1. **[2p]** Dado el siguiente código Java,
   a) Dibujar un diagrama de clases que muestre todas las clases/interfaces involucradas (salvo `Object`) y sus relaciones. No es necesario incluir los atributos y métodos, a menos que sean relevantes para mostrar las relaciones entre las clases.
   b) Justificar: ¿La salida del programa es determinística?
   c) El `println` en la línea 16 de `Main.java` nunca se ejecuta. ¿Por qué? ¿En qué estado queda cada uno de los hilos?

Productor.java

```
1  public class Productor extends Thread {
2    ArrayBlockingQueue<String> queue;
3
4    public Productor(ArrayBlockingQueue<String> queue) {
5      this.queue = queue;
6    }
7
8    @Override
9    public void run() {
10     for (int i = 0; i < 5; i++) {
11       String item = "Item " + i + " desde " +
         Thread.currentThread().getName();
12       queue.put(item);
13     }
14   }
15 }
```

Mundo.java

```
1  public class Consumidor extends Thread {
2    ArrayBlockingQueue<String> queue;
3
4    public Consumidor(ArrayBlockingQueue<String> queue) {
5      this.queue = queue;
6    }
7
8    @Override
9    public void run() {
10     while (true) {
11       String item = queue.take();
12       System.out.println("Consumido: " + item);
13     }
14   }
15 }
```

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3        ArrayBlockingQueue<String> q = new
         ArrayBlockingQueue<>(10);
4        Productor productor1 = new Productor(q);
5        Productor productor2 = new Productor(q);
6        Consumidor consumidor = new Consumidor(q);
7
8        productor1.start();
9        productor2.start();
10       consumidor.start();
11
12       productor1.join();
13       productor2.join();
14       consumidor.join();
15
16       System.out.println("Listo.");
17   }
18 }
```

**Criterios de corrección:**

- En el diagrama de clases verificar principalmente que usen bien las flechas de relaciones (composición, generalización, etc). No es tan importante la diferencia entre composición y agregación.
- No es necesario que escriban los nombres de atributos y métodos.
- Verificar que usen los términos correctos de OOP en la justificación.
- La salida no es determinística, porque el orden de ejecución de los hilos productores depende del scheduler.
- El `println` nunca se ejecuta porque el hilo consumidor se queda bloqueado en el `take()` esperando más elementos, ya que los productores ya terminaron y no van a agregar más elementos a la cola. El hilo principal queda bloqueado en el `join()` del consumidor.

2. **[1p]** Considerar el siguiente código. Identificar un principio de diseño que se viola, indicar cuál es y justificar. Proponer una solución para evitar la violación del principio (no es necesario escribir el código).

```java
public class App {
  public void mostrarMenu(String usuario, String contrasena) {
    Cuenta c = Cuenta.buscar(usuario);
    if (c == null || !c.getContrasena().equals(contrasena)) {
      System.out.println("Usuario o contraseña incorrectos");
      return;
    }
    System.out.println("Bienvenido " + usuario);
  }

  public void mostrarDatosPersonales(String usuario, String contrasena) {
    Cuenta c = Cuenta.buscar(usuario);
    if (c == null || !c.getContrasena().equals(contrasena)) {
      System.out.println("Usuario o contraseña incorrectos");
      return;
    }
    System.out.println("Perfil de " + usuario + ":")
    // ...
  }
}

public class Cuenta {
  public String getUsuario() { ... }
  public String getContrasena() { ... }

  public static Cuenta buscar(String usuario) { ... }
}
```

**Criterios de corrección:**

- Verificar que identifiquen correctamente el principio de diseño violado (TDA o DRY, pero puede ser otro si está bien justificado).
- De nuevo, verificar que usen los términos correctos en la justificación y en la solución propuesta.

3. **[2p]** Escribir en Prolog la regla `repetido/2` que dado un valor y una lista determina si el valor aparece al menos 2 veces en la lista.

```
?- repetido(c, [a, b, c, d, b]).
false.
?- repetido(b, [a, b, c, d, b]).
true.
```

Se permite utilizar la regla estándar `member/2`:

```
?- member(d, [a, b, c]).
false.
?- member(b, [a, b, c]).
true.
```

**Criterios de corrección:**

Solución:

```
repetido(X, [X|T]) :- member(X, T).
repetido(X, [_|T]) :- repetido(T, X).
```

4. **[2p]** Explicar en téminos de programación funcional el funcionamiento del siguiente código en Clojure. Para cada una de las funciones involucradas, describir su propósito y cómo contribuye al resultado final, e indicar si son puras/impuras y si son de orden superior.

```
(reduce * (take 4 (repeat 2)))
;; resultado: 16
```

**Criterios de corrección:**

- `(repeat 2)` (pura) arma una secuencia perezosa de `(2 2 2 2 ...)`
- `(take 4 ...)` (pura) toma los primeros 4 elementos de la secuencia, resultando en `(2 2 2 2)`
- `(reduce * ...)` (pura, orden superior) aplica la función de multiplicación `*` acumulativamente a los elementos de la secuencia, calculando `2 * 2 * 2 * 2`, que da como resultado `16`.

5. **[2p]** Dada la siguiente expresión lambda,
   a) Identificar las variables libres y ligadas.
   b) Mostrar los pasos de reducción usando orden normal (β-redex más externa desde la izquierda) hasta llegar a la forma normal (si existe). Aplicar reducciones α según sea necesario para evitar conflictos de nombres.

```
((λx.(x (x w))) ((λy.(y v)) w))
```

**Criterios de corrección:**
https://projectultimatum.org/cgi-bin/lambda?t=((%CE%BBx.(x%20(x%20w)))%20((%CE%BBy.(y%20v))%20w))&r=&m=normal%20order

- No es necesario que dibujen el árbol sintáctico, solo identificar las variables libres y ligadas.
- El resultado en forma normal es `w v (w v w)`.

6. **[1p]** Dado el siguiente código JavaFX, explicar brevemente el funcionamiento del programa, en términos de programación GUI y orientada a eventos. ¿Qué sucede cuando el usuario interactúa con la interfaz? ¿Cómo se logra este comportamiento?

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 200, 200);

        scene.setOnMouseClicked(e -> {
            Circle pelota = new Circle(10, Color.BLUE);
            pelota.setLayoutX(e.getSceneX());
            pelota.setLayoutY(e.getSceneY());

            root.getChildren().add(pelota);
            AnimationTimer t = new AnimationTimer() {
                double velocidad = 0;

                @Override
                public void handle(long l) {
                    if (pelota.getLayoutY() > scene.getHeight()) {
                        root.getChildren().remove(pelota);
                        this.stop();
                    }
                    pelota.setLayoutY(pelota.getLayoutY() + velocidad);

                    velocidad += 0.001;
                }
            };
            t.start();
        });

        stage.setScene(scene);
        stage.show();
    }
}
```

**Criterios de corrección:**
Verificar que usen los términos correctos en la explicación:

Cuando el usuario hace clic en la escena, se lanza el evento MouseClicked, y se ejecuta el handler registrado con `setOnMouseClicked`, que:

- crea un círculo azul en la posición del clic.
- agrega el círculo al grupo raíz para que sea visible.
- inicia un `AnimationTimer` que, en cada cuadro de animación ejecuta el handler, que:
  - actualiza la posición vertical del círculo según su velocidad.
  - incrementa la velocidad para simular aceleración (gravedad).
  - cuando el círculo cae fuera de la escena, se elimina del grupo raíz y se detiene el timer.