RESPUESTAS INVESTIGACIÓN #2
(Maza Alcalde Luisin Enrique)

Respuesta 01: VENTAJAS Y DESVENTAJAS DE EMPLEAR JAVA INTERFACE O ABSTRACT CLASS

Java interface:

- ✓ Las interfaces muchas veces son definidas como un tipo de contrato entre las clases que la implementen, ya que la clase que lo haga se encuentra obligada a definir los métodos abstractos que la componen.
- ✓ Las interfaces simulan la herencia múltiple ya que una clase puede implementar cualquier número de interfaces.
- ✓ Todos los métodos de una interfaz son implícitamente public abstract.
- ✓ Todas las propiedades y atributos de una interfaz son implícitamente constantes (public static final).
- ✓ Una interfaz puede heredar (extends) de una o más interfaces.
- ✓ Los tipos de las interfaces pueden ser utilizados polimórficamente.
- ✓ Una interface en java se usa para añadir mayor flexibilidad al programa.
- ✓ Proporciona mayor escalabilidad en el desarrollo del software.
- ✓ Permite desacoplar el desarrollo del software.
- ✓ Permite la reutilización y extensibilidad.

Abstract class:

- ✓ Una clase abstracta no puede ser instanciada, solo puede ser heredada.
- ✓ Si al menos un método de la clase es abstract, esto obliga a que la clase completa sea definida abstract, sin embargo la clase puede tener el resto de métodos no abstractos.
- ✓ Los métodos abstract no llevan cuerpo.
- ✓ Permite mantener el desarrollo más organizado y fácil de entender.
- ✓ Al no poder instanciar una clase abstracta nos aseguramos de que las propiedades específicas de esta, solo estén disponibles para sus clases hijas.

- ✓ Si tengo una clase que hereda de otra abstracta, estoy obligado a poner en el código, todos los métodos abstractos de la clase padre, pero esta vez serán métodos concretos y su funcionalidad o cuerpo será definido dependiendo de para que la necesite, de esa manera si tengo otra clase que también hereda del mismo padre, implementaré el mismo método pero con un comportamiento distinto.

- ✓ Se utiliza el concepto de Polimorfismo para crear objetos de una clase abstracta por medio de sus clases hijas.

Respuesta 02: DIFERENCIAS ENTRE ARRAYLIST Y HASHMAP

ARRAYLIST	HASHMAP
Implementa List Interface.	Implementa Map interface.
Almacena solo el valor de elemento y mantiene internamente los índices para cada elemento.	Almacena pares clave y valor, para cada valor debe haber una clave asociada.
Mantiene el orden de inserción.	No mantiene orden de inserción.
Permite elementos duplicados.	No permite claves duplicadas, sino valores duplicados.
Puede tener cualquier número de elementos nulos.	Permite una clave nula y cualquier número de valores nulos.
Se obtiene elemento especificando el índice.	Los elementos se recuperan especificando la clave correspondiente.

Respuesta 03: MEJORAS EN STREAM DE JAVA 12

- ✓ Un stream consiste en una secuencia de objetos a los cuales se les pueden aplicar ciertas operaciones de conjuntos (pipeline de funciones).

- ✓ Permiten código más legible, cómodo de implementar y lo más importante eficiente.

- ✓ Existen dos tipos de streams:

- **Stream:** Al aplicarse sobre una colección, genera un stream secuencial de cada uno de los elementos de una colección. Ejemplo de uso: array.stream()

- **Parallel Stream:** A diferencia del stream normal, este genera un stream paralelo, donde cada elemento no depende de otro para ser procesado (se realizan en paralelo). Ejemplo de uso: array.parallelStream()

✓ Tipos de operaciones:

- Las operaciones principales de streams se dividen en intermedias (intermediate operations) y terminales (terminal operations) según sus características.

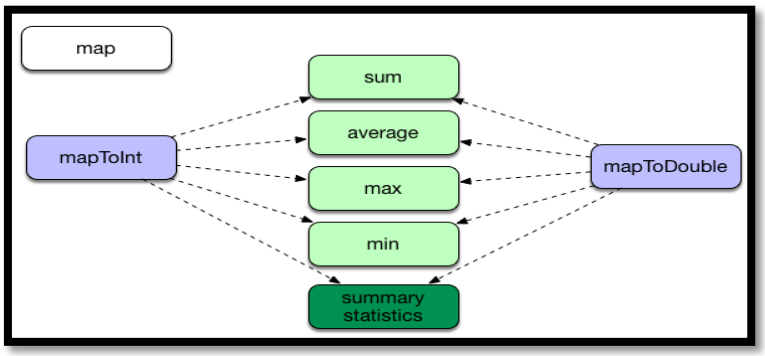
- Las operaciones intermedias(*) se ejecutan siempre de manera lazy, es decir, que en caso de ejecutarse una operación de búsqueda que por ejemplo quisiera encontrar el primer número primo de una colección, esta no debería examinar todos los valores. Al ejecutarse, su resultado es un nuevo stream con el resultado de la operación aplicada.

- Las operaciones terminales(**), a diferencia de las intermedias, debe pasar por todos los elementos del stream para generar un resultado. Cuando se ejecuta una operación de este tipo, el stream se da por concluido/cerrado y no se pueden realizar más funciones de cualquier tipo (terminales o no). Se considera este tipo de operaciones eager, por su característica de ejecutarse antes de retornar el resultado.

✓ **Map(*):** Altera los elementos en base a una función y los devuelve alterados. También permite realizar proyecciones de atributos.

El método map viene con dos métodos adicionales orientados a trabajar con datos numéricos. Estos métodos son maptoInt y mapToDouble. Si cambiamos nuestro método de map a maptoInt o maptoDouble se nos abrirá la posibilidad de acceder a métodos adicionales muy orientados a estadísticas.

```
List<Integer> result = vehiculos.stream()
        .map(v -> v.getKilometros()*2)
        .collect(Collectors.toList());
```



- ✓ **Filter(*):** Filtra los elementos a partir de la condición de la función pasada como parámetro.

```
List<Vehiculo> result = vehiculos.stream()
        .filter(v -> v.getModelo().equals(Modelo.AUDI))
        .collect(Collectors.toList());
```

- ✓ **Sorted(*):** Ordena los elementos siguiendo la condición de la función como parámetro.

```
List<Vehiculo> result = vehiculos.stream()
       .sorted((v1, v2) -> Integer.compare(v1.getKilometros(), v2.getKilometros()))
       .collect(Collectors.toList());
```

- ✓ **Distinct(*):** Retorna un stream con elementos no repetidos basándose en la comparación entre objetos con la función equals (Object.equals(object)).

```
List<Modelo> result = vehiculos.stream()
        .map(Vehiculo::getModelo)
        .distinct()
        .collect(Collectors.toList());
```

- ✓ **Peek(*):** Esta operación no realiza cambios sobre el stream y lo devuelve tal y como entra. El propósito principal consiste en hacer debugging al ejecutar cualquier otra operación, ya que permite imprimir valores de los elementos del stream.

```
List<String> result = vehiculos.stream()
        .filter(v -> v.getKilometros() > 20000)
        .peek(v -> System.out.println(v))
        .map(Vehiculo::getMatricula)
        .peek(v -> System.out.println(v))
        .collect(Collectors.toList());
```

- ✓ **Limit(*):** Limita el número de elementos que tiene como salida el nuevo stream.

```
List<Vehiculo> result = vehiculos.stream()
        .limit(5)
        .collect(Collectors.toList());
```

- ✓ **ForEach(**):** Realiza la acción de la función por parámetro para cada elemento.

```
vehiculos.parallelStream()
        .limit(5)
        .forEach(System.out::println);
```

- ✓ **Collect(**):** Ejecuta lo que denominan mutable reduction, que consiste en acumular los resultados en una Collection a medida que los va procesando en la pipeline del stream.

```
List<Vehiculo> vehiculosCollect = vehiculos.parallelStream()
        .limit(5)
        .collect(Collectors.toList());
```

- ✓ **Reduce(**):** Permite hacer una reducción de los resultados del stream, lo cual consiste en acumular el resultado en un resultado resumido de la entrada, por ejemplo, encontrar la suma de un stream de enteros. También hay otras operaciones que utilizan la operación reduce en background y son terminales, como sum().

```
Integer result = vehiculos.stream()
        .map(v -> v.getKilometros())
        .reduce(0, Integer::sum);
```

- ✓ Predicados:

    - Por definición en lógica matemática, un predicado, expresado como P: X→ {true, false}, se entiende como una función tal que su resultado es cierto o falso dependiendo del valor de sus variables.

```
public class PredicadosVehiculo {

        static Predicate<Vehiculo> modeloIgualA(Modelo modelo) {
                return v -> modelo.equals(v.getModelo());
        }

        static Predicate<Vehiculo> matriculaIgualA(String matricula) {
                return  v -> matricula.equals(v.getMatricula());
        }

        static Predicate<Vehiculo> kilometrosEntre(int desde, int hasta) {
                return v -> v.getKilometros() >= desde && v.getKilometros() <= hasta;
        }
}
```

```
vehiculos.stream()
        .filter(modeloIgualA(Modelo.BMW).and(kilometrosEntre(10000, 100000)))
        .collect(Collectors.toList());
```

```
vehiculos.stream()
        .filter(modeloIgualA(Modelo.BMW).or(kilometrosEntre(10000, 100000)))
        .collect(Collectors.toList());
```

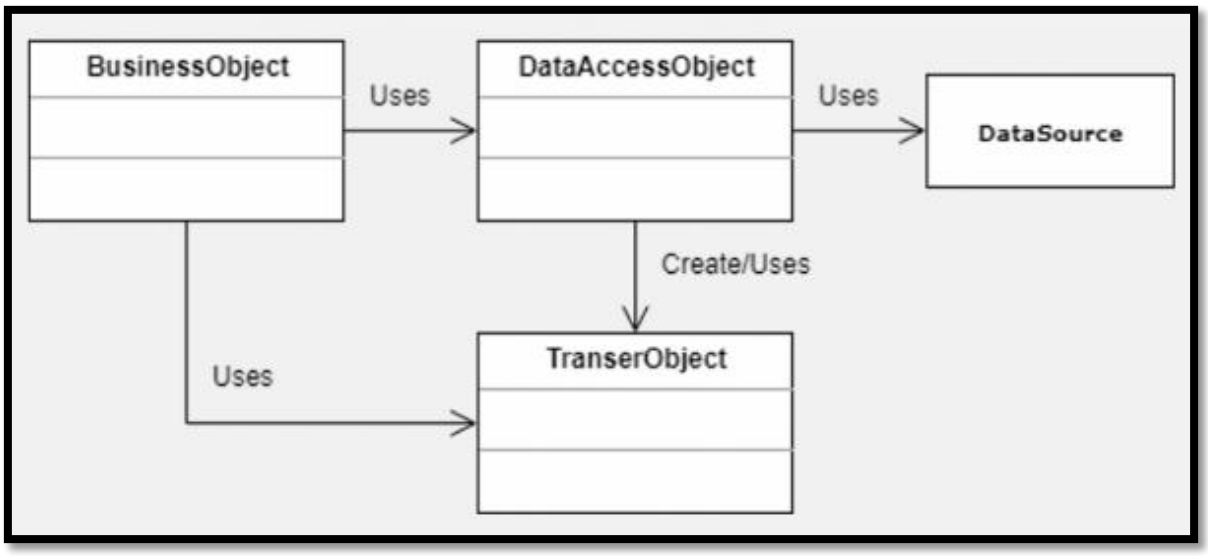
- ✓ Dado que la función filter de stream puede ser reusable por otros componentes de dominio de una aplicación, puede ser de gran utilidad crear una interfaz de tipo genérico que podamos implementar con los objetos de dominio y pasarle su predicado correspondiente.

```
public interface DomainFunctions<T> {

        default List<T> filter(List<T> list, Predicate<T> predicate) {
                return list.stream()
                        .filter(predicate)
                        .collect(Collectors.<T>toList());
        }

}

public Vehiculo implements DomainFunctions {
        ...
}
```

Respuesta 04: VENTAJAS Y UTILIDADES DEL PATRÓN DAO, SINGLETON Y BUILDER

Patrón DAO:

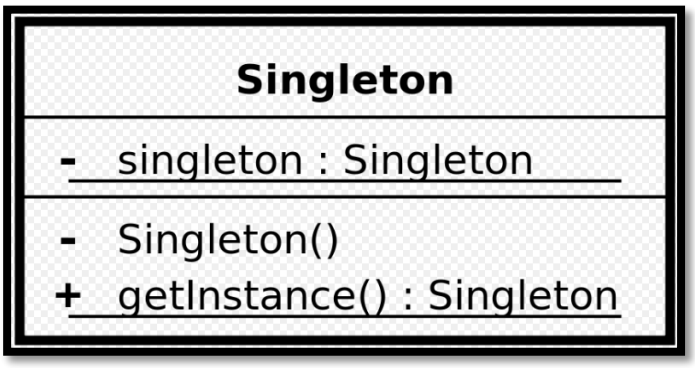
- ✓ Patrón DAO propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el patrón DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información, por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el patrón DAO para interactuar con la fuente de datos.



- ✓ Los compones que conforman el patrón son:
    - **BusinessObject:** Representa un objeto con la lógica de negocio.
    - **DataAccessObject:** Representa una capa de acceso a datos que oculta la fuente y los detalles técnicos para recuperar los datos.
    - **TransferObject:** Este es un objeto plano que implementa el patrón Data Transfer Object (DTO), el cual sirve para transmitir la información entre el DAO y el Business Service.
    - **DataSource:** Representa de forma abstracta la fuente de datos, la cual puede ser una base de datos, datos en memoria, servicios web, archivos de texto, entre otros.
- ✓ Mediante el patrón Abstract Factory podemos definir una serie de familias de clases que permitan conectarnos a las diferentes fuentes de datos.

Patrón SINGLETON:

- ✓ El patrón Singleton es utilizado limitar la creación de objetos pertenecientes a una clase. El objetivo de este patrón es el de garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella. El patrón Singleton por ejemplo, suele ser utilizado para las conexiones a bases de datos.
- ✓ Este patrón es ampliamente utilizado ya que hay multitud de ocasiones en las que se necesita que necesita compartir alguna información en la aplicación, tener un único punto de acceso a un recurso o cualquier situación en la que se necesite tener un solo objeto de una clase.
- ✓ Este patrón se implementa haciendo privado el constructor de la clase y creando un método que crea una instancia del objeto si este no existe.



Patrón BUILDER:

- ✓ El patrón Builder consiste básicamente en una clase especializada en construir instancias de otra clase que podemos hacer usable.

- ✓ En el caso de una lista larga de argumentos algunos puedan tomar valores por defecto creando métodos telescópicos (donde hay varios constructores y cada uno solo añade un nuevo argumento al anterior), en el caso de argumentos opcionales nos obliga a crear un constructor por cada combinación de argumentos, peor aún, ambas cosas se pueden producir a la vez.

- ✓ Por ejemplo, supongamos que tenemos una entidad de dominio Usuario en la que el correo electrónico es requerido siendo opcionales su nombre, apellidos teléfono o dirección. Sin usar el patrón de diseño Builder probablemente tendríamos los siguientes constructores o tener solo el último de ellos y en los no necesarios usar como valor del argumento null.

```
1 // Contructores con el problema de ser telescópicos y ser mútiples por combinación de parámetros
2 public Usuario(String email)
3 public Usuario(String email, String nombre, String apellidos)
4 public Usuario(String email, String telefono)
5 public Usuario(String email, String direccion)
6 public Usuario(String email, String nombre, String apellidos, String telefono)
7 public Usuario(String email, String nombre, String apellidos, String direccion)
8 public Usuario(String email, String telefono, String direccion)
9 public Usuario(String email, String nombre, String apellidos, String telefono, String direccion)
```

- ✓ La solución a los constructores telescópicos y combinación de argumentos es usar el patrón de diseño Builder.
- ✓ Su uso sería de la siguiente manera algo más autoexplicativa y legible que la opción de usar constructores.

```
1  package io.github.picodotdev.pattern.builder;
2
3  public class Main {
4
5    public static void main(String[] args) {
6      Usuario usuario = new UsuarioBuilder()
7          .email("nombre.apellido@gmail.com")
8          .nombre("Nombre", "Apellido")
9          .telefono("555123456")
10         .direccion("c\\ Rue el Percebe 13").build();
11   }
12 }
```

- ✓ La instancia de la clase UsuarioBuilder en su uso recoge los datos usando una API fluida, el método build es el que construye la instancia del usuario mediante el constructor con visibilidad de paquete en el que se valida que los datos al construir el objeto Usuario sean válidos, en este caso que el email es requerido.