

Objetivos:

- Conceptos de OOP en Java, Herencia, Interfaces, Clases Abstractas.
- Estructuras de control, Colecciones, Iteradores.
- Casting
- Default en Interfaces, Static Imports, Notations
- Uso de patrones de diseño.

En esta práctica usaremos algunas clases definidas en las prácticas 1 y 2. Es por ello que se recomienda finalizar las prácticas anteriores antes de comenzar con esta.

Ejercicio 1.

Dada la siguiente interface:

```
public interface VideoClip {  
    /*comienza la reproduccion del video*/  
    public void play();  
    /*reproduce el clip en un bucle, un loop infinito*/  
    public void bucle();  
    /*detiene la reproducción del video*/  
    public void stop();  
}
```

- a) Realice una clase llamada **ReproductorMultimedia** que implementa la interface VideoClip.

Ejercicio 2.

- a) Cree una clase Pasajero y haga que la misma implemente la interface Frecuencia en la cual están definidos los siguientes métodos:

```
public int millas();  
    /* Retorna las millas acumuladas de ese pasajero.*/  
  
public Date ultimoViaje();  
    /*Retorna la fecha del último viaje realizado.*/
```

- b) Ahora implemente la clase vehículo que implemente la misma interface Frecuencia.
- c) Extienda el ejercicio 4 del Práctico 1 para que los choferes y los colectivos implementen la interface *Frecuencia*.

Ejercicio 3.

Se desea modelar un editor gráfico (**GraphEditor**). El mismo contiene una colección de **FiguraGeometrica** (en este caso FiguraGeometrica es una Interface), las mismas pueden ser del tipo **Rectángulo**, **Cuadrado**, **Circulo**, y **Triangulo**.

- a) Implemente en JAVA el **GraphEditor**. Tenga en cuenta que todas las figuras están guardadas en la misma colección. El editor debe permitir agregar y eliminar figuras.
- b) Extienda el **GraphEditor** para que ahora este pueda (basándose en las figuras que contiene) retornar la suma del área cubierta por cada figura. Esto es:



- c) Implemente en el **GraphEditor** el método mover. Dicho método mueve el origen de cada una de las figuras, una determinada cantidad de puntos, en la dirección que se indica en el parámetro.

Todas las figuras deben entender los siguientes mensajes:

```
public void mover(int unPunto, String dirección){  
    /*Desplaza el origen de la figura*/  
}
```

Donde **dirección** puede ser: *North, South, West, East*.

Nota: Preste atención en **cómo deberían definirse las constantes** que indican la dirección.

Realice varias pruebas del **GraphEditor** para ver que todo funciona correctamente.

Ejercicio 4.

Dada una clase abstracta denominada **Animal** de la cual derivan las clases **Gato** y **Perro**. Ambas clases redefinen el método *habla* declarada abstracta en la clase **Animal** como se muestra a continuación:

```
public abstract class Animal {  
    public abstract void habla();  
}  
  
public class Perro extends Animal{  
    public void habla() {  
        System.out.println("¡Guau!");  
    }  
}  
  
public class Gato extends Animal{  
    public void habla() {  
        System.out.println("¡Miau!");  
    }  
}
```

Se desea hacer una clase llamada **TalkAdmin** que posee animales y a cada uno de ellos “los hace” hablar.

```
public class TalkAdmin {  
    public static void main(String[] args) {  
        Gato gato=new Gato();  
        hacerHablar(gato);  
        Perro perro =new Perro();  
        hacerHablar(perro);  
    }  
    public static void hacerHablar(Animal sujeto){  
        sujeto.habla();  
    }  
}
```

- a) Implemente todas las clases mencionadas en el ejercicio
- b) Si bien ésta solución es válida, no es extensible. Por ejemplo, este mismo **TalkAdmin** nos puede servir para “hacer hablar” un reloj CuCu. Por lo tanto, una solución sería que usted implemente una interface llamada **Hablador** de la siguiente manera:

```
public interface Hablador {  
    public abstract void habla();  
}
```

- c) Luego haga que la clase **Animal** implemente dicha interface:

```
public abstract class Animal implements Hablador {  
    public abstract void habla();  
}
```

- d) Implemente la clase **RelojCucu** que implementa la interface **Hablador**.

```
public class Cucu implements Hablador {  
    public void habla() {  
        System.out.println("¡Cucu, cucu, ...!");  
    }  
}
```

- e) Pruebe varios ejemplos. ¿Que parámetro recibe ahora el método hacer hablar de la clase **TalkAdmin**? ¿Por qué?

Ejercicio 5.

En un sistema en funcionamiento tenemos una clase **Impresora** capaz de gestionar una impresora real. Entre las características de esta clase, disponemos de *trabajosEnCola*, *encendida* e *imprimiendo*, además de un método *update* encargado de actualizar la información contenida en el objeto con la situación real del aparato.

Sabiendo que el método *update* es el único capaz de cambiar el estado de la impresora, elaborar una clase **SistemaDeInformación** capaz de informar el estado de las impresoras conectadas en el Sistema.

Además de la funcionalidad antes mencionada, se desea simular la impresión documentos. Para ello se tiene la interface **Documento** la cual se define de la siguiente manera:

```
public interface Documento {  
    public void imprime();  
}
```

Es decir que los documentos que implementan dicha interface deberán implementar el método *imprime*, el cual imprime por consola con contenido del mismo. Por otro lado, la cola de *trabajosEnCola* ahora deberá ser del tipo Documento.

El sistema además deberá contar con los siguientes métodos:

```
public void imprimirTodo()  
// el cual imprime todos los documentos de la cola  
  
public void imprimir(Documento documento)  
// el cual imprime el documento que se pasa por parámetro.
```

- Modele el sistema en UML.
- Indique el patrón utilizado.
- Desarrolle el sistema en JAVA

Ejercicio 6.

Utilizando el ejercicio 6 del trabajo práctico 1 (el ejercicio del clima) se desea modelar una estación meteorológica. La misma maneja el clima actual de una ciudad determinada. Cuando la estación se crea se define la ciudad y esta no se puede cambiar. El clima actual es un objeto clima del Tp1Ej6.

Además del clima actual la estación guarda un historial de todos los climas. La estación meteorológica actualiza el clima actual de la ciudad cada 15 minutos de manera automática (no se debe modelar o hacer métodos especiales para esto por ahora). Esto significa que se crea un nuevo objeto clima, el cual pasa a ser el clima actual, con los valores correspondientes y el clima anterior pasa al historial.

La estación meteorológica debe (al menos) tener la siguiente funcionalidad:

- `public Clima climaActual()` //El cual retorna el clima actual
- `public void agregarClima(Clima unClima)` //Agrega unClima al historial
- `public void eliminarClima(Clima unClima)` //Elimina unClima del historial
- `public List<Clima> getClimas()` //Retorna todos los climas del historial
- `public void ordenarClimaPorFecha()` //Ordena el historial por fecha
- `public void ordenarClimaTemperatura()` //Ordena el historial por temperatura. Sin tener en cuenta la escala. Solo se ordena por el valor de la temperatura.

Default Methods

Ejercicio 7.

A partir de Java8 las interfaces pueden tener Default Methods.

- a) Busque en la bibliografía cuáles son los defaults methods de las interfaces.
- b) Brinde ejemplos de uso.
- c) Compare con las Interfaces comunes y describa sus ventajas

Ejercicio 8.

Modifique el Ejercicio 3 para que ahora la interface `FiguraGeometrica` implemente el **default method** `imprimir()`. El cual debe imprimir el nombre + color + área de la figura.

NOTA: Es probable que tenga que modificar / refactorizar su clases

Statics Imports

Ejercicio 9.

A partir de Java8 los *imports* pueden ser *statics*



- a) Busque en la bibliografía que son los *statics imports*.
- b) Brinde ejemplos de uso.



Streams

Ejercicio 10

Dado el siguiente código:

```
List<String> myList = Arrays.asList("a1", "b1", "c2", "c1");  
  
myList.stream()  
    .filter(s -> s.startsWith("c"))  
    .map(String::toUpperCase)  
    .sorted()  
    .forEach(System.out::println);
```

- a) ¿Qué retorna su ejecución?

Departamento de Informática y Tecnología

- b) ¿Qué son los Stream?
- c) Brinde otro ejemplo de uso de los mismos

Ejercicio 11

Dado el siguiente código:

```
List<String> lines = Arrays.asList("Java", "PHP", "C++");

private List<String> getFilterOutput(List<String> lines, String filter) {
    List<String> result = new ArrayList<>();
    for (String line : lines) {
        if (!"PHP".equals(line)) {
            result.add(line);
        }
    }
    return result;
}
```

- a) ¿Qué retorna su ejecución?
- b) Re escriba el código usando Streams

Ejercicio 12

Refactorizar los siguientes ejercicios ya realizados para usar **Streams** en lugar de **For-Each**

- a) Ej1 del Tp2
- b) Ej2 del Tp2

Ejercicio 13 (A Entregar)

- a) Realizar una monografía sobre Streams y su uso. La misma debe contener al menos la siguiente información:
 - a) ¿Qué son?
 - b) Cómo se crean / Creación de los mismos.
 - c) Operaciones intermedias
 - d) Finalizadores u Operaciones terminales
 - e) ¿A que se refiere con que los *Streams son Lazy*?
 - f) Orden de ejecución.
 - g) Buenas prácticas de uso
- b) En cada punto anterior brinde ejemplos.
- c) Las entregas son **individuales** por medio de la plataforma.

- d) El formato de las entregas debe ser PDF y debe indicar el nombre, apellido y número de alumno.
- e) Se tomará una exposición oral de la monografía. Es decir, los alumnos deberán mostrar ejemplos y explicar el funcionamiento de los mismos.