1) Explicar el concepto de encapsulamiento en la Programacion orientada a objetos.

El concepto de encapsulamiento en la Programación Orientada a Objetos (POO) se refiere al **ocultamiento del estado interno de un objeto**, de forma que **sólo puede ser modificado a través de los mensajes** (métodos) que el objeto entiende, protegiendo así la integridad de los datos y evitando accesos indebidos

2) Comparar la herencia de estructura o atributos en Smalltalk y C++

En **Smalltalk**, la **herencia de estructura** es **total**, es decir, un objeto hereda todos los atributos de su superclase. No hay restricciones en cuanto a qué atributos se heredan.

En **C++**, la herencia de atributos es **selectiva**, ya que solo se heredan los **miembros públicos y protegidos**, mientras que los **miembros privados** no se heredan. Además, es posible que algunas funciones miembro, como los constructores, no se hereden automáticamente

3) Dada una coleccion col=(6,12,1,9,7), Como se puede obtener una nueva coleccion ordenada de mayor a menor?

Para obtener una nueva colección ordenada de mayor a menor a partir de `col = (6,12,1,9,7)` en un entorno como Smalltalk, puedes usar el método `sort:` con un bloque de comparación. El bloque debe devolver `true` cuando el primer argumento sea mayor que el segundo. Un ejemplo sería:

```
col := #(6 12 1 9 7).
sortedCol := col asSortedCollection: [:a :b | a > b].
```

Esto retornará una nueva colección `sortedCol` con los elementos ordenados de mayor a menor, es decir:

`#(12 9 7 6 1)`.

4) Que diferencia hay entre un metodo y un mensaje?

La diferencia clave entre un **método** y un **mensaje** en la Programación Orientada a Objetos es la siguiente:

Un **mensaje** es una **solicitud de ejecución** que se envía a un objeto, indicando que debe realizar una acción. Contiene el nombre del método a invocar y, opcionalmente, los argumentos.

Un **método** es la **implementación** que responde a ese mensaje. Es el bloque de código que define qué ocurre cuando un objeto recibe un mensaje en particular.

5) Explicar la ventaja de escribir un programa usando Programacion Funcional, con respecto al Paradigma Imperativo.

La **principal ventaja** de la **Programación Funcional** frente al **Paradigma Imperativo** es la **transparencia referencial** y la **ausencia de efectos secundarios**, lo que significa que las funciones en la programación funcional siempre producen el mismo resultado dado el mismo conjunto de entradas, sin modificar el estado global del programa. Esto facilita:

Depuración y pruebas: Al no haber cambios en el estado global, es más fácil razonar sobre el comportamiento del programa y probarlo.

Paralelismo: La ausencia de efectos secundarios permite que las funciones se ejecuten en paralelo sin conflictos, mejorando el rendimiento en sistemas concurrentes.

Modularidad: Las funciones son tratadas como bloques autónomos, lo que facilita su reutilización y composición

6) Explicar que son los tipos genericos en programacion funcional. Ejemplificar:

Los **tipos genéricos** en programación funcional permiten escribir funciones o estructuras de datos que pueden operar sobre **cualquier tipo de dato**, haciéndolas más flexibles y reutilizables. En lugar de definir una función para cada tipo de dato, los genéricos permiten definir una sola función que pueda manejar diferentes tipos.

Ejemplo: En Haskell, los tipos genéricos se representan con variables de tipo. Aquí hay un ejemplo de una función genérica que devuelve el primer elemento de una lista, sin importar el tipo de elementos que contenga:

```
firstElement :: [a] -> a
firstElement (x:_) = x
```

En este ejemplo, `a` es un tipo genérico. La función `firstElement` puede operar con listas de cualquier tipo, como una lista de enteros `[1, 2, 3]` o una lista de caracteres `['a', 'b', 'c']`.

7) Dar un ejemplo de funcion definida con analissi por casos.

Un ejemplo de función definida con análisis por casos en programación funcional es el cálculo del valor absoluto de un número. En Haskell, se puede definir de la siguiente manera:

```
absoluto :: Int -> Int
absoluto x
  | x < 0     = -x
  | otherwise = x
```

En este caso, la función `absoluto` evalúa el valor de `x` y, dependiendo de si es negativo o no, aplica el caso correspondiente:

Si `x` es menor que 0, devuelve `-x` (el opuesto de `x`).

En caso contrario (`otherwise`), devuelve `x` tal cual.

8) Detallar como esta formada en la Programacion logica una base de conocimiento.

En la **Programación Lógica**, una **base de conocimiento** está formada por un conjunto de **hechos** y **reglas** que describen relaciones entre entidades. Estos elementos son expresiones lógicas que permiten realizar deducciones a partir de los datos conocidos.

Componentes de una base de conocimiento:

Hechos: Son enunciados siempre verdaderos que describen relaciones entre los objetos del dominio. Por ejemplo:

hombre(juan).

Aquí se declara que *Juan* es un hombre.

Reglas: Expresan relaciones más complejas y permiten hacer deducciones. Se componen de una **cabeza** (lo que se quiere demostrar) y un **cuerpo** (las condiciones que deben cumplirse para que la cabeza sea verdadera). Por ejemplo:

mortal(X) :- humano(X).

sta regla dice que *X* es mortal si *X* es humano.

Consultas: Son preguntas que se hacen a la base de conocimiento para validar si una afirmación es verdadera o para obtener valores que satisfacen ciertas condiciones. Por ejemplo:

?- mortal(juan).

Esta consulta pregunta si *Juan* es mortal, y se evalúa a partir de los hechos y reglas.

9) En que situaciones el motor de inferencias responde Yes? Ejemplifique.

El **motor de inferencias** en programación lógica responde **Yes** cuando puede **demostrar que una consulta es verdadera** utilizando los hechos y reglas de la base de conocimiento. Esto ocurre cuando encuentra una secuencia de deducciones lógicas que satisfacen la consulta.

Ejemplo:

Si en la base de conocimiento tenemos los siguientes hechos y reglas:

```
hombre(juan).
hombre(pedro).
mortal(X) :- humano(X).
humano(X) :- hombre(X).
```

Y hacemos la consulta --> ?- mortal(juan).

El motor de inferencias responde **Yes** porque puede deducir lo siguiente:

hombre(juan) es un hecho verdadero.

Como **hombre(juan)**, entonces **humano(juan)** (por la regla `humano(X) :- hombre(X)`).

Como **humano(juan)**, entonces **mortal(juan)** (por la regla `mortal(X) :- humano(X)`).

Dado que todas las condiciones se cumplen, la consulta es verdadera y el motor responde **Yes**.

10) Dar un ejemplo de un hecho y una regla.

Ejemplo Hecho --> gato(michi).

Este hecho declara que *michi* es un gato. No tiene condiciones, simplemente es un enunciado que siempre es verdadero.

Ejemplo Regla --> animal(X) :- gato(X).

Esta regla establece que *X* es un animal si *X* es un gato. Aquí, la cabeza de la regla es `animal(X)` y el cuerpo es `gato(X)`. Para que `animal(X)` sea verdadero, el motor debe poder demostrar que `gato(X)` es verdadero.

Consulta: Si consultamos --> ?- animal(michi). El motor de inferencias verificará que `gato(michi)` es un hecho verdadero y responderá Yes.