1.Create an inheritance hierarchy of Rodent: Mouse, Gerbil, Hamster,etc. In the base class, provide methods that are common to all Rodents, and override these in the derived classes to perform different behaviors depending on the specific type of Rodent. Create an array of Rodent, fill it with different specific types of Rodents, and call your base-class methods to see what happens. Make the methods of Rodent abstract whenever possible and all classes should have default constructors that print a message about that class.

Approach

Abstract Classes and Methods:
The Rodent class is abstract, meaning it cannot be instantiated directly. It serves as a base class for other rodent types.
Abstract methods in the Rodent class (eat, sleep, playingGames) must be implemented by any concrete subclass.

Inheritance and Polymorphism:
The Mouse, Gerbil, and Hamster classes all extend the Rodent class and provide specific implementations for the abstract methods.
Polymorphism allows the program to treat instances of Mouse, Gerbil, and Hamster as instances of Rodent. This enables the use of a single array to store different types of rodents and call their methods in a uniform way.

Constructors:
Constructors in derived classes (Mouse, Gerbil, Hamster) call the constructor of the base class (Rodent) implicitly or explicitly.
This is demonstrated by the messages printed in the constructors of Rodent and its subclasses.

Arrays and Looping:
An array of Rodent is created to hold different rodent objects.
A for-each loop is used to iterate over the array and call the methods of each rodent.

2.Create a Cycle class, with subclasses Unicycle, Bicycle and Tricycle. Add a balance( ) method to Unicycle and Bicycle, but not to Tricycle. Create instances of all three types and upcast them to an array of Cycle. Try to call balance( ) on each element of the array and observe the results. Downcast and call balance( ) and observe what happens.

Approach
Base class-Cycle class
Child Class-UniCycle,BiCycle,TriCycle
So we can call the subclass instance with the help of base class this help us to achieve polymorphism

3. Create three interfaces, each with two methods. Inherit a new interface that combines the three, adding a new method. Create a class by implementing the new interface and also inheriting from a concrete class. Now write four methods, each of which takes one of the four interfaces as an argument. In main( ), create an object of your class and pass it to each of the methods.
Approach
Interfaces-InterfaceOne,InterfaceTwo,InterfaceThree,NewInterface
Abstract class-ClassImplementation
So we can call the subclass by suing the help of the interface or the parent class; this help us to achieve polymorphism.

Approach

4.Create a Cycle interface, with implementations Unicycle, Bicycle and Tricycle. Create factories for each type of Cycle, and code that uses these factories.

-The code uses the Factory Method Design Pattern to create instances of different types of cycles (UniCycle, BiCycle, TriCycle) without specifying the exact class of the object that will be created.
-The Cycle interface defines a common type for all cycles, and concrete classes provide specific implementations. The CycleFactory interface provides a way to create these cycles, and concrete factory classes instantiate the appropriate cycle type.
-By using factory classes to create objects, the code is decoupled from the specific classes that it instantiates, making it more flexible and easier to extend.

5. Create a class with an inner class that has a non-default constructor (one that takes arguments). Create a second class with an inner class that inherits from the first inner class.

Approach
Creating an Instance of OuterClass.InnerClass:
OuterClass.InnerClass inner = new OuterClass().new InnerClass("Hello from InnerClass");
The InnerClass constructor is called with the message "Hello from InnerClass".
The displayMessage() method prints "Message: Hello from InnerClass".

Creating an Instance of SecondClass.SecondInnerClass:
SecondClass.SecondInnerClass secondInner = new SecondClass().new SecondInnerClass(new OuterClass(), "Hello from InheritedInnerClass");
The SecondInnerClass constructor is called with a new instance of OuterClass and the message "Hello from InheritedInnerClass".
The displayMessage() method of SecondInnerClass prints "Inherited Message: " and then calls the superclass displayMessage() method, which prints "Message: Hello from InheritedInnerClass".