Explicación de Ejercicios:

Ejercicio 1: En este ejercicio por un lado creamos la interfaz ReproductorMusica. Dentro de esta interfaz se observa el **comportamiento**, ya que define los métodos reproducir(), pausar() y detener(). La interfaz establece qué debe hacer cualquier reproductor de música, pero no indica cómo hacerlo.

Luego creamos las clases ReproductorMP3 y ReproductorStreaming. En ambas clases se cumple la **herencia** junto con la **implementación**, ya que las dos implementan la interfaz ReproductorMusica. Esto las obliga a sobrescribir (@Override) todos los métodos definidos en la interfaz. Acá es donde se da la implementación concreta, porque cada clase especifica cómo reproducir, pausar y detener música según su tipo: en el caso de MP3 trabajando con archivos locales y en el caso de Streaming con una URL en la nube. Además, en la sobrescritura de estos métodos también se refleja el **polimorfismo**, ya que aunque el nombre del método sea el mismo (reproducir(), pausar(), detener()), la acción depende de la clase que lo implemente.

Finalmente en el main se observa de forma clara el **polimorfismo**. La variable ReproductorMusica reproductor=null; esta declarada con el tipo de la interfaz, lo que le permite dirigirse tanto a un objeto de la clase ReproductorMP3 como a uno de la clase ReproductorStreaming. Cuando se llama a un método como reproductor.reproducir(), el comportamiento que se ejecuta depende del tipo de objeto que tenga en ese momento (MP3 o Streaming). Esto es polimorfismo, ya que un mismo mensaje genera distintas acciones según la implementación.

Ejercicio 2: En este ejercicio creamos la interfaz Pago, que define los métodos realizarPago(double monto) ,obtenerComprobante() y getMonto(). Dentro de esta interfaz se representa el **comportamiento**, porque se establece qué debe hacer cualquier forma de pago, sin indicar cómo se implementa.

Luego creamos la clase PagoEfectivo, PagoTarjetaCredito y PagoTransferencia. En cada una se cumple la **herencia**, ya que todas implementan la interfaz Pago mediante implements, lo que las obliga a sobrescribir todos los métodos de la interfaz. Además, en estas clases se ve la **implementación**, porque cada una define cómo se realiza el pago y cómo se genera el comprobante, mostrando distintos mensajes según el tipo de pago. Cada clase sobrescribe (@Override) los métodos de la interfaz, y al hacerlo cada objeto ejecuta su propia versión de realizarPago(double monto), obtenerComprobante() y getMonto(), lo que permite que el mismo método se comporte distinto según el tipo de pago (**polimorfismo**).

Después creamos la clase HistorialPagos, que se encarga de guardar y manejar todos los pagos realizados en el sistema. En esta clase se nota el **polimorfismo**, porque aunque la lista guarde objetos de tipo Pago, en realidad ahí adentro puede haber pagos en efectivo, pagos con tarjeta de crédito o pagos por transferencia. Cuando se llaman los métodos obtenerComprobante() o getMonto(), se ejecuta la acción propia de cada tipo de pago, sin importar de qué clase sea. Además, el historial **implementa** métodos para registrar un pago nuevo, mostrar todos los comprobantes y calcular el total recaudado. De esta forma se puede trabajar con distintos tipos de pagos de manera general gracias al polimorfismo.

Finalmente en el main creamos un menú principal que le permite al usuario interactuar con el sistema. Desde acá se puede elegir el tipo de pago, ingresar el monto y, si corresponde, los datos adicionales como número de tarjeta o cuenta, guardar el pago en el historial, consultar los comprobantes y ver el total recaudado. Lo interesante es que cuando se registra un pago, se aplica el **polimorfismo**, porque aunque trabajamos con referencias de tipo Pago, cada objeto (PagoEfectivo, PagoTarjetaCredito o PagoTransferencia) responde con su propia forma de realizar el pago. Gracias a este menú, el usuario puede probar fácilmente cómo funciona todo el sistema y ver las diferencias entre los distintos tipos de pagos.

Ejercicio 3: En este ejercicio por un lado creamos la clase abstracta Empleado, donde dentro de esta clase podemos ver que se cumple el **comportamiento** en los métodos abstractos trabajar(), calcularBono() y getDetalles() ya que estos métodos nos indican lo que todos los empleados deben hacer(trabajar), calcular o mostrar pero no indican el cómo ya que su implementación queda en manos de las subclases.

Luego creamos las clases Programador y Diseñador, ambas **heredan** de la clase Empleado mediante el extends. En sus constructores utilizan el super para inicializar los atributos heredados y, al heredar de Empleado, deben **implementar** los métodos trabajar(), calcularBono() y getDetalles(), especificando cómo se realiza cada acción según el tipo de empleado que sea. Cada subclase sobrescribe (@Override) los métodos abstractos de Empleado, y al hacerlo, cada objeto ejecuta su propia versión de trabajar(), calcularBono() y getDetalles(), mostrando cómo un mismo mensaje puede comportarse distinto según el tipo de empleado (**polimorfismo**).

Después creamos la clase Empresa, que se encarga de manejar a todos los empleados. Acá se ve el **polimorfismo**, porque aunque la lista guarde objetos de tipo Empleado, en realidad ahí se pueden guardar tanto programadores como diseñadores. Cuando se llama a los métodos getDetalles() o calcularBono(), se ejecuta la versión propia de cada tipo de empleado. Además, la clase **implementa** métodos para agregar nuevos empleados, mostrarlos con su nombre, salario y tipo, y calcular el gasto total de la empresa. Así se pueden manejar distintos empleados de forma general gracias al polimorfismo.

Finalmente en el main armamos un menú que deja al usuario agregar empleados, mostrarlos y calcular el gasto total de la empresa. En este bloque también aparece el **polimorfismo**, porque aunque se creen objetos de tipo Programador o Diseñador, todos se guardan en referencias del tipo Empleado. Al llamar a getDetalles() o calcularBono(), cada uno responde con su propia implementación. Esto permite trabajar con distintos empleados sin tener que estar comprobando de qué tipo es cada uno, lo que simplifica mucho el manejo del programa.

Ejercicio 4: En este ejercicio primero creamos la interfaz Figura, dentro de ella podemos ver el **comportamiento** a través de sus métodos area() y perimetro() ya que la interfaz define que métodos debe cumplir cualquier figura, pero no cómo se implementan.

Luego creamos la clase Circulo, Rectangulo y Triangulo. En cada una se cumple la **herencia**, ya que todas implementan la interfaz Figura mediante el implements, lo que las obliga a sobrescribir (@Override) los métodos area() y perimetro() de la interfaz. Además, en estas clases se ve la **implementación**, porque cada una define cómo calcular el área y el perímetro según la figura correspondiente, mostrando que un mismo método se comporta distinto según el tipo de objeto (**polimorfismo**).

Después creamos la clase GestorFiguras, que se encarga de manejar todas las figuras creadas. En esta clase se nota el **polimorfismo**, porque aunque la lista guarda objetos de tipo Figura, en realidad puede contener círculos, rectángulos o triángulos. Cuando se llama a los métodos area() y perimetro() de cada figura, se ejecuta la versión que corresponde según el tipo de figura que sea. Además, el gestor **implementa** métodos para agregar nuevas figuras, mostrarlas indicando su tipo, área y perímetro, y calcular el área total ocupada. De esta manera se pueden manejar distintos tipos de figuras de forma general gracias al polimorfismo.

Finalmente, en el main hicimos un menú para que el usuario pueda interactuar con el programa. Desde ahí se pueden agregar figuras, mostrarlas en pantalla y calcular el área total. Acá también se ve claramente el **polimorfismo**, porque todas las figuras se manejan con la referencia Figura, pero cada una responde con su propia forma de calcular el área y el perímetro. Esto permite trabajar con diferentes tipos de figuras sin necesidad de comprobar de qué clase es cada una, haciendo que el código sea más simple y fácil de mantener.

Ejercicio 5: En este ejercicio primero creamos la interfaz Vehiculo, dentro de ella podemos ver el **comportamiento** a través de sus métodos acelerar() y frenar() ya que la interfaz define qué métodos debe cumplir cualquier vehiculo, pero no cómo se implementan.

Luego creamos la clase Auto, Moto y Camión. En cada una se cumple la **herencia**, ya que todas implementan de la interfaz Vehiculo mediante el implements, lo que las obliga a sobrescribir (@Override) los métodos acelerar() y frenar() de la interfaz. Además, en estas clases se ve la **implementación**, porque cada una define cómo acelerar y frenar según el tipo de vehículo.

Después creamos la clase VehiculoFactory, que se encarga de fabricar los vehiculos según lo que el usuario elija. De esta forma, el programa no necesita saber cómo se crea cada objeto en detalle, solo le pide a la fábrica un tipo (auto, moto o camión) y esta devuelve el vehículo correspondiente. Esto hace que el código sea más ordenado y fácil de mantener, porque la lógica de creación está toda en un solo lugar y no repartida por el resto del programa.

Por otro lado creamos la clase GestorVehiculos, que se encarga de manejar todos los vehículos del sistema. En esta clase se nota el **polimorfismo**, porque aunque la lista guarde objetos de tipo Vehiculo, en realidad puede contener autos, motos o camiones. Cuando se llama a los métodos acelerar() y frenar() de cada uno, se ejecuta la acción propia de su clase, sin importar de qué tipo sea. Además, el gestor **implementa** métodos para agregar vehículos que vienen de la fábrica, mostrar los que están registrados y simular

que todos aceleren o frenen juntos. De esta forma se puede trabajar con distintos vehículos de manera general gracias al polimorfismo.

Finalmente en el main armamos el menú principal que le permite al usuario interactuar con el sistema. Desde ahí se pueden crear vehículos nuevos, mostrarlos y hacer que todos aceleren o frenen. Lo interesante es que, cuando elegimos las opciones de acelerar o frenar, se aplica el **polimorfismo** porque aunque trabajamos con una lista de tipo Vehiculo, cada objeto (auto, moto o camión) responde con su propio comportamiento. Gracias a este menú, el usuario puede probar fácilmente cómo funciona todo el sistema y ver las diferencias entre los distintos vehículos.

Ejercicio 7: En este ejercicio primero creamos la clase Libro, dentro de ella se observan los **comportamientos** propios de un libro a través de sus métodos prestar(), devolver() e isPrestado(), y a la vez su implementación, ya que cada método modifica o consulta directamente el estado interno del objeto. También se sobrescribe el método toString() heredado de Object, lo que constituye un ejemplo de polimorfismo, porque al imprimir un libro se muestra automáticamente su título y estado. Además, se cumple la **herencia**, aunque sea implícita, ya que Libro hereda de Object.

Luego creamos la clase Biblioteca, que se encarga de gestionar un conjunto de libros mediante un Map. Acá se observa la **implementación**, ya que contiene la lógica de agregar, mostrar, prestar y devolver libros. Además, la biblioteca hace uso de los métodos de Libro (prestar(), devolver() e isPrestado()) para consultar o cambiar su estado según corresponda. También se aprecia el **polimorfismo** en el método mostrarLibros(), porque al recorrer e imprimir los objetos Libro, se ejecuta automáticamente su toString() sobrescrito, mostrando el título y el estado de cada uno.

Finalmente en el main creamos un menú que le permite al usuario mostrar libros, prestar, devolver o salir. Acá no se programa la lógica interna, sino que se instancia un objeto Biblioteca y se delegan todas las acciones a sus métodos. De esta manera, el main solo envía los mensajes, sin importar cómo se realizan internamente las operaciones. Nuevamente se aplica el **polimorfismo** al mostrar los libros, ya que cada objeto Libro ejecuta su propio toString() sobrescrito y muestra correctamente su estado.

.