.
- For example, Alice is working on two projects, Bob on one, and Diana on one.

- Using flatMap, we generate a flat list of all unique project names across all employees.
- In this case, we extract the projects for each employee, flattening all sets into a single collection.

Slide 24:

- Then, to calculate the total salary of all employees, we use foldLeft. This operation aggregates data by applying a binary function to each element, starting with an initial value.

- Using reduce, we find the employee with the highest salary. reduce works by repeatedly applying a binary function to pairs of elements, returning a single result.

- Finally, we use parallel collections to compute the total salary of employees more efficiently.

Slide 25:

- To summarize, we've explored the power of higher-order functions like map and reduce, the modularity provided by currying, and the versatility of abstract data types like Map and Set.