Nguyễn Hữu Hoàng Hải Anh

20226010

1. **What are the advantages of polymorphism?**
   - Code Reusability: Polymorphism allows you to write code that can work with objects of various classes if they inherit from a common superclass or implement a common interface. This promotes code reuse and reduces redundancy, as you can write generic code that can be applied to a variety of specific cases.
   - Flexibility and Extensibility: Polymorphism makes your code more flexible and extensible. You can add new classes that inherit from the common superclass or implement the common interface without modifying existing code. This makes it easier to extend your codebase and accommodate changes in requirements.
   - Simplified Code Maintenance: By relying on polymorphism, you can write code that is easier to maintain. When you need to update how objects are processed, you can often do so by modifying the superclass or interface, rather than making changes throughout the codebase.
   - Improved Readability and Organization: Polymorphism promotes a more hierarchical and organized code structure. By defining common behavior in a superclass or interface, you make it clear to other developers how different classes relate to each other and what functionality they share.
   - Dynamic Binding: Polymorphism enables dynamic binding, where the appropriate method implementation is determined at runtime based on the actual type of the object. This allows for more flexible and dynamic behavior, as the method invoked can vary depending on the object being operated upon.
2. **How is inheritance useful when using polymorphism in Java?**
   - Code Reuse: Inheritance allows subclasses to inherit attributes and methods from their superclass. This means that subclasses automatically have access to the behavior defined in the superclass. When using polymorphism, you can define a common superclass with generic behavior, and then create subclasses that provide specialized implementations. This promotes code reuse, as you don't need to duplicate code that is common across multiple classes.
   - Subtype Polymorphism: Inheritance enables subtype polymorphism, also known as runtime polymorphism. This means that a reference variable of a superclass type can refer to an object of any subclass type. When you invoke a method on this reference variable, the JVM dynamically determines which implementation of the method to execute based on the actual type of the object. This allows for flexible and dynamic behavior, as different objects can exhibit different behaviors based on their specific implementations of inherited methods.

- Extensibility: Inheritance facilitates the extension of existing classes to create new classes with additional functionality. You can create new subclasses that inherit from existing classes and then add new attributes or methods, or override existing methods to provide specialized behavior. This makes it easy to extend and modify the behavior of your code without having to modify the original class.
- Code Organization: Inheritance promotes a hierarchical organization of classes, where subclasses inherit from a common superclass. This helps to organize your codebase and makes it easier to understand the relationships between different classes. By defining common behavior in a superclass and specific behavior in subclasses, you can create a clear and logical structure for your code.

3. What are the differences between Polymorphism and Inheritance in Java?

Feature	Polymorphism	Inheritance
Purpose	Allows objects of different classes to be treated as objects of a common superclass.	Enables code reuse by allowing a new class to inherit attributes and methods from an existing class.
Relationship	Achieved through inheritance, interfaces, and method overriding.	Involves creating a hierarchical relationship between classes where subclasses inherit from superclasses.
Type of Behavior	Allows for dynamic binding and method overriding, enabling objects to exhibit different behaviors at runtime.	Facilitates code reuse and extension by allowing subclasses to inherit and override or extend behavior from their superclass.
Usage	Used to create flexible and dynamic behavior, promote code reuse, simplify code maintenance, and enable extensibility.	Used to model "is-a" relationships between classes, promote organization, and facilitate code reuse.
Implementation	Implemented through method overriding (runtime polymorphism), method overloading (compile-time polymorphism), interfaces, and abstract classes.	Implemented by extending a superclass using the **extends** keyword to create a subclass. Subclasses inherit all non-private members of the superclass.