LTP – Ejercicios de Java

Bloque 1: Ejercicios de Herencia

1. Escribe una clase *Multimedia* para almacenar información de objetos de tipo multimedia (películas, discos, sonido en mp3, vídeos en mp4…). Esta clase contiene los siguientes atributos: *título*, *autor*, *formato* y *duración*. El formato puede ser uno de los siguientes[1]: wav, mp3, midi, avi, mov, mpg, cd de audio y dvd. El valor de todos los atributos se pasa por parámetro en el momento de crear el objeto. Esta clase tiene, además, un método consultor para devolver cada uno de los atributos y un método *toString()* que devuelve en un *String* la información del objeto. Por último, un método *equals()* que recibe un objeto de tipo *Multimedia* y devuelve *true* en caso de que el título y el autor sean iguales y *false* en caso contrario.

Solución:

Una implementación válida es la siguiente:

```
public enum Formato {
    WAV, MP3, MIDI, AVI, MOV, MPG, CDAUDIO, DVD
}

public class Multimedia {

    private String titulo;
    private String autor;
    private Formato formato;
    private double duracion;

    public Multimedia(String titulo, String autor, Formato
                        formato, double duracion) {
        this.titulo = titulo;
        this.autor = autor;
        this.formato = formato;
        this.duracion = duracion;
    }

    public String getTitulo() {
        return titulo;
    }

    public String getAutor() {
        return autor;
    }

    public Formato getFormato() {
```

[1] Para restringir un atributo a un conjunto finito de valores constantes, puedes utilizar las *clases enumeradas* de Java. Más información aquí: https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

```
        return formato;
    }


    public double getDuracion() {
        return duracion;
    }

    public String toString() {
        return "Titulo: " + titulo + " Autor: " + autor + "
            Formato " + formato + " Duracion: " + duracion;
    }

    public boolean equals(Multimedia m) {
        return titulo.equals(m.getTitulo()) &&
                              autor.equals(m.getAutor());
    }
}
```

---

2. Escribe una clase *Película* que herede de la clase *Multimedia* anterior. La clase *Película* tiene, además de los atributos heredados, un actor principal y una actriz principal. La clase debe tener dos métodos consultores para obtener los nuevos atributos y debe sobrescribir el método *toString()* para reflejar la nueva información.

Solución:

Una implementación válida es la siguiente:

```
public class Pelicula extends Multimedia {
    private String actorPrincipal;
    private String actrizPrincipal;

    public Pelicula(String titulo, String autor,
                    Formato formato, double duracion,
                    String actor, String actriz) {
        super(titulo, autor, formato, duracion);
        actorPrincipal = actor;
        actrizPrincipal = actriz;
    }

    public String getActorPrincipal() {
        return actorPrincipal;
    }

    public String getActrizPrincipal() {
        return actrizPrincipal;
    }
```

```
    @Override
    public String toString() {
        String s = "Protagonizada por: ";
        if (actrizPrincipal != null) {
            s += actrizPrincipal;
            if (actorPrincipal != null) {
                s += " y " + actorPrincipal;
            }
        } else {
            if (actorPrincipal != null) {
                s += actorPrincipal;
            } else {
                S += "¡NADIE!";
            }
        }
        return super.toString() + s;
    }

}
```

---

3. Escribe una clase *Disco*, que herede de la clase *Multimedia* ya realizada. La clase *Disco* tiene, aparte de los elementos heredados, un atributo para almacenar el género al que pertenece (con valores posibles rock, pop y ópera). La clase debe tener un método para obtener el nuevo atributo y debe sobrescribir el método *toString()* para que devuelva toda la información.

Solución:

Una implementación válida es la siguiente:

```
public enum Genero {
    ROCK, POP, OPERA
};

public class Disco extends Multimedia {
    private Genero genero;

    public Disco(String titulo, String autor,
                 Formato formato, double duracion,
                 Genero genero) {
        super(titulo, autor, formato, duracion);
        this.genero = genero;
    }

    public Genero getGenero() {
        return genero;
    }

    public String toString() {
```

```
        return super.toString() + "\nGenero: " + genero;
    }

}
```

---

4. Escribe una clase *ListaMultimedia* para almacenar objetos de tipo multimedia. La clase debe tener un atributo que sea un array de objetos *Multimedia* y un entero para contar cuántos objetos hay almacenados. Además, tiene un constructor y los siguientes métodos:

- el constructor recibe un entero por parámetro indicando el número máximo de objetos que va a almacenar.
- *talla()*: devuelve el número de objetos que hay en la lista.
- *agregar(Multimedia m)*: añade el objeto *m* al final de la lista, y devuelve *true*; en caso de que la lista esté llena, devuelve *false*.
- *Multimedia obtener(int pos)*: devuelve el objeto situado en la posición especificada.
- *String toString()*: devuelve la información de los objetos que están en la lista.

Solución:

Una implementación válida es la siguiente:

```
public class ListaMultimedia {
    private Multimedia[] lista;
    private int contador;

    public ListaMultimedia(int tallaMaxima) {
        lista = new Multimedia[tallaMaxima];
        contador = 0;
    }

    public int talla() {
        return contador;
    }

    public boolean agregar(Multimedia m) {
        if (contador == lista.length) { //Está llena
            return false;
        } else {
            lista[contador] = m;
            contador++;
            return true;
        }
    }

    public Multimedia obtener(int pos) {
        if (pos >= contador) {
            return null;
        }
```

```
        return lista[pos];
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < contador; i++) {
            s += lista[i].toString() + "\n";

        }
        return s;
    }
}
```

---

5. Escribe una aplicación donde:

- Se cree un objeto de tipo *ListaMultimedia* de tamaño 10.
- Se creen dos películas y se añadan a la lista.
- Se creen dos discos y se añadan a la lista.
- Se muestre la lista por pantalla.

Solución:

Un ejemplo de solución es el siguiente:

```
public class MainLista {
    public static void main(String[] args) {

        ListaMultimedia lista = new ListaMultimedia(10);

        lista.add(new Pelicula("Good morning, Vietnam",
                               "Barry Levinson",
                                Formato.AVI,
                                121,
                                "Robin Williams",
                                null));
        lista.add(new Pelicula("Campeones",
                               "Javier Fesser",
                               Formato.MPG,
                               100,
                               "Javier Gutiérrez",
                               "Gloria Ramos"));
        lista.add(new Disco("Christmas",
                            "Michael Bublé",
                            Formato.CDAUDIO,
                            51,
                            Genero.POP));
        lista.add(new Disco("El Vals del Obrero",
                            "Ska-p",
                            Formato.CDAUDIO,
                            50,
```

```
                                Genero.ROCK));

        System.out.println(lista.toString());
    }

}
```

---

6. Escribe una clase *Lavadora* con las siguientes características:

- Sus atributos son *precio base*, *color*, *consumo energético* (letras entre A y F), *peso* y *carga máxima*. Dichos atributos deben ser privados.
- La clase contendrá tres constructores. Uno sin parámetros que inicialice todos los atributos a sus valores por defecto; otro que reciba el precio base y el peso (el resto de atributos se inicializarán por defecto); y un último constructor con todos los atributos.
- Por defecto, el color será blanco, el consumo energético será F, el precio base es de 100 €, el peso de 5 kg y la carga de 5 kg.
- Los colores disponibles son: blanco, negro, rojo, azul y gris. Usa una clase enumerada para representarlos.

Además, la clase debe implementar los siguientes métodos:

- Métodos consultores para todos los atributos.
- comprobarPrecioBase(double precio): comprueba que el precio es válido (no es un número negativo) y, si lo es, lo devuelve. En caso contrario, devolverá el precio base por defecto. Será un método privado a invocar en el segundo y tercer constructor.
- comprobarConsumoEnergetico(char letra): comprueba que la letra es válida (está entre la A y la F) y, si lo es, la devuelve. En caso contrario, devolverá la letra por defecto. Será un método privado a invocar en el tercer constructor (el que recibe argumentos para todos los atributos).
- precioFinal(): aplica un incremento al precio base según el consumo energético, el peso y la carga. Este incremento no debe afectar al valor del atributo precioBase, simplemente se devolverá el valor incrementado como resultado del método. Los incrementos se aplican de acuerdo a las siguientes tablas:

CONSUMO ENERGÉTICO	INCREMENTO DE PRECIO
A	+100 €
B	+80 €
C	+60 €
D	+50 €
E	+30 €
F	+10 €

PESO	INCREMENTO DE PRECIO
> 80 kg	+100 €
>50 kg y <=79 kg	+80 €
>20 kg y <=49 kg	+50 €
>=0 kg y <=19 kg	+10 €

CARGA MÁXIMA	INCREMENTO DE PRECIO
>30 kg	+50 €

Solución:

Una implementación válida es la siguiente:

```
public enum Color {
    BLANCO, NEGRO, ROJO, AZUL, GRIS
}

public class Lavadora {
    private double precioBase;
    private Color color;
    private char consumoEnergetico;
    private double peso;
    private double cargaMaxima;

    public Lavadora() {
        this.precioBase = 100.0;
        this.color = Color.BLANCO;
        this.consumoEnergetico = 'F';
        this.peso = 5.0;
        this.cargaMaxima = 5.0;
    }

    public Lavadora(double precioBase, double peso) {
        this.precioBase =
             comprobarPrecioBase(precioBase);
        this.color = Color.BLANCO;
        this.consumoEnergetico = 'F';
        this.peso = peso;
        this.cargaMaxima = 5.0;

    }

    public Lavadora(double precioBase, Color color,
                    char consumoEnergetico, double peso,
                    double cargaMaxima) {
        this.precioBase =
             comprobarPrecioBase(precioBase);
        this.color = color;
        this.consumoEnergetico =
             comprobarConsumoEnergetico(consumoEnergetico);
        this.peso = peso;
        this.cargaMaxima = cargaMaxima;

    }

    public double getPrecioBase() {
```

```
        return this.precioBase;
    }

    public Color getColor() {
        return this.color;
    }

    public char getConsumoEnergetico() {
        return this.consumoEnergetico;
    }

    public double getPeso() {
        return this.peso;
    }

    public double getCargaMaxima() {
        return this.cargaMaxima;
    }

    private double comprobarPrecioBase(double precio) {
        if(precio < 0.0) {
            return 100.0;
        }
        return precio;

    }

    private char comprobarConsumoEnergetico(char letra) {
        if(!(letra >= 'A' && letra <= 'F')) {
            return 'F';
        }
        return letra;
    }

    public double precioFinal() {
        double precioFinal = precioBase;

        switch(consumoEnergetico) {
            case 'A': precioFinal += 100.0; break;
            case 'B': precioFinal += 80.0; break;
            case 'C': precioFinal += 60.0; break;
            case 'D': precioFinal += 50.0; break;
            case 'E': precioFinal += 30.0; break;
            case 'F': precioFinal += 10.0; break;
        }

        if(peso > 80.0) {
            precioFinal += 100.0;
        } else if (peso > 50.0) {
            precioFinal += 80.0;
        } else if(peso > 20.0) {
```

```
            precioFinal += 50.0;
        } else {
            precioFinal += 10.0;
        }

        if(cargaMaxima > 30.0) {
            precioFinal += 50.0;
        }

        return precioFinal;
    }

}
```

---

7. Escribe una clase *Televisor* con las siguientes características:

- Sus atributos son *precio base*, *color*, *consumo energético* (letras entre A y F), *peso*, *resolución* (en pulgadas) y si es compatible con *imagen en 3D*. Dichos atributos deben ser privados.
- La clase contendrá tres constructores. Uno sin parámetros que inicialice todos los atributos a sus valores por defecto; otro que reciba el precio base y el peso (el resto de atributos se inicializarán por defecto); y un último constructor con todos los atributos.
- Por defecto, el color será blanco, el consumo energético será F, el precio base es de 100 €, el peso de 5 kg, la resolución de 20 pulgadas y no es compatible con 3D.
- Los colores disponibles son: blanco, negro, rojo, azul y gris. Usa una clase enumerada para representarlos.

Además, la clase debe implementar los siguientes métodos:

- Métodos consultores para todos los atributos.
- comprobarPrecioBase(double precio): comprueba que el precio es válido (no es un número negativo) y, si lo es, lo devuelve. En caso contrario, devolverá el precio base por defecto. Será un método privado a invocar en el segundo y tercer constructor.
- comprobarConsumoEnergetico(char letra): comprueba que la letra es correcta (está entre la A y la F) y, si lo es, la devuelve. En caso contrario, devolverá la letra por defecto. Será un método privado a invocar en el tercer constructor (el que recibe argumentos para todos los atributos).
- precioFinal(): aplica un incremento al precio base según el consumo energético, el peso, la resolución y si emite imagen en 3D. Este incremento no debe afectar al valor del atributo precioBase, simplemente se devolverá el valor incrementado como resultado del método. Los incrementos se aplican de acuerdo a las siguientes tablas:

CONSUMO ENERGÉTICO	INCREMENTO DE PRECIO
A	+100 €
B	+80 €
C	+60 €

D	+50 €
E	+30 €
F	+10 €

PESO	INCREMENTO DE PRECIO
> 80 kg	+100 €
>50 kg y <=79 kg	+80 €
>20 kg y <=49 kg	+50 €
>=0 kg y <=19 kg	+10 €

RESOLUCIÓN	INCREMENTO DE PRECIO
>40 pulgadas	+30%

COMPATIBLE CON 3D	INCREMENTO DE PRECIO
Sí	+50 €

Solución:

Una implementación válida es la siguiente:

```
public enum Color {
    BLANCO, NEGRO, ROJO, AZUL, GRIS
}

public class Televisor {
    private double precioBase;
    private Color color;
    private char consumoEnergetico;
    private double peso;
    private int resolucion;
    private boolean imagen3D;

    public Televisor() {
        this.precioBase = 100.0;
        this.color = Color.BLANCO;
        this.consumoEnergetico = 'F';
        this.peso = 5.0;
        this.resolucion = 20;
        this.imagen3D = false;
    }

    public Televisor(double precioBase, double peso) {
        this.precioBase =
             comprobarPrecioBase(precioBase);
        this.color = Color.BLANCO;
        this.consumoEnergetico = 'F';
        this.peso = peso;
        this.resolucion = 20;
        this.imagen3D = false;
    }
```

```
public Televisor(double precioBase, Color color,
                 char consumoEnergetico, double peso,
                 int resolución, boolean imagen3D) {
    this.precioBase =
         comprobarPrecioBase(precioBase);
    this.color = color;
    this.consumoEnergetico =
         comprobarConsumoEnergetico(consumoEnergetico);
    this.peso = peso;
    this.resolucion = resolucion;
    this.imagen3D = imagen3D;
}

public double getPrecioBase() {
    return this.precioBase;
}

public Color getColor() {
    return this.color;
}

public char getConsumoEnergetico() {
    return this.consumoEnergetico;
}

public double getPeso() {
    return this.peso;
}

public int getResolucion() {
    return this.resolucion;
}

public int tieneImagen3D() {
    return this.imagen3D;
}

private double comprobarPrecioBase(double precio) {
    if(precio < 0.0) {
        return 100.0;
    }
    return precio;

}

private char comprobarConsumoEnergetico(char letra) {
    if(!(letra >= 'A' && letra <= 'F')) {
        return 'F';
    }
    return letra;
```

```
    }

    public double precioFinal() {
        double precioFinal = precioBase;

        switch(consumoEnergetico) {
            case 'A': precioFinal += 100.0; break;
            case 'B': precioFinal += 80.0; break;
            case 'C': precioFinal += 60.0; break;
            case 'D': precioFinal += 50.0; break;
            case 'E': precioFinal += 30.0; break;
            case 'F': precioFinal += 10.0; break;
        }

        if(peso > 80.0) {
            precioFinal += 100.0;
        } else if (peso > 50.0) {
            precioFinal += 80.0;
        } else if(peso > 20.0) {
            precioFinal += 50.0;
        } else {
            precioFinal += 10.0;
        }

        if(resolucion > 40) {
            precioFinal += (precioFinal * 0.3);
        }

        if(imagen3D) {
            precioFinal += 50.0;
        }

        return precioFinal;
    }

}
```

---

8. Escribe una aplicación donde:

- Se cree un array de objetos de 10 posiciones.
- Se creen 2 objetos de la clase Lavadora y otros 2 de la clase Televisor (valores arbitrarios). Estos objetos se tienen que insertar intercalados en el array (una lavadora, un televisor, una lavadora y un televisor o a la inversa).
- Se recorra el array (sólo las posiciones que tengan objetos) y se muestren los precios finales de cada uno de los objetos.

Solución:

Un ejemplo de solución es el siguiente:

```
public class PruebaApp {
    public static void main(String[] args) {

        Object[] array = new Object[10];
        int contador = 0;

        Lavadora o1 = new Lavadora();
        Televisor o2 = new Televisor();
        Lavadora o3 = new Lavadora(150,20);
        Televisor o4 = new Televisor(200,10);
        array[0] = o1;
        array[1] = o2;
        array[2] = o3;
        array[3] = o4;
        contador = 4;

        for(int i = 0; i < contador; i++) {
            if(array[i] instanceof Lavadora) {
                Lavadora l = (Lavadora) array[i];
                System.out.println("Lavadora: " +
                                   l.precioFinal());
            } else if(array[i] instanceof Televisor) {
                Televisor t = (Televisor) array[i];
                System.out.println("Televisor: " +
                                   t.precioFinal());

            } else {
                System.out.println("ERROR: Objeto no
                                   reconocido");
            }
        }

    }
}
```

---

9. Observa que tanto las *Lavadoras* como los *Televisores* comparten características comunes (atributos, métodos, etc.). Empleando la herencia, reescribe el programa de los ejercicios 6 y 7 de forma que dicha información común quede condensada en una clase *Electrodoméstico*, y se reutilice la mayor porción de código posible. No se deben poder instanciar objetos de la clase *Electrodoméstico*.

Solución:

Una implementación válida es la siguiente:

```
public enum Color {
    BLANCO, NEGRO, ROJO, AZUL, GRIS
}

public abstract class Electrodomestico {
    private double precioBase;
    private Color color;
    private char consumoEnergetico;
    private double peso;

    public Electrodomestico() {
        this.precioBase = 100.0;
        this.color = Color.BLANCO;
        this.consumoEnergetico = 'F';
        this.peso = 5.0;
    }

    public Electrodomestico(double precioBase, double peso)
    {
        this.precioBase =
             comprobarPrecioBase(precioBase);
        this.color = Color.BLANCO;
        this.consumoEnergetico = 'F';
        this.peso = peso;
    }

    public Electrodomestico(double precioBase, Color color,
                    char consumoEnergetico, double peso) {
        this.precioBase =
             comprobarPrecioBase(precioBase);
        this.color = color;
        this.consumoEnergetico =
             comprobarConsumoEnergetico(consumoEnergetico);
        this.peso = peso;
    }

    public double getPrecioBase() {
        return this.precioBase;
    }

    public Color getColor() {
        return this.color;
    }

    public char getConsumoEnergetico() {
        return this.consumoEnergetico;
    }

    public double getPeso() {
        return this.peso;
    }
```

```
    private double comprobarPrecioBase(double precio) {
        if(precio < 0.0) {
            return 100.0;
        }
        return precio;

    }

    private char comprobarConsumoEnergetico(char letra) {
        if(!(letra >= 'A' && letra <= 'F')) {
            return 'F';
        }
        return letra;
    }

    public double precioFinal() {
        double precioFinal = precioBase;

        switch(consumoEnergetico) {
            case 'A': precioFinal += 100.0; break;
            case 'B': precioFinal += 80.0; break;
            case 'C': precioFinal += 60.0; break;
            case 'D': precioFinal += 50.0; break;
            case 'E': precioFinal += 30.0; break;
            case 'F': precioFinal += 10.0; break;
        }

        if(peso > 80.0) {
            precioFinal += 100.0;
        } else if (peso > 50.0) {
            precioFinal += 80.0;
        } else if(peso > 20.0) {
            precioFinal += 50.0;
        } else {
            precioFinal += 10.0;
        }

        return precioFinal;
    }

}

public class Lavadora extends Electrodomestico {
    private double cargaMaxima;

    public Lavadora() {
        super();
        this.cargaMaxima = 5.0;
    }
```

```
    public Lavadora(double precioBase, double peso) {
        super(precioBase, peso);
        this.cargaMaxima = 5.0;

    }

    public Lavadora(double precioBase, Color color,
                    char consumoEnergetico, double peso,
                    double cargaMaxima) {
        super(precioBase, color, consumoEnergetico, peso);
        this.cargaMaxima = cargaMaxima;

    }

    public double getCargaMaxima() {
        return this.cargaMaxima;
    }

    public double precioFinal() {
        double precioFinal = super.precioFinal();

        if(cargaMaxima > 30.0) {
            precioFinal += 50.0;
        }

        return precioFinal;
    }

}

public class Televisor extends Electrodomestico {
    private int resolucion;
    private boolean imagen3D;

    public Televisor() {
        super();
        this.resolucion = 20;
        this.imagen3D = false;
    }

    public Televisor(double precioBase, double peso) {
        super(precioBase, peso);
        this.resolucion = 20;
        this.imagen3D = false;

    }

    public Televisor(double precioBase, Color color,
                    char consumoEnergetico, double peso,
                    int resolucion, boolean imagen3D) {
        super(precioBase, color, consumoEnergetico, peso);
```

```
        this.resolucion = resolucion;
        this.imagen3D = imagen3D;

    }

    public int getResolucion() {
        return this.resolucion;
    }

    public int tieneImagen3D() {
        return this.imagen3D;
    }

    public double precioFinal() {
        double precioFinal = super.precioFinal();

        if(resolucion > 40) {
            precioFinal += (precioFinal * 0.3);
        }

        if(imagen3D) {
            precioFinal += 50.0;
        }

        return precioFinal;
    }

}
```

---

10. Reescribe la aplicación del ejercicio 8 adaptándola a la nueva jerarquía de herencia.

Solución:

Un ejemplo de solución es el siguiente:

```
public class PruebaApp {
    public static void main(String[] args) {

        Electrodomestico[] array =
                                new Electrodomestico[10];
        int contador = 0;

        Lavadora e1 = new Lavadora();
        Televisor e2 = new Televisor();
        Lavadora e3 = new Lavadora(150,20);
        Televisor e4 = new Televisor(200,10);
        array[0] = e1;
        array[1] = e2;
        array[2] = e3;
```

```
        array[3] = e4;
        contador = 4;

        for(int i = 0; i < contador; i++) {
            Electrodomestico e = array[i];
            System.out.println("Elec: " + e.precioFinal());
        }

    }
}
```

---

11. Reflexiona sobre las implicaciones de adoptar la solución del ejercicio 8 o la del ejercicio 10 desde el punto de vista del mantenimiento del software. ¿Qué ocurriría si se añadiera una tercera clase que herede de *Electrodoméstico*? ¿Qué habría que modificar de ambas soluciones para que continúe funcionando igual? Observa también que en el ejercicio 10 hemos dejado de imprimir la clase del objeto que hemos encontrado, puesto que para mantener esa información tendríamos que seguir efectuando los cástings sobre los objetos del array. ¿Hay alguna manera de obtener un resultado por pantalla idéntico al del ejercicio 8 sin recurrir a polimorfismo *ad-hoc*?

Solución:

La primera solución nos obligaría a modificar la aplicación de prueba añadiendo otra cláusula if-else para comprobar si el objeto del array es una instancia de la nueva clase. En la solución que utiliza herencia, al tratarse de una subclase de *Electrodoméstico*, no habría que cambiar nada. Respecto de la impresión por pantalla del tipo de objeto sin utilizar coerción, sería posible hacerlo si trasladamos la responsabilidad a las clases de la jerarquía de herencia. Por ejemplo, se podría declarar un método abstracto *imprimirTipoObjeto* en la clase Electrodoméstico, implementarlo en las subclases *Lavadora* y *Televisor*, y llamarlo desde la aplicación de prueba para que cada objeto imprima su propio tipo. Otra opción sería sobrescribir el método *toString()*, en todas las clases del programa, de forma que cada objeto ya imprima su tipo y su precio final.

Bloque 2: Ejercicios de Interfaces

1. Escribe una interfaz, llamada *ColeccionInterfaz*, que declare los siguientes métodos:

- *estaVacia()*: método booleano que devuelve *true* si la colección está vacía y *false* en caso contrario.
- *extraer()*: devuelve y elimina el primer elemento (Object) de la colección.
- *primero()*: devuelve el primer elemento (Object) de la colección, sin eliminarlo.
- *agregar(Object)*: método booleano que añade un objeto por el extremo que corresponda, y devuelve *true* si se ha añadido y *false* en caso contrario.

A continuación, escribe una clase *Pila*, que implemente esta interfaz, utilizando para ello un array de *Object* y un contador de objetos.

Solución:

Una implementación válida es la siguiente:

```
public interface ColeccionInterface {
    boolean estaVacia();
    Object extraer();
    Object primero();
    boolean agregar(Object objeto);
}

public class Pila implements ColeccionInterface {

    private Object[] array;
    private int contador;

    public Pila(int tallaMaxima) {
        array = new Object[tallaMaxima];
        contador = 0;
     }

    @Override
    public boolean estaVacia() {
        return contador == 0;
    }

    @Override
    public Object extraer() {
        if (estaVacia()) {
            return null;
        } else {
            contador--;
            Object elemento = array[contador];
            array[contador] = null;
            return elemento;
        }
```

```
    }

    @Override
    public Object primero() {
        if (estaVacia()) {
            return null;
        } else {
            return array[contador - 1];
        }
    }


    @Override
    public boolean agregar(Object objeto) {
        if (contador == array.length) { //Pila llena
            return false;
        } else {
            array[contador] = objeto;
            contador++;
            return true;
        }
    }

    public String toString() {
        String s = "[";
        if (!estaVacia()) {
            for (int i = contador - 1; i > 0; i--) {
                s += array[i].toString() + ", ";
            }
            s += array[0].toString();
        }
        s += "]";
        return s;
    }

}
```

---

2. Escribe una clase, de nombre *PruebaPila*, en la que se implementen dos métodos:

- *rellenar()*: recibe por parámetro un objeto de tipo *ColeccionInterfaz*, y añade los números del 1 al 10.
- *imprimirYVaciar()*: recibe por parámetro un objeto de tipo *ColeccionInterfaz* y va extrayendo e imprimiendo los datos de la colección hasta que se quede vacía.
- Crea un objeto de tipo *Pila* y pruébalo.

Solución:

Un ejemplo de solución es el siguiente:

```
public class PruebaPila {

    public static void rellenar(ColeccionInterface c) {
        for (int i = 0; i <= 10; i++) {
            c.agregar(i);
        }
    }

    public static void imprimirYVaciar(
                        ColeccionInterface col) {
        while (!col.estaVacia()) {
            System.out.println(col.extraer());
        }
    }

    public static void main(String[] args) {
        Pila p = new Pila(20);
        rellenar(p);
        System.out.println("La pila es: " + p);
        imprimirYVaciar(p);
        System.out.println("Ahora la pila es: " + p);

    }

}
```

---

3. Escribe una clase *Cola* que implemente la interfaz *ColeccionInterfaz*, usando un objeto de la clase *LinkedList*.

Solución:

Una implementación válida es la siguiente:

```
import java.util.LinkedList;

public class Cola implements ColeccionInterface {
    private LinkedList lista;

    public Cola() {
        lista = new LinkedList();
    }

    @Override
    public boolean estaVacia() {
        return lista.isEmpty();
    }

    @Override
```

```
    public Object extraer() {
        return lista.remove(0);
    }

    @Override
    public Object primero() {
        return lista.get(0);
    }

    @Override
    public boolean agregar(Object o) {
        lista.add(o);
        return true;
    }

}
```

---

4. Escribe un programa para una biblioteca que contenga libros y revistas. Las **características comunes** que se almacenan tanto para las revistas como para los libros son el *código*, el *título*, y el *año de publicación*. Estas tres características se pasan por parámetro en el momento de crear los objetos. Los libros tienen además un atributo *prestado*. Los libros, cuando se crean, no están prestados. Las revistas tienen un número. En el momento de crear las revistas se pasa el número por parámetro. Tanto las revistas como los libros deben tener (aparte de los constructores) un método *toString()* que devuelve el valor de todos los atributos en una cadena de caracteres. También tienen un método que devuelve el año de publicación, y otro el código. Para prevenir posibles cambios en el programa se tiene que implementar una interfaz *Prestable* con los métodos *prestar()*, *devolver()* y prestado. La clase *Libro* implementa esta interfaz.

Solución:

Una implementación válida es la siguiente:

```
public class Publicacion {
    private String codigo;
    private String titulo;
    private int anyo;

    public Publicacion(String codigo, String titulo,
                       int anyo) {
        this.codigo = codigo;
        this.titulo = titulo;
        this.anyo = anyo;
    }

    public String getCodigo() {
        return codigo;
    }
```

```
    public String getTitulo() {
        return titulo;
    }

    public int getAnyo() {
        return anyo;
    }

    @Override
    public String toString() {
    return "Publicacion [codigo=" + codigo + ", titulo=" +
                        titulo + ", anyo=" + anyo + "]";
    }

}

public class Revista extends Publicacion {
    private int numero;

    public Revista(String codigo, String titulo, int anyo,
                    int numero) {
        super(codigo, titulo, anyo);
        this.numero = numero;
    }

    @Override
    public String toString() {
        return super.toString() + "Numero: " + numero;
    }

}

public interface Prestable {
    void prestar();
    void devolver();
    boolean prestado();
}

public class Libro extends Publicacion implements Prestable
{
    private boolean prestado;

    public Libro(String codigo, String titulo, int anyo) {
        super(codigo, titulo, anyo);
        prestado = false;
    }

    @Override
    public void prestar() {
        prestado = true;
```

```
    }

    @Override
    public void devolver() {
        prestado = false;
    }

    @Override
    public boolean prestado() {
        return prestado;
    }

    @Override
    public String toString() {
        return super.toString()+ "Libro [prestado=" +
                                   prestado + "]";
    }

}
```

---

5. Escribe una aplicación en la que se implementen dos métodos:

- *cuentaPrestados()*: recibe por parámetro un array de objetos, y devuelve cuántos de ellos están prestados.
- *publicacionesAnterioresA()*: recibe por parámetro un array de *Publicaciones* y un año, y devuelve cuántas publicaciones tienen fecha anterior al año recibido por parámetro.
- En el método *main()*, crear un array de *Publicaciones*, con 2 libros y 2 revistas, prestar uno de los libros, mostrar por pantalla los datos almacenados en el array y mostrar por pantalla cuántas hay prestadas y cuantas hay anteriores a 1990.

Solución:

Un ejemplo de solución es el siguiente:

```
public class Main {
    public static int cuentaPrestados(Object[] lista) {
        int contador = 0;
        for (Object obj : lista) { //Para cada obj en lista
            if (obj instanceof Prestable) {
                Prestable p = (Prestable) obj;
                if(p.prestado()) {
                    contador++;
                }
            }
        }
        return contador;
    }
```

```
    public static int publicacionesAnterioresA(
                        Publicacion[] lista, int anyo) {
        int contador = 0;
        for (Publicacion p : lista) {
            if (p.getAnyo() < anyo) {
                contador++;
            }
        }
        return contador;
    }

    public static void main(String[] args) {
        Publicacion[] biblioteca = {
                new Libro("CL1", "Farina", 2015),
                new Revista("CR2", "Mongolia", 2012, 1),
                new Libro("CL3", "Fahrenheit 451", 1953),
                new Revista("CR4", "Quo", 2007, 85)
                                    };

        Libro l = (Libro) biblioteca[0];
        l.prestar();
        for (Publicacion p : biblioteca) {
            System.out.println(p);
        }

        System.out.println(
            publicacionesAnterioresA(biblioteca, 1990));
        System.out.println(cuentaPrestados(biblioteca));
    }

}
```

Bloque 3: Ejercicios de Genericidad

1. Escribe una clase *Pila* genérica usando para ello un atributo del tipo *LinkedList*. La clase *Pila* tendrá los siguientes métodos:

- *estaVacia()*: devuelve *true* si la pila está vacía y *false* en caso contrario.
- *extraer()*: devuelve y elimina el primer elemento de la colección.
- *primero()*: devuelve el primer elemento de la colección
- *agregar()*: añade un objeto por el extremo que corresponda.
- *toString()*: devuelve en forma de *String* la información de la colección

Solución:

Una implementación válida es la siguiente:

```
import java.util.LinkedList;

public class Pila<E> {
    private LinkedList<E> lista;

    public Pila() {
        lista = new LinkedList<E>();
    }

    public boolean estaVacia() {
        return lista.isEmpty();
    }

    public E extraer() {
        return lista.remove(0);
    }

    public E primero() {
        return lista.get(0);
    }

    public boolean agregar(E o) {
        lista.add(o);
        return true;
    }

    public String toString() {
        return lista.toString();
    }
}
```

---

2. Implementa una pila utilizando como atributos un array genérico y un entero que cuente el número de objetos insertados. La clase se debe llamar *PilaArray* y tiene los mismos métodos que la pila del ejercicio anterior.

Solución:

Una implementación válida es la siguiente:

```
public class PilaArray<E> {
    private E[] arrayGenerico; // Array de tipo genérico
    private int contador;

    public PilaArray(int tallaMaxima) {
        //No se pueden crear arrays genéricos, pero sí
        //se puede hacer casting desde Object
        arrayGenerico = (E[]) new Object[tallaMaxima];
        contador = 0;
    }

    public boolean estaVacia() {
        return contador == 0;
    }

    public E extraer() {
        if (estaVacia()) {
            return null;
        } else {
            contador--;
            E elemento = arrayGenerico[contador];
            arrayGenerico[contador] = null;
            return elemento;
}
    }

    public E primero() {
        if (estaVacia()) {
            return null;
        } else {
            return arrayGenerico[contador - 1];
        }
    }

    public boolean agregar(E objeto) {
        if (contador == arrayGenerico.length) { //Llena
            return false;
        } else {
            arrayGenerico[contador] = objeto;
            contador++;
            return true;
        }
    }

    public String toString() {
        return arrayGenerico.toString();
    }
```

```
}
```

---

3. Escribe una clase *Matriz* genérica con los siguientes métodos:

- constructor que recibe por parámetro el número de filas y columnas de la matriz.
- *set()* recibe por parámetro la fila, la columna y el elemento a insertar. El elemento es de tipo genérico. Este método inserta el elemento en la posición indicada.
- *get()* recibe por parámetro la fila y la columna. Devuelve el elemento en esa posición. El elemento devuelto es genérico.
- *columnas()* devuelve el número de columnas de la matriz.
- *filas()* devuelve el número de filas de las matriz.
- *toString()* devuelve en forma de *String* la información de la matriz.

Solución:

Una implementación válida es la siguiente:

```
public class Matriz<E> {
    private E[][] tabla;

    public Matriz(int filas, int columnas) {
        tabla = (E[][]) new Object[filas][columnas];
    }

    public void set(int fila, int columna, E elemento) {
        tabla[fila][columna] = elemento;
    }

    public E get(int fila, int columna) {
        return tabla[fila][columna];
    }

    public int columnas() {
        return tabla[0].length;
    }

    public int filas() {
        return tabla.length;
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < tabla.length; i++) {
            for (int j = 0; j < tabla[0].length; j++) {
                s += tabla[i][j] + "\t";
            }
        }
        return s;
```

```
    }
}
```

---

4. Escribe una aplicación que:

- Cree una matriz de enteros de 4 filas y 2 columnas
- Rellénala con números consecutivos comenzando por el 1.
- Muestra por pantalla la matriz.
- Muestra por pantalla el contenido de la celda en la fila 1, columna 2

Solución:

Un ejemplo de solución es el siguiente:

```
public class MatrizMain {
    public static void main(String[] args) {

        Matriz<Integer> matriz = new Matriz<Integer>(4, 2);
        int numero = 1;

        for (int i = 0; i < matriz.filas(); i++) {
            for (int j = 0; j < matriz.columnas(); j++) {
                matriz.set(i, j, numero);
                numero++;
            }
        }

        System.out.println(matriz.toString());
        System.out.println(matriz.get(1, 2));

    }

}
```

---

5. Escribe una interfaz *ColeccionSimpleGenerica*, que como su propio nombre indica, es genérica, con los siguientes métodos:

- *estaVacia()*: devuelve true si la pila está vacía y false en caso contrario
- *extraer()*: devuelve y elimina el primer elemento de la colección.
- *primero()*: devuelve el primer elemento de la colección.
- *agregar()*: añade un objeto por el extremo que corresponda.

Solución:

Una implementación válida es la siguiente:

```
public interface ColeccionSimpleGenerica<E> {
    boolean estaVacia();
    E extraer();
```

```
    E primero();
    boolean agregar(E o);
}
```

---

6. Escribe una clase genérica *ListaOrdenada* con un tipo parametrizado *E* que sea *Comparable* (genericidad restringida). La clase debe tener lo siguiente:

- Un constructor
- *boolean add(E o)*: agrega el elemento *o* en la posición que le corresponda, garantizando que la lista se mantiene ordenada.
- *E get(int index)*: devuelve el element en la posición *index*.
- *int size()*: devuelve la cantidad de elementos actual de la lista
- *boolean isEmpty()*: devuelve true si la lista está vacía y false en caso contrario
- *boolean remove(E o)*: elimina el elemento *o* y devuelve *true*, si está en la lista. En caso contrario, devuelve *false*.
- *int indexOf(E o)*: busca el elemento *o* en la lista y devuelve su posición.
- *String toString():* imprime los elementos de la lista.

Solución:

Una implementación válida es la siguiente:

```
public class ListaOrdenada<E implements Comparable> {
    private E[] array;
    int contador;

    public ListaOrdenada(int tallaMaxima) {
        array = (E[]) new Object[tallaMaxima];
        contador = 0;
    }

    public boolean add(E o) {
        if(contador == array.length) { //Lista llena
            return false;
        }
        for (int i = 0; i < contador; i++) {
            if (o.compareTo(array[i]) < 0) {
                    for(int j = contador-1; j >= i; j++) {
                        array[j+1] = array[j];
                    }
                array[i] = o;
                contador++;
                return true;
            }
        }
        array[contador] = o;
        contador++;
        return true;
    }
```

```
    public E get(int index) {
        if(index < 0 || index > array.length) {
            return null;
        }
        return array[index];
    }

    public int size() {
        return contador;
    }

    public boolean remove(E o) {
        for(int i = 0; i < contador; i++) {
            if(o.equals(array[i])) {
                array[i] = null;
                for(int j = i+1; j < contador; j++) {
                    array[j-1] = array[j];
                }
                contador--;
                return true;
            }
        }
        return return false;
    }

    public boolean isEmpty() {
        return contador == 0;
    }

    public int indexOf(E o) {
        for(int i = 0; i < contador; i++) {
            if(o.equals(array[i])) {
                return i;
            }
        }
        return -1;
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < contador; i++) {
            s += array[i] + "\n";
        }
        return s;
    }

}
```

Dado que no se especifica cómo implementar la clase ListaOrdenada, otra solución válida es la siguiente:

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

public class ListaOrdenada<E implements Comparable> {
    private List<E> lista;

    public ListaOrdenada() {
        lista = new ArrayList<E>();
    }

    public boolean add(E o) {
        for (int i = 0; i < lista.size(); i++) {
            if (o.compareTo(lista.get(i)) < 0) {
                lista.add(i, o);
                return true;
            }
        }
        lista.add(o);
        return true;
    }

    public E get(int index) {
        return lista.get(index);
    }

    public int size() {
        return lista.size();
    }

    public boolean remove(E o) {
        return lista.remove(o);
    }

    public boolean isEmpty() {
        return lista.isEmpty();
    }

    public int indexOf(E o) {
        return lista.indexOf(o);
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < lista.size(); i++) {
            s += lista.get(i) + "\n";
        }
        return s;
    }

}
```

7. Escribe una clase, de nombre *ArrayListOrdenado*, que herede de *ArrayList*. Esta clase solo debe aceptar, como parámetro genérico de tipo, clases que implementen *Comparable*. La clase *ArrayListOrdenado* debe sobrescribir el método *add(E objeto)* para que añada los elementos en orden.

Solución:

Una implementación válida es la siguiente:

```
import java.util.ArrayList;

public class ArrayListOrdenado<E implements Comparable>
                                    extends ArrayList {
    @Override
    public boolean add(E o) {
        for (int i = 0; i < size(); i++) {
            if (o.compareTo(get(i)) < 0) {
                add(i, o);
                return true;
            }
        }
        super.add(o);
        return true;
    }

}
```

8. Escribe una interfaz genérica *Operable*, que sea genérica y que declare las cuatro operaciones básicas: suma, resta, producto y división. Cada operación se debe definir entre el objeto actual (`this`) y un parámetro `par`.

Solución:

Una implementación válida es la siguiente:

```
public interface Operable<E> {
    E suma(E par);
    E resta(E par);
    E producto(E par);
    E division(E par);

}
```