**Chapter 9**

**Exercises**

**9.3 (Using Composition Rather Than Inheritance)**

Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class BasePlusCommissionEmployee (Fig. 9.11) of the CommissionEmployee–BasePlusCommissionEmployee hierarchy to use composition rather than inheritance.

**9.4 (Software Reuse)**

Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors.

**1. Promoting Software Reuse:**

* **Code Reusability:**
  + Inheritance allows a new class (subclass or derived class) to inherit properties and methods from an existing class (superclass or base class). This means that code written in the superclass doesn't need to be rewritten in the subclass.
  + Instead, the subclass can reuse the existing code and extend or modify it as needed.
* **Creating Hierarchies:**
  + Inheritance facilitates the creation of class hierarchies, which model real-world relationships. For example, a "Vehicle" superclass can have subclasses like "Car," "Truck," and "Motorcycle."
  + This hierarchical structure promotes a clear organization of code and encourages the development of reusable components.
* **Abstracting Common Functionality:**
  + Superclasses can be used to define abstract or general functionality that is common to multiple subclasses. This allows developers to create reusable building blocks that can be adapted to specific needs.
  + For example, a superclass could define a general "shape" class, and subclasses could be "circle", "square" and "triangle". The super class could define general methods like "calculateArea" that the subsclasses then override to implement the specific calculations.

**2. Saving Time During Program Development:**

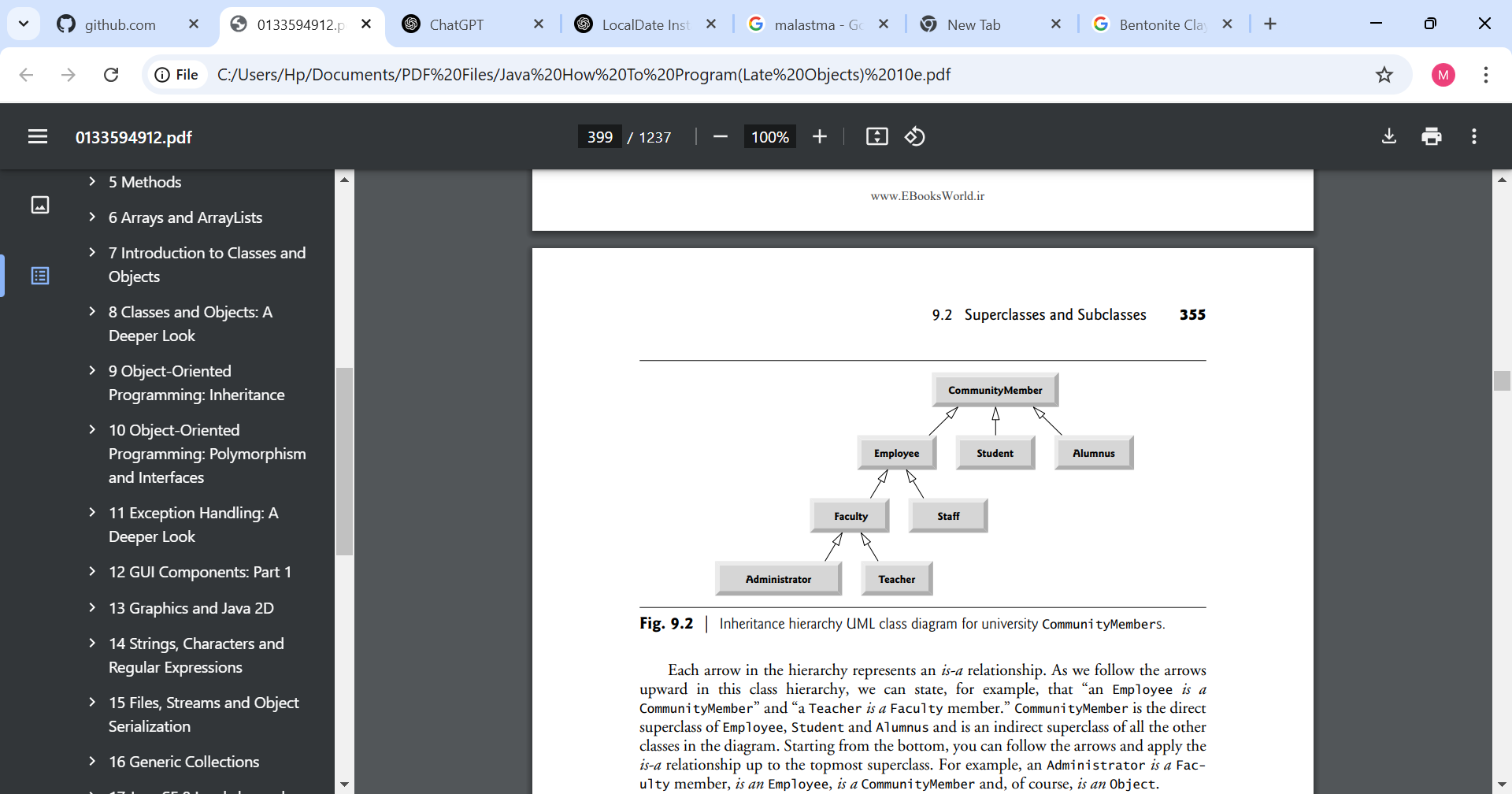
* **Reduced Development Time:**
  + By reusing existing code, developers can significantly reduce the amount of time spent writing new code.
  + Instead of starting from scratch, they can build upon existing classes, which speeds up the development process.
* **Faster Prototyping:**
  + Inheritance enables rapid prototyping by allowing developers to quickly create new classes that inherit from existing ones.
  + This allows for faster experimentation and iteration during the development cycle.
* **Focus on Specialization:**
  + Developers can focus their time on implementing the unique functionalities of the subclass, rather than re-implementing the common functionalities already present in the superclass.

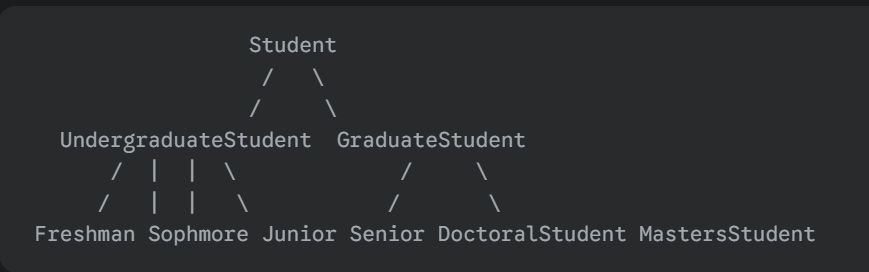
**3. Helping Prevent Errors:**

* **Reduced Code Duplication:**
  + Code duplication is a common source of errors. Inheritance helps to eliminate code duplication by allowing developers to reuse existing code.
  + When code is reused, there is less chance of introducing new errors or inconsistencies.
* **Improved Code Maintainability:**
  + Inheritance makes code easier to maintain by centralizing common functionality in superclasses.
  + If a change needs to be made to the common functionality, it only needs to be made in the superclass, and all subclasses will automatically inherit the change.
* **Increased Code Reliability:**
  + Code that has been thoroughly tested in a superclass can be reused with confidence in subclasses.
  + This helps to increase the overall reliability of the software.
* **Enforcing Consistency:**
  + By having a superclass enforce certain behaviors, all of the subclasses will behave in a predictable and consistent manner. This reduces unexpected bugs caused by inconsistent code.
* **Well Defined Interfaces:**
  + Superclasses can define interfaces, or abstract methods that must be implemented by the subclasses. This enforces a contract, and helps to make sure that the subclasses implement the needed functionality.

**9.5 (Student Inheritance Hierarchy)**

Draw an inheritance hierarchy for students at a university similar to the hierarchy shown in Fig. 9.2.

Use Student as the superclass of the hierarchy, then extend Student with classes UndergraduateStudent and GraduateStudent. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, Freshman, Sophomore, Junior and Senior might extend UndergraduateStudent, and DoctoralStudent and MastersStudent might be subclasses of GraduateStudent. After drawing the hierarchy, discuss the relationships that exist between the classes. [Note: You do not need to write any code for this exercise.]

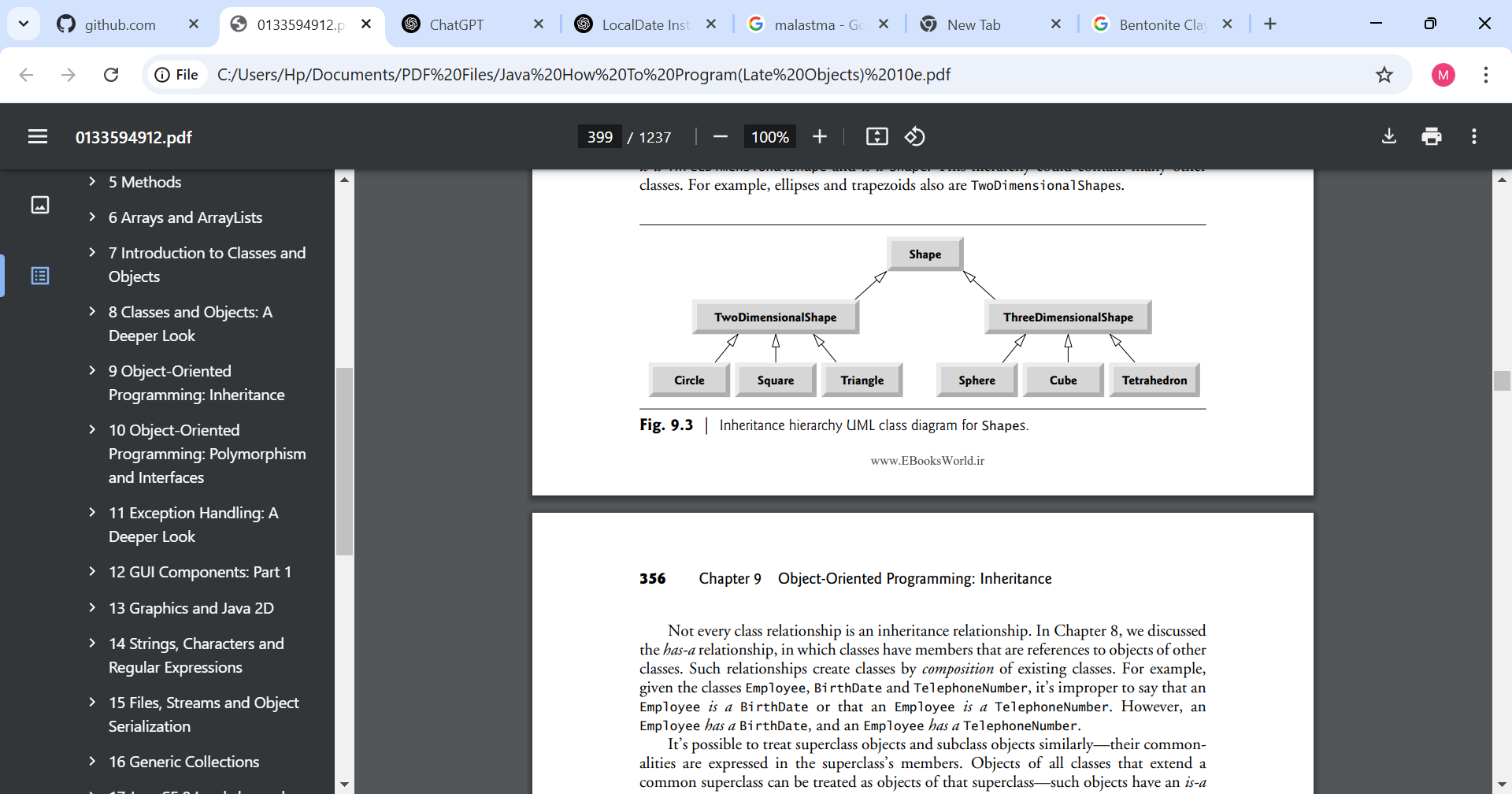


**Explanation of the Hierarchy**

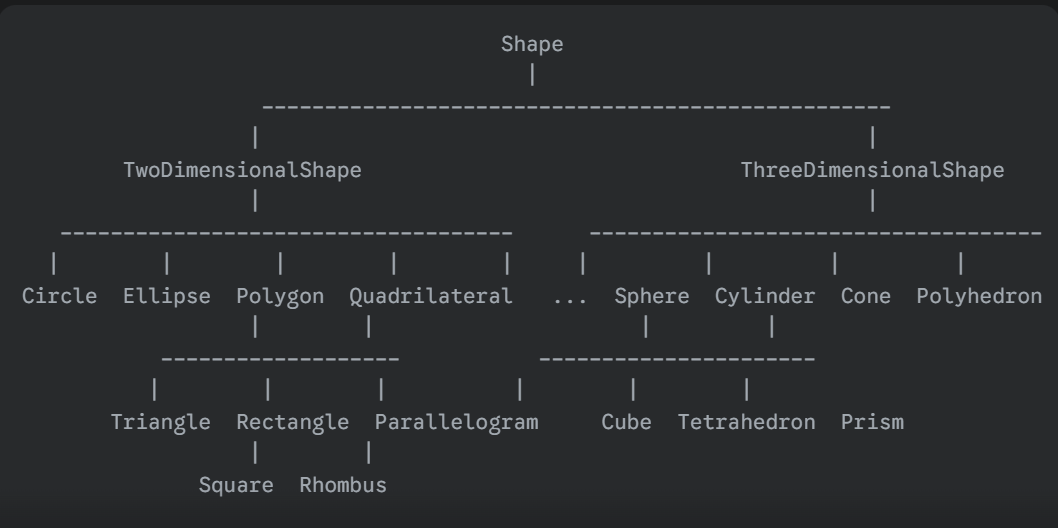
* **Student (Superclass):**
  + This is the base class, representing the most general concept of a student. It would contain common attributes and methods shared by all students, such as student ID, name, enrolled courses, etc.
* **UndergraduateStudent (Subclass):**
  + This class inherits from Student and represents students pursuing a bachelor's degree.
  + It would include attributes and methods specific to undergraduates, such as major, GPA, and expected graduation year.
* **GraduateStudent (Subclass):**
  + Also inherits from Student, representing students pursuing advanced degrees (master's or doctorate).
  + It would include attributes and methods specific to graduate students, such as research area, advisor, and thesis/dissertation status.
* **Freshman, Sophomore, Junior, Senior (Subclasses of UndergraduateStudent):**
  + These classes further specialize UndergraduateStudent based on academic year.
  + They might include attributes or methods specific to each year, such as course load requirements or eligibility for certain programs.
* **DoctoralStudent, MastersStudent (Subclasses of GraduateStudent):**
  + These classes specialize GraduateStudent based on the type of graduate degree being pursued.
  + They would have distinct attributes and methods related to the specific degree requirements.

**9.6 (Shape Inheritance Hierarchy)**

The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 9.3.



Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete Shape hierarchy with as many levels as possible. Your hierarchy should have class Shape at the top. Classes TwoDimensionalShape and ThreeDimensionalShape should extend Shape. Add additional subclasses, such as Quadrilateral and Sphere, at their correct locations in the hierarchy as necessary.



**9.7 (protected vs. private)**

Some programmers prefer not to use protected access, because they believe it breaks the encapsulation of the superclass. Discuss the relative merits of using protected access vs. using private access in superclasses.

**Private Access in Superclasses**

* **Merits:**
  + **Strong Encapsulation:** private access provides the strongest form of encapsulation. Only the superclass itself can directly access its private members. This prevents subclasses (and other classes) from inadvertently modifying or relying on the internal state of the superclass.
  + **Reduced Coupling:** private access minimizes the coupling between the superclass and its subclasses. Subclasses are forced to interact with the superclass through its public interface, which promotes loose coupling and makes the code more robust to changes.
  + **Improved Maintainability:** Changes to the internal implementation of the superclass are less likely to affect subclasses when private access is used. This makes it easier to maintain and evolve the superclass without breaking existing code.
  + **Increased Code Safety:** Preventing subclasses from directly accessing the superclass's internals reduces the risk of unintended side effects and errors.
* **Demerits:**
  + **Reduced Flexibility:** private access can limit the flexibility of subclasses. If a subclass needs to access or modify a member of the superclass, it has to rely on public methods, which might not always provide the desired level of access.
  + **Increased Boilerplate Code:** Subclasses may need to call many getter and setter methods to interact with the superclass, leading to more verbose and potentially less efficient code.
  + **Difficulty in Extending Functionality:** Some functionality that would be naturally implemented with direct access to superclass members might become convoluted and less efficient when forced to use only public methods.

**Protected Access in Superclasses**

* **Merits:**
  + **Increased Flexibility:** protected access allows subclasses to directly access and modify the protected members of the superclass. This can be useful for implementing specialized behavior in subclasses.
  + **Reduced Boilerplate Code:** Subclasses can directly access protected members, which can reduce the need for getter and setter methods and make the code more concise.
  + **Facilitates Inheritance:** protected access is often seen as a way to facilitate inheritance by providing subclasses with the necessary access to the superclass's internals.
  + **Improved Performance:** Direct access to protected members can sometimes lead to performance improvements compared to using public methods.
* **Demerits:**
  + **Weakened Encapsulation:** protected access weakens the encapsulation of the superclass. Subclasses can directly access and modify the protected members, which can lead to unintended side effects and errors.
  + **Increased Coupling:** protected access increases the coupling between the superclass and its subclasses. Subclasses become more dependent on the internal implementation of the superclass, which can make the code more fragile.
  + **Reduced Maintainability:** Changes to the protected members of the superclass can affect subclasses, making it more difficult to maintain and evolve the superclass.
  + **Increased Risk of Errors:** Direct access to the superclass's internals increases the risk of subclasses introducing errors or inconsistencies.

**9.8 (Quadrilateral Inheritance Hierarchy)**

Write an inheritance hierarchy for classes Quadrilateral, Trapezoid, Parallelogram, Rectangle and Square. Use Quadrilateral as the superclass of the hierarchy. Create and use a Point class to represent the points in each shape. Make the hierarchy as deep (i.e., as many levels) as possible. Specify the instance variables and methods for each class. The private instance variables of Quadrilateral should be the x-y coordinate pairs for the four endpoints of the Quadrilateral. Write a program that instantiates objects of your classes and outputs each object’s area (except Quadrilateral).