ITID433
Programación Avanzada
Facultad de Ingeniería Eléctrica y Electrónica

Práctica 05

Herencia

Objetivos:

Esta práctica de la asignatura persigue los siguientes objetivos:

- Practicar los conceptos relativos a herencia.
- Reforzar la comprensión sobre superclase y subclase.
- Reforzar la comprensión sobre interfaz, clase abstracta y clase anidada.

Instrucciones y Actividades:

1. Marco Teórico

La herencia en Java permite reutilizar y extender código de manera eficiente. Al aplicar superclases, subclases, clases abstractas, interfaces, sobrecarga y clases anidadas, puedes crear aplicaciones complejas y bien estructurados.

Superclase (clase padre, clase base, clase ancestro)

Una superclase es una clase base de la cual otras clases heredan, su propósito es compartir atributos y métodos comunes entre varias subclases.

Subclase (clases hija, clase derivada, clase descendiente)

Una subclase es una clase que hereda de una superclase, su propósito es extender o especializar la funcionalidad de la superclase.

Ejemplo de superclase/subclase:

```
// superclase
public class Animal {
    private String nombre;
    public void hacerSonido() {
        System.out.println("Sonido genérico");
    }
}
// subclase
public class Perro extends Animal {
    @Override
```

```
    public void hacerSonido() {
        System.out.println("Ladrar");
    }
}
```

Clase Abstracta

Es una clase que **NO** puede ser instanciada directamente y puede contener métodos abstractos (sin implementación), su propósito es proporcionar una base para que otras clases hereden y definan los métodos abstractos.

Ejemplo:

```
public abstract class Vehiculo {
    private String marca;

    public abstract void conducir();
}
```

Interfaz

Una interfaz es una colección de métodos abstractos que una clase implemente, su propósito es definir un contrato que las clases deben cumplir.

Ejemplo:

```
public interface Volador {
    void volar();
}
```

Sobrescritura de Métodos

La sobrescritura es la redefinición de un método en una subclase que ya está definido en su superclase, su propósito es modificar una implementación de un método que está definido en la superclase.

Ejemplo:

```
public class Animal {
    public void hacerSonido() {
        System.out.println("Sonido genérico");
    }
}

public class Perro extends Animal {
```

```
    @Override
    public void hacerSonido() {
        System.out.println("Ladrar");
    }
}
```

Sobrecarga de Métodos

La sobrecarga es la creación de múltiples métodos con el mismo nombre, pero con diferentes parámetros en la misma clase.

Propósito: Permitir la misma operación con diferentes tipos o números de parámetros.

Ejemplo:

```
public class Calculadora {
    public int sumar(int a, int b) {
        return a + b;
    }

    public double sumar(double a, double b) {
        return a + b;
    }
}
```

Clase Anidada

Es una clase que está definida dentro de otra clase, su propósito es agrupar clases que están estrechamente relacionadas, mejorando la encapsulación y legibilidad.

Ejemplo de clase interna:

```
public class Contenedor {
    private String mensaje = "Hola";

    public class Interna {
        public void mostrarMensaje() {
            System.out.println(mensaje);
        }
    }
}
```

2. Creación del juego 21

Para consolidar estos conceptos, los aplicaremos para construir un juego de 21. Para el juego se tendrán las siguientes clases:

- La superclase `Carta`, la cual define los atributos y métodos básicos de una carta.
- La subclase `Carta21`, que extiende `Carta` y añade funcionalidad específica para el 21.
- La clase abstracta `Jugador`, la cual define la estructura general de un jugador.
- La clase `JugadorHumano`, que es una subclase concreta de `Jugador`.
- La interfaz `Repartidor`, que define el comportamiento para manejar la baraja.
- La clase `Baraja`, que implementa la interfaz `Repartidor`. La clase anidada `CartaEspecial` que se define dentro de la clase `Baraja`, la cual permite manejar cartas especiales, como el Jocker.
- La clase `Juego21` encargada de la ejecución del juego.

1. Crea la clase `Carta`.
   La clase `Carta` tendrá dos variables de instancia para almacenar la familia (`String familia`) y el valor (`String valor`).
   La clase `Carta` será pública, mientras que los atributos serán privados.
   Agrega un constructor con parámetros que permita inicializar ambos atributos.
   Agrega los métodos *getter* para recuperar el valor y la familia.
   Sobrecarga el método `toString()` de la clase `Object` para entregar la información de la carta (el formato será `valor + " de " + familia`, por ejemplo "`2 de Corazones`"), esto permitirá que puedas pasar como argumento objetos de tipo `Carta` al método `println()`.

```
public class Carta {
    // atributos
    private String familia;
    private String valor;

    // constructor con parametros
    public Carta(String familia, String valor) {
        this.familia = familia;
        this.valor = valor;
    }

    // getters
    public String getFamilia() {
        return familia;
    }

    public String getValor() {
        return valor;
    }
```

```
    // sobrecarga del método toString()
    @Override
    public String toString() {
        return valor + " de " + familia;
    }
}
```

2. Crea una clase `Carta21` derivada de `Carta`.
   Agrega un constructor y usando `super()` llama al constructor de la clase base para inicializar sus atributos.
   Agrega el método `getPuntos()` y devuelve, con base al valor de la carta su valor numérico (es decir para As, 11; para J, Q y K 10; para el resto de cartas retorna el valor que tiene la carta). Recuerda que la clase envoltura `Integer` dispone del método `parseInt()` que puede usarse para transformar una cadena en entero.

```
public class Carta21 extends Carta {
    public Carta21(String familia, String valor)
{
        super(familia, valor);
    }

    public int getPuntos() {
        switch (getValor()) {
            case "A":
                return 11;
            case "J":
            case "Q":
            case "K":
                return 10;
            default:
                return
Integer.parseInt(getValor());
        }
    }
}
```

3. Crea una clase abstracta llamada `Jugador`.
   Para simplificar la implementación de `Jugador` se usarán listas y arreglos de listas, las listas permitirán almacenar las cartas (objetos de tipo Carta21). Para esto importa `java.util.ArrayList` (arreglo de listas) y `java.util.List` (listas).

Agrega como atributos el nombre del Jugador (`String nombre`), y un arreglo de listas con objetos de tipo `Carta21` que representen la mano del jugador, para facilitar la programación usa un objeto de tipo `ArrayList`, el cual almacene una lista de objetos de tipo `Carta21`, específicamente un `List<Carta21>` para almacenar la mano (`List<Carta21> mano`).
Agrega un constructor que acepte como argumento el nombre del jugador, e inicializa el objeto que representa la mano del jugador.
La clase abstracta será pública, los atributos serán protegidos.
Agrega el *getter* que permita obtener el nombre del jugador.
Agrega el método `agregarCartaMano()` que permita agregar una carta a la mano, es decir debe aceptar como argumento un objeto de tipo `Carta21`.
Agrega el método abstracto `calcularPuntaje()`.
Agrega el método `mostrarMano()`. Dado que sobrecargaste el método `toString()`, un objeto de la clase `Carta` puede pasarse como argumento al método `println()` de `System.out`, y se imprimirá la información usando el método `toString()`.

```
import java.util.ArrayList;
import java.util.List;

public abstract class Jugador {
    protected String nombre;
    protected List<Carta21> mano;

    public Jugador(String nombre) {
        this.nombre = nombre;
        this.mano = new ArrayList<>();
    }

    public String getNombre() {
        return nombre;
    }

    public void agregarCartaMano(Carta21 carta) {
        mano.add(carta);
    }

    public abstract int calcularPuntaje();

    public void mostrarMano() {
        for (Carta21 carta : mano) {
            System.out.println(carta);
```

```
            }
        }
    }
```

4. Agrega la clase `JugadorHumano` derivada de `Jugador`.
   Agrega el constructor con argumentos que acepte el nombre y usando `super()` invoca al constructor de la clase base para inicializar el nombre del jugador.
   Sobrecarga el método `calcularPuntaje()` para determinar el valor de la mano. Determina cuantas cartas As hay en la mano. Suma los valores de las cartas existentes en la mano. En caso de sobrepasar 21 puntos y si hay varias cartas As, reduce en 10 el valor de la mano para no superar los 21 puntos.

```
public class JugadorHumano extends Jugador {
    public JugadorHumano(String nombre) {
        super(nombre);
    }

    @Override
    public int calcularPuntaje() {
        int puntaje = 0;
        int cantAs = 0;

        for (Carta21 carta : mano) {
            puntaje += carta.getPuntos();
            if (carta.getValor().equals("A")) {
                cantAs++;
            }
        }

        while (puntaje > 21 && cantAs > 0) {
            puntaje -= 10;
            cantAs--;
        }

        return puntaje;
    }
}
```

5. Crea la interfaz `Repartidor`.
   Define el método `void barajar()`, el cual servirá para mezclar las cartas de la baraja.

Define el método `Carta21 repartirCarta()`, el cual servirá para sacar una carta de la baraja, es decir devolverá un objeto de tipo `Carta21`.

```
public interface Repartidor {
    void barajar();

    Carta21 repartirCarta();
}
```

6. Crea la clase `Baraja` la cual implemente la interfaz `Repartidor`.
   Para simplificar la implementación de `Baraja` se usarán colecciones y pilas, las colecciones permitirán tener acceso a métodos para mezclar las cartas, mientras que las pilas ayudarán a retirar una carta de la baraja al entregar la misma a un jugador para que no se repitan. Para esto importa `java.util.Collections` (colecciones) y `java.util.Stack` (pilas).
   Agrega un objeto de tipo `Stack` denominado `mazo` que permita gestionar las 52 cartas, es decir que almacene objetos de tipo `Carta21`.
   Agrega un constructor para inicializar el objeto `Stack`, en el constructor agrega un arreglo para gestionar las cuatro familias de cartas y un arreglo para gestionar los 13 posibles valores de las cartas. Añade las 52 cartas en el objeto `Stack`, para lo cual puedes iterar usando la familia y los valores, así como crear un objeto `Carta21` con la familia y valor correspondiente.
   Sobrecarga el método `barajar` y usa el método `shuffle()` de `Collections` para mezclar el contenido del objeto `mazo`.
   Sobrecarga el método `repartirCarta` y usa el método `pop()` del objeto `Stack` para retirar una carta de la baraja.
   Agrega una clase anidada estática en Baraja denominada `CartaEspecial` que se derive de `Carta21`. Dentro de esta clase agrega un constructor que llame al constructor de la clase base, usando como familia “Especial” y como valor “Jocker”, sobrecarga el método `getPuntos()` para que devuelva el valor 0.

```
import java.util.Collections;
import java.util.Stack;

public class Baraja implements Repartidor {
    private Stack<Carta21> mazo;

    public Baraja() {
        mazo = new Stack<>();
```

```
        String[] familias = {"Corazones",
"Diamantes", "Tréboles", "Picas"};
        String[] valores = {"2", "3", "4", "5",
"6", "7", "8", "9", "10", "J", "Q", "K", "A"};

        for (String familia : familias) {
            for (String valor : valores) {
                mazo.push(new Carta21(familia,
valor));
            }
        }
    }

    @Override
    public void barajar() {
        Collections.shuffle(mazo);
    }

    @Override
    public Carta21 repartirCarta() {
        return mazo.pop();
    }

    public static class CartaEspecial extends
Carta21 {
        public CartaEspecial() {
            super("Especial", "Joker");
        }

        @Override
        public int getPuntos() {
            return 0;
        }
    }
}
```

7. Implementa la lógica del juego en la clase `Juego21`.
   Importa la clase `Scanner` para poder solicitar al usuario el nombre del jugador.
   Crea 3 atributos: uno para gestionar la baraja (`Baraja baraja`), otro para gestionar a un jugador (`JugadorHumano jugador`) y otro para el juez (`JugadorHumano juez`).
   Crea un constructor el cual acepte como argumento el nombre del jugador y en el cual se inicialice la baraja, y se barajee la misma. Así también se inicialice al jugador y al juez.

Crea el método `jugar()` en el cual repartas las cartas y las asignes a la mano de cada jugador; para el juez crea una `CartaEspecial` y agrega a la mano del juez. Presenta la mano de cada jugador, así como el puntaje obtenido, y determina el ganador.

```
import java.util.Scanner;

public class Juego21 {
    private Baraja baraja;
    private JugadorHumano jugador;
    private JugadorHumano juez;

    public Juego21(String nombreJugador) {
        baraja = new Baraja();
        baraja.barajar();
        jugador                =                new
JugadorHumano(nombreJugador);
        juez = new JugadorHumano("Juez");
    }

    public void jugar() {

jugador.agregarCartaMano(baraja.repartirCarta());
jugador.agregarCartaMano(baraja.repartirCarta());

juez.agregarCartaMano(baraja.repartirCarta());

juez.agregarCartaMano(baraja.repartirCarta());

        System.out.println("Mano del jugador:");
        jugador.mostrarMano();
        System.out.println("Puntaje  del  jugador:
" + jugador.calcularPuntaje());

        Baraja.CartaEspecial     c     =     new
Baraja.CartaEspecial();
        juez.agregarCartaMano(c);

        System.out.println("Mano del juez:");
        juez.mostrarMano();
        System.out.println("Puntaje del juez: " +
juez.calcularPuntaje());

        // Lógica para determinar el ganador
```

```
        if      (jugador.calcularPuntaje()      >
juez.calcularPuntaje()                          &&
jugador.calcularPuntaje() <= 21) {

System.out.println(jugador.getNombre()     +     "
gana!");
        } else if (juez.calcularPuntaje() <= 21)
{
            System.out.println("El juez gana!");
        } else {
            System.out.println("Empate!");
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Ingresa tu nombre: ");
        String          nombreJugador          =
scanner.nextLine();

        Juego21        juego        =        new
Juego21(nombreJugador);
        juego.jugar();
    }
}
```

8. Modifica el método `jugar` para que la lógica de determinar ganador se remueva y se realice en otro método; así también, una vez asignado un par de cartas al jugador y al juez solicita al usuario si desea otra carta, si la respuesta es sí, entrega una nueva carta, si la respuesta es no, le corresponde jugar al juez, para quien debes entregar cartas hasta el valor de 17. Determina el ganador.

9. Modifica el método `getPuntos` de `CartaEspecial` para que retorne el valor 11. Indica si el juez gana con esta modificación o no.

10. Modifica la lógica del método `jugar` y la lógica para determinar ganador para que haya 3 jugadores (tú, tu compañero y el juez). Agrega un nuevo constructor que permita inicializar a los 3 jugadores. Modifica el `main` para que se use este nuevo constructor. IMPORTANTE: si modificas código asegúrate de comentar el código inicial para poder revisar que si realizaste las modificaciones.

3. Entregables

Para esta práctica debes realizar lo siguiente:

1. Completa el código solicitado.
2. Ubica la carpeta del proyecto generado, en caso de que hayas usado Eclipse o IntelliJ, elimina la carpeta `bin` o `out` y procede a comprimir las carpetas del proyecto. Este archivo comprimido (.ZIP), con el nombre cumpliendo el formato indicado en clase, es la que debes subir en el Práctica 05. En caso de que uses Replit; descarga los archivos `.java`, colócalos en una carpeta y comprímela, y ese archivo cárgalo en la plataforma.

4. Realizado por:

David Mejía N.