OOP and SOLID benefits:

I think that oop and solid principles are essential for developing a good project. The lack of this principles starts to feel especially when the scope of the projects expands. By working on personal projects and school projects (MyGame, Final-Project-Java-College-Class / github), I realized that without these good practices, it would have been very hard to add new functionality on existing modules without having to modify a lot of previous code, scale the projects and overall maintain stability.

To better illustrate how I used some of this principles I will present examples from a school project in Java: <https://github.com/RobertOanta27/Final-project-Java-College-Class>

As my data model in the domain level, I had a WinterGame class and an Enrollment class, one of the private attributes of enrollment was a variable of type WinterGame, this kind of connection between classes is called **Composition (OOP)**, this classes also implemented an interface with a set/get for id.

All the access modifiers of the attributes of these classes are declared private with respective set/get methods, this practice is called data **Encapsulation (OOP)**. This being especially useful when you want to add conditions or limit the possibilities for attributes.

For the repository part, there is a blueprint interface with generic parameters for a repository, this interface being implemented by an abstract class (Abstract\_repository) with generic CRUD operations( using abstract classes and interfaces for hiding data and implementation -> **Abstraction(OOP)** ).

There are also two in-memory repositories for each of my classes which **Inherit(OOP)** from this Abstract class, in this way having access to the implementation of the CRUD operations, this encourages code reusability.

Also there is **Polymorphism(OOP)**, but I don’t recall using it in this specific example project, although it is very useful in inheritance hierarchies by using down/up casting.

Moving on to the SOLID principles, in general I have tried to create classes in such a way that they respect the **Single-Responsibility Principle (SOLID)**, and delegate the tasks that are not their responsibility to other classes. As a more concrete example, ServicesException and RepositoryException have one single purpose : to throw exceptions.

As an example of using the **Open-closed Principle (SOLID)** , the abstract repository can be extended with new functionality, at the same time it’s derived class can use the new functionality without needing modification, this also uses the **Dependency inversion principle (SOLID),** functionality and implementation to depend on abstractions not on concretions.

Also by designing interfaces in a minimalistic way without unnecessary or targeted method ( example Identiafiable -> esential get/set id , necessary for every class) we respect the **Interface Segregation Principle (SOLID).**