1. Describe the principle of polymorphism and how it was used in Task 1
   - **Polymorphism:** Polymorphism in OOP allows objects of various classes to be treated as if they were objects of the same superclass. This allows for the use of a single interface for several data types, increasing code reusability and flexibility.
   - **Usage in Task 1:** Since the "Thing class" is abstract, both the File and Folder subclasses override its methods Size() and Print(). This demonstrates polymorphism by enabling the addition of both files and folders to the "_contents" list in the FileSystem and Folder classes.

2. 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?
   The FileSystem and Folder classes each play a unique role in Task 1, despite the fact that at first glance their functions may appear to overlap.
   - **Purpose:** The FileSystem class, which stores numerous objects like files and directories and so represents a full file system, serves as an overarching framework. In contrast, the Folder class mimics real-world directory hierarchies by acting as a sub-structure that may include more files and sub-folders.
   - **Scalability and Maintainability:** Separating them enables for more flexible and adaptable architecture. If one were to implement unique methods or characteristics relevant to either the file system or individual folders in future iterations (such as rights, ownership, or mounting capabilities for FileSystem), having discrete classes would make this process simple.

   Therefore, despite the fact that both classes handle and store "Thing" objects, their responsibilities, scalability, and representation in a common file system point to the requirement of both types. Their separation follows the Single Responsibility Principle, which ensures that each class has only one cause to change.

3. What is wrong with the class name Thing? Suggest a better name for the class and explain the reasoning behind your answer.
   The category name Particularly in the context of a file system, Thing is somewhat unclear and does not provide an intuitive idea of its function or the kinds of items it's designed to represent.
   - **Specificity and Context:** One would anticipate class names that reflect real-world elements in a system that deals with files and folders. A class's purpose in representing entities in a file system cannot be understood by its general name, Thing.
   - **Improved Name Suggestion:** FileSystemEntity or FileSystemItem could be more appropriate names for the Thing class. Both approaches give a more straightforward representation of the purpose of the class and provide clarity when developers or collaborators interact with the codebase.

By renaming the Thing class to something more descriptive, we ensure that the name of the class is more directly related to its purpose and function, enabling improved readability and maintainability.

4. Define the principle of abstraction and explain how you would use it to design a class to represent a Book.
    - **Abstraction:** Abstraction in OOP refers to the notion of concealing complicated implementation details and displaying only the basic properties of an object. It is concerned with simplifying complicated reality by modelling classes based on the traits and behaviors that an item should have.
    - **Application for a Book class:** If we were to use abstraction to create a class for a book, we would concentrate on the fundamental attributes and operations that characterise a book. The following are some examples of attributes and methods: ReadPage(), BookmarkPage(), ISBN, Title, Author, and PageCount. The colour of the book is an example of a non-essential element that we wouldn't add unless it was pertinent to the application at hand. By doing this, we are able to convey the core of what a book is and how it works without becoming mired down in pointless details.