# Describe the principle of polymorphism and how it was used in Task 1:

Polymorphism in OOP is a principle that allows objects of different classes to be treated as objects of a common class. This is done using inheritance where different classes inherit from the same parent class, inheriting the parents’ fields and methods. This allows the same methods to be able to work in multiple ways depending on the object it’s called upon. Polymorphism also allows the objects created from the children classes to be of more than one class type.

In Task 1, the Folder and File classes are polymorphic as they inherit from the Thing class. This means they inherit the \_name field and the Name, Size and Print methods from the Thing class. Also, objects created from either of these classes belong to both the child and parent class i.e. an object created from the File class would be of type File and of type Thing. And since both Folder and File classes are polymorphic, they can be both be treated as a “Thing” and when methods like Size or Print are called, the specific implementation of each class is implemented, showcasing that objects created from a File or Folder class can be treated as a File/Folder object or a Thing object, hence again showcasing polymorphism.

# Consider the FileSystem and Folder classes from the updated design in Task 1. Do we need both of these classes? Explain why or why not:

Yes, we do need both the FileSystem and Foler classes because they have different and distinct responsibilities even if they share similarities.

The FileSystem class represents a high level abstraction of how files and folders are organized and managed and if the program were to be developed more this class might even need more methods like initializing the system which would not translate for the Folder class.

The Folder class represents just a collection of file or other folders and is a much simpler abstraction that wouldn’t require the complex functionalities of FileSystem if the program were to be developed further.

It would be poor abstraction to merge them to have the same responsibilities as both these classes are not supposed to have the same functionalities. For example, if FileSystem were to have a method like Size, it would mean computing the size of the entire system which wouldn’t make sense whereas for the Folder class which would just compute the size of its contents.

# What is wrong with the class name Thing? Suggest a better name for the class, and explain the reasoning behind your answer:

The class name Thing is quite vague and doesn’t represent its child classes accurately enough. A better class name would be “Item” as the term items are more commonly associated with lists and directories.

# Define the principle of abstraction, and explain how you would use it to design a class to represent a Book:

In OOP, abstraction is the principle of constructing the ideas and blueprints of each component in a program before the code is written. If we were to create a Book class using abstraction, at first, we would try to determine how many components there were and what their responsibilities/characteristics were:

* A Book has a title, an author, a date published.
* A book has many pages.
* You should be able to turn to the next and previous page.
* A page has text on it.
* A page is supposed to be read.

Using these key ideas of what our programs characteristics/responsibilities are, we can start constructing our abstractions.

There should be two classes, a Page class, and a Book class. The Page class should contain a text field that contains the writing of the page and a method that displays the text. The Book class should contain fields and properties for its title, author and publish date. It should also contain a List field containing Page objects which represent each page of the book and it should have a method that allows the user to turn to the next or previous Page item in the list.