LINQ

Simon Robinson

Software Developer

@TechieSimon www.SimonRobinson.com

Overview

LINQ Queries

- Remove duplicates
- Group data
  - Sort the grouped data
  - Aggregate the grouped data
- Flatten data
- Join multiple lists

Write custom LINQ extension methods

Leverage lazy evaluation

The Demo

Analyzing student exam result data

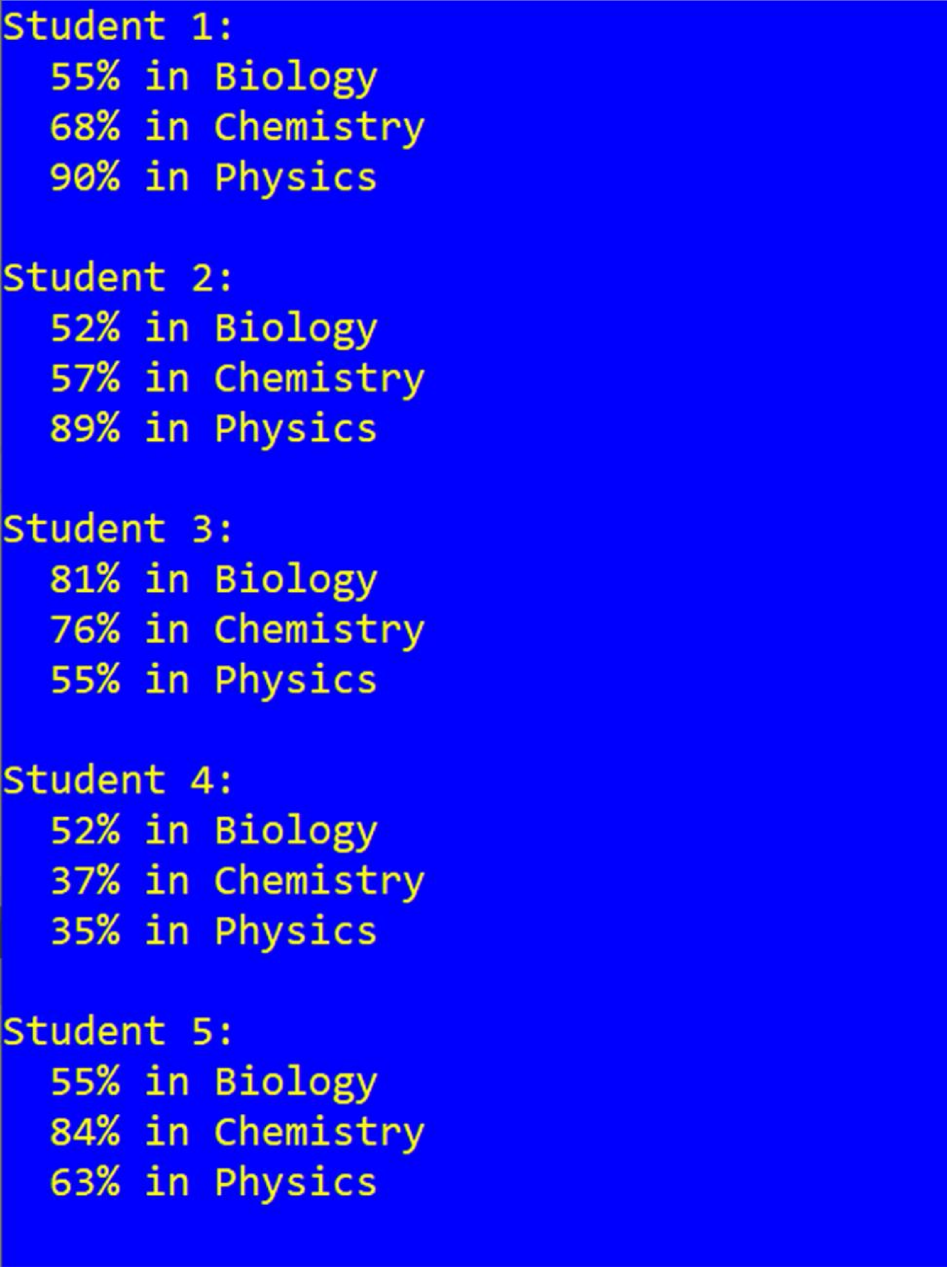
Removing Duplicates

Demo

Setting up the exam result data

- The data is dirty and contains duplicates
- We’ll remove the duplicates

Grouping Your Data


```
Student 1:
  55% in Biology
  68% in Chemistry
  90% in Physics

Student 2:
  52% in Biology
  57% in Chemistry
  89% in Physics

Student 3:
  81% in Biology
  76% in Chemistry
  55% in Physics

Student 4:
  52% in Biology
  37% in Chemistry
  35% in Physics

Student 5:
  55% in Biology
  84% in Chemistry
  63% in Physics
```

This data has a natural hierarchy

The results make sense grouped by the student

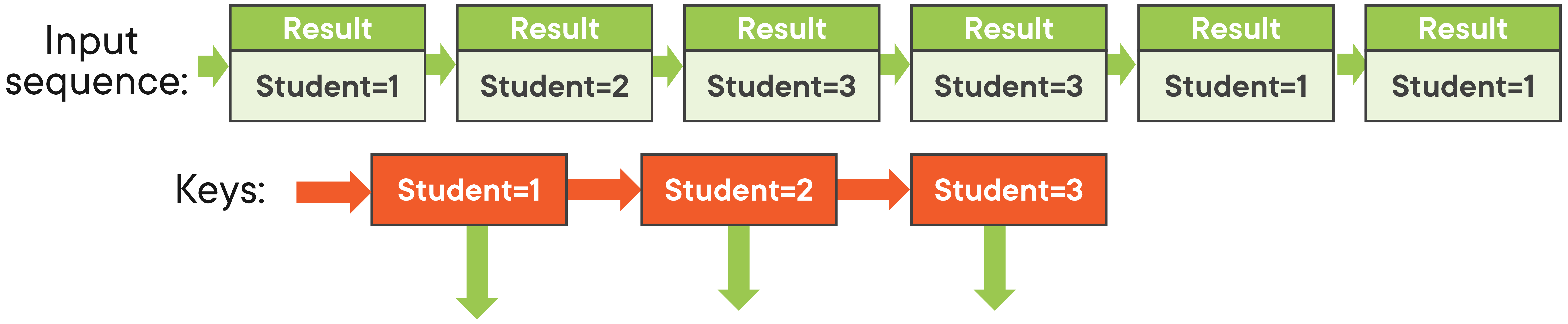
Demo

Grouping in LINQ

- Modify the query so that it generates exam results grouped by student

Grouping

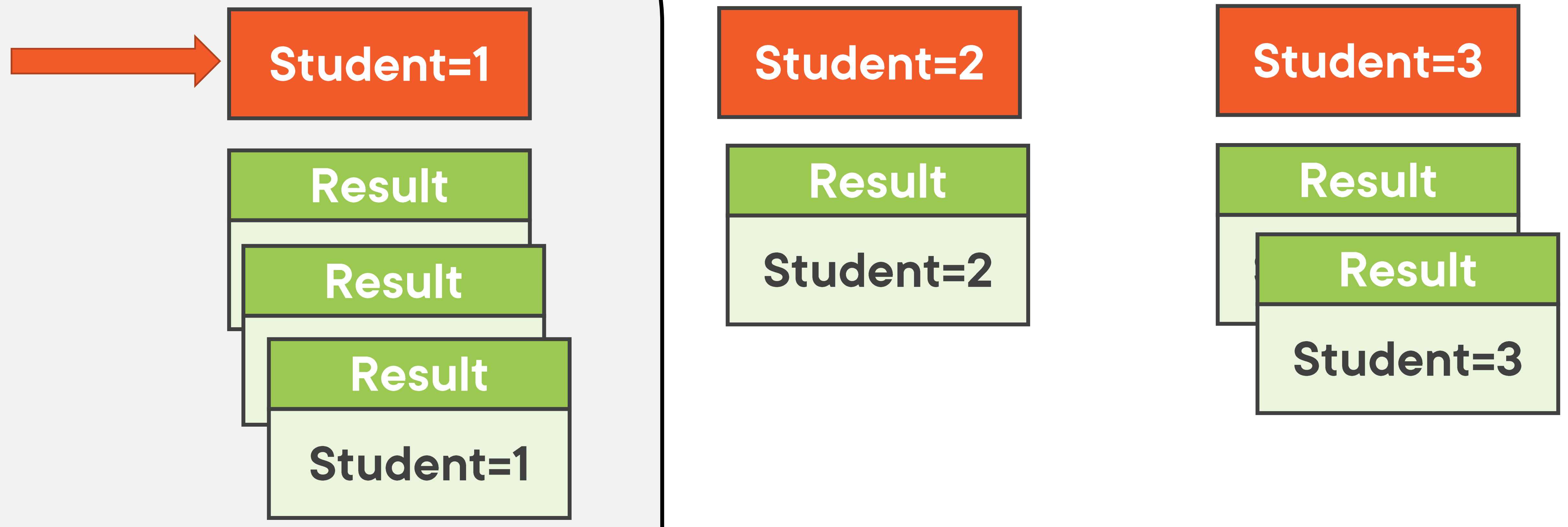
```
var resultsByStudent =
    from result in resultsDistinct
    orderby result.StudentId, result.Subject
    group result by result.StudentId;
```



Grouping

This is a grouping

The key:



```
IGrouping<TKey, TElement>
```

In this case:

```
IGrouping<int, ExamResult>
```

Flattening a List of Lists

Demo

Start with the grouped exam results

- Turn that back into a flat sequence

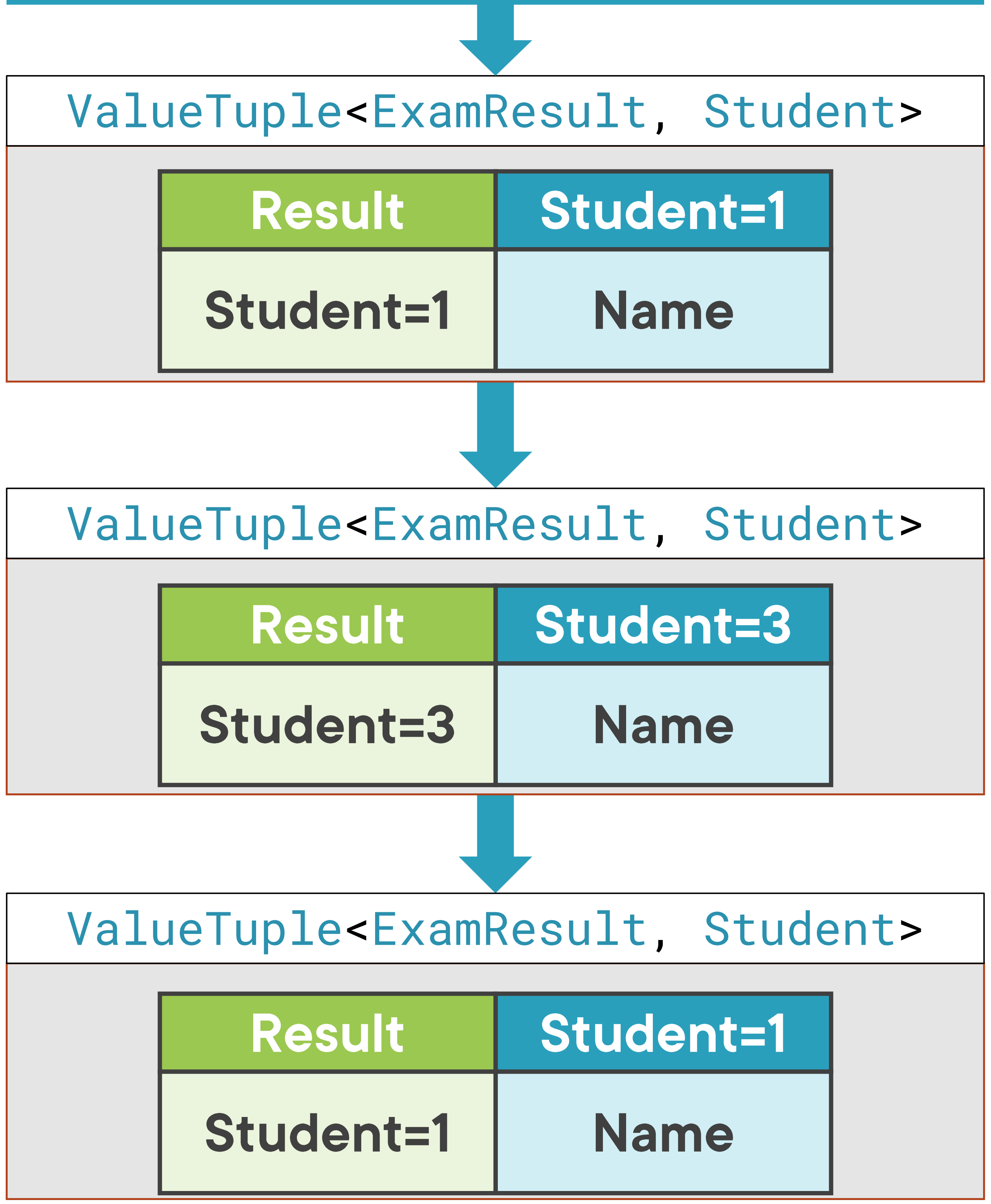
Join Multiple Lists

Demo

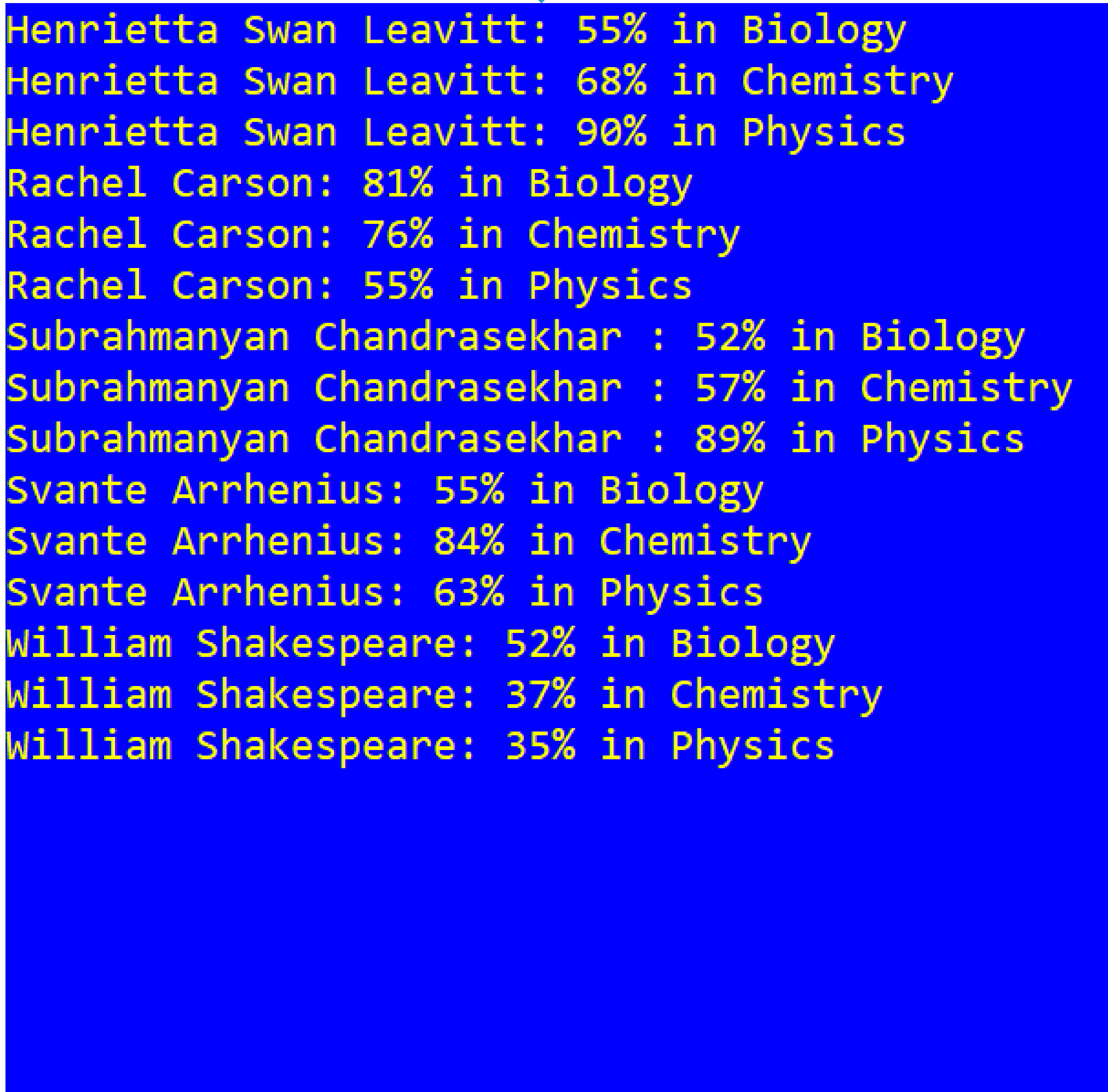
Show exam results with student names

- Requires joining exam results to a list of student names

Desired query results:

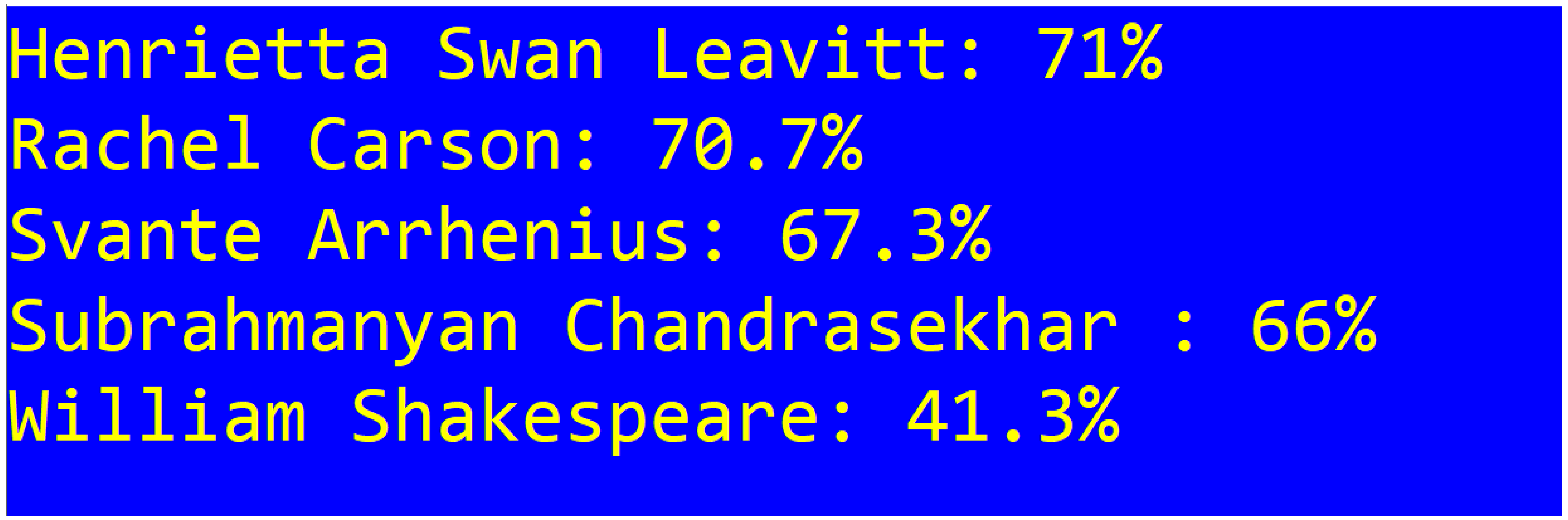

ValueTuple<ExamResult, Student>
Result
Student=1
Student=1
Name
ValueTuple<ExamResult, Student>
Result
Student=3
Student=3
Name
ValueTuple<ExamResult, Student>
Result
Student=1
Student=1
Name

To allow this output:


```
Henrietta Swan Leavitt: 55% in Biology
Henrietta Swan Leavitt: 68% in Chemistry
Henrietta Swan Leavitt: 90% in Physics
Rachel Carson: 81% in Biology
Rachel Carson: 76% in Chemistry
Rachel Carson: 55% in Physics
Subrahmanyan Chandrasekhar : 52% in Biology
Subrahmanyan Chandrasekhar : 57% in Chemistry
Subrahmanyan Chandrasekhar : 89% in Physics
Svante Arrhenius: 55% in Biology
Svante Arrhenius: 84% in Chemistry
Svante Arrhenius: 63% in Physics
William Shakespeare: 52% in Biology
William Shakespeare: 37% in Chemistry
William Shakespeare: 35% in Physics
```

Calculating and Ordering by an Aggregate

The Aim: Ordered Average Marks


```
Henrietta Swan Leavitt: 71%
Rachel Carson: 70.7%
Svante Arrhenius: 67.3%
Subrahmanyan Chandrasekhar : 66%
William Shakespeare: 41.3%
```

We want students ordered by average mark

Getting an aggregate normally just means calling the aggregate method

```
// This is the usual solution – but is problematic here
var x = results.Average();
```

This problem has a twist: The averages are inside the groupings!

Demo

LINQ query

- Take average for each student
- Order by those averages

Creating Custom LINQ Extension Methods

New Requirements:

Slow down LINQ queries (For example, to simulate a slow data source connection)

Log which items are being enumerated

Demo

Implement slowing down and logging

- Write as LINQ extension methods
- Consume them in a LINQ query

Taking Advantage of Lazy Evaluation

Lazy Evaluation (Deferred Execution)

LINQ queries don’t (usually) run when they are set up

They only run when something tries to consume their results

So you don’t use resources getting results that you don’t need

Demo

Investigate lazy evaluation

- Using the `Throttle()` and `Log()` extension methods

Lazy evaluation happens by default in LINQ – you don’t need to do anything to activate it!

Cache results into a collection it you are likely to reuse them

Which LINQ Methods Are Lazy-Evaluated?

Generally executed immediately:

Methods that store results into a collection

```
TSource[] ToArray<TSource>(/* ... */) {}
```

Methods that return a single value

```
double Average(/* ... */) {}

TSource? FirstOrDefault<TSource>(/* ... */) {}
```

Generally lazy-evaluated (Deferred execution):

Methods that return an enumerable (but not a collection)

```
IEnumerable<TSource> Where<TSource>(/* ... */) {}
```

Summary

Queries

- `Distinct()` to remove duplicates
- `group by` to group a flat list
- Multiple `from` clauses to ungroup
  - `SelectMany()` in fluent syntax
- `join` to join lists
- `join into` to group-join lists
  - `Join()` and `GroupJoin()` in fluent syntax

Summary

Extending Linq

- Write extension methods that take `IEnumerable<T>` as first argument

Lazy Evaluation (Deferred execution)

- Avoid calling methods that consume sequences, until required