1.0 Program Code Overview

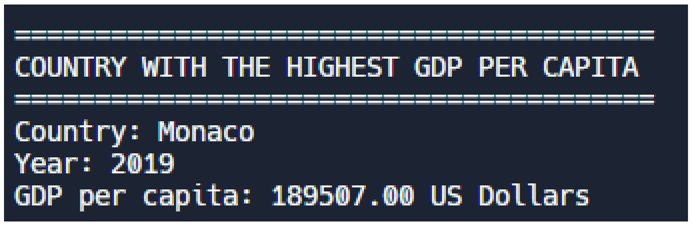
Function Name	**Description**
readFile	Reads a CSV file and extracts relevant data to create instances of *CountryGDP* objects.
getCountry	Retrieves the country name from a *CountryGDP* object.
calculateAverageGDPPerCapita	Calculates the average GDP per capita for a given collection of *CountryGDP* objects.
filterLatestYearData	Filters the *CountryGDP* data to keep only the latest year's data for each country.
findMaxBy	Finds the maximum element in a collection based on a given property.
findGDPMalaysia	Calculates the average GDP per capita for *Malaysia* from the provided data.
findLowestAverageGDPCountries	Finds the countries with the lowest average GDP per capita from the provided data.
main	The main entry point of the program that executes the desired tasks and prints the results.

Case Class	**Description**
CountryGDP	Represents a single data point of GDP information for a specific country. Contains the *country* name, *year*, and *GDP per capita*.

See full code at:
https://replit.com/@prg2104oop042023/Assignment-2-prg2104oop042023-56

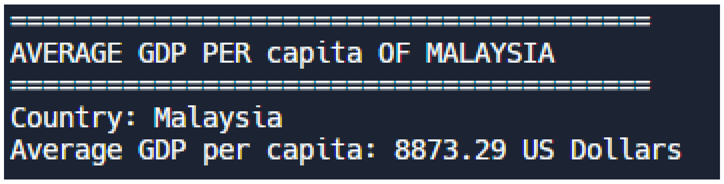
2.0 Result display/Visualization

2.1 Task 1



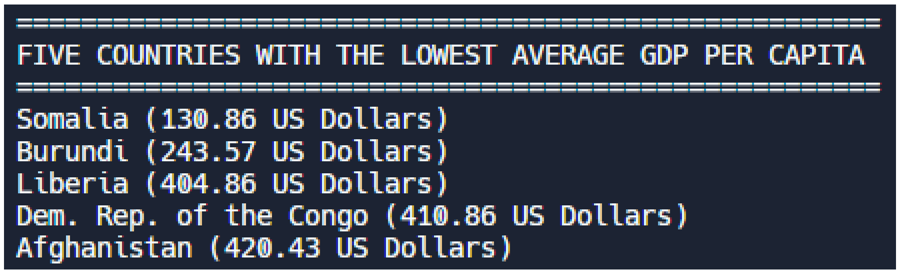
```
========================================
COUNTRY WITH THE HIGHEST GDP PER CAPITA
========================================
Country: Monaco
Year: 2019
GDP per capita: 189507.00 US Dollars
```

2.2 Task 2



```
========================================
AVERAGE GDP PER capita OF MALAYSIA
========================================
Country: Malaysia
Average GDP per capita: 8873.29 US Dollars
```

2.3 Task 3



```
======================================================
FIVE COUNTRIES WITH THE LOWEST AVERAGE GDP PER CAPITA
======================================================
Somalia (130.86 US Dollars)
Burundi (243.57 US Dollars)
Liberia (404.86 US Dollars)
Dem. Rep. of the Congo (410.86 US Dollars)
Afghanistan (420.43 US Dollars)
```

3.0 Polymorphism in Collection API

3.1 Parametric Polymorphism (Generics)

Parametric polymorphism, also known as generics, takes a type as a parameter. They are particularly useful for collection classes (Generic Classes, n.d.). It provides flexibility and reusability by abstracting over specific types.

In our code, parametric polymorphism is used in the following instances:

1. The *calculateAverageGDPPerCapita* function is defined with a type parameter *A*, allowing it to calculate the average GDP per capita for different types of data (Generic Classes, n.d.).

2. The *findMaxBy* function is defined with type parameters *A* and *B*, allowing it to find the maximum element in a collection based on a given property, irrespective of the element and property type (Generic Classes, n.d.).

3.2 Inclusion Polymorphism (Subtyping)

Inclusion polymorphism, also known as subtyping polymorphism, uses base class pointers and references to access derived classes. It allows objects of different classes to be used interchangeably as instances of a common supertype.

In our code, we did not explicitly create an inclusion polymorphism, but in the Collection API, subtyping is used extensively . For instance:

1. Throughout our code, we utilized the *Buffer* class from the Collection API to store *CountryGDP* objects. The *Buffer* class is a mutable supertype in the Collection API hierarchy, which encompasses various implementation classes such as *ArrayBuffer* and *ListBuffer* (Mutable and Immutable Collections, n.d.). By using the *Buffer* supertype, we were able to leverage specific implementation classes that are subtypes of *mutable.Buffer*.

4.0 Benefits of using polymorphism in collection API

4.1 Code Reusability

With polymorphism, we can write generic codes that can operate on different types of collections. This enhances code reusability by reducing duplications of similar codes as methods can be defined in a base class, then overridden in derived classes (Generic Classes, n.d.).

4.2 Code maintainability

Polymorphism improves code maintainability by encapsulating implementation in specific collection classes. This allows easier code updates and changes to the related collection implementations without affecting the code that uses them (Generic Classes, n.d.)..

4.3 Code extensibility

Polymorphism allows easy extension of code as new classes that inherit from existing classes can be created easily, overriding or adding methods that are needed (Scala | Polymorphism, 2019).

5.0 Limitations of using polymorphism in collection API

5.1 Performance overhead

A crucial part of polymorphism is method calling as the specialized collection classes are abstracted in the polymorphism process. This can add on computational cost and introduce a slight performance overhead compared to using the specialized collection class directly.

5.2 Code Complexity

Implementing polymorphism can lead to complex interdependencies between functions, which can make it challenging for programmers to understand the code logic, especially when multiple levels of inheritance are involved (Scala | Polymorphism, 2019). This increased complexity can make the code more difficult to maintain and debug.

References

Generic Classes. (n.d.). Scala Documentation. Retrieved July 15, 2023, from https://docs.scala-lang.org/tour/generic-classes.html

Mutable and Immutable Collections. (n.d.). Scala Documentation. Retrieved July 15, 2023, from https://docs.scala-lang.org/overviews/collections-2.13/overview.html

Scala | Polymorphism. (2019, June 19). GeeksforGeeks.

https://www.geeksforgeeks.org/scala-polymorphism/