1. Dado la función `countChar()` en la que pretendemos contar el número de veces que un determinado carácter aparece en un determinado `String` (pasados ambos por parámetro) ¿Qué error hemos cometido en su implementación?

```
public static int countChar(String text, char c) {
    int sum = 0;
    for (int i = 0; i < text.length(); i++) {
        if (text.charAt(i) == c)
            sum++;
    }
    return sum;
}
```

A. No comprobamos que el `String` pasado por parámetro pueda ser `null`, lo que provocará que se lance una `NullPointerException` al llamar a `text.length()`.

B. Hacemos una comparación de caracteres usando `==` cuando deberíamos usar la función `equals` de la siguiente forma: `text.charAt(i).equals(c)`

C. El bucle lanza un `StringIndexOutOfBoundsException` al salirse del rango de índices del `String`.

D. No hay ningún error y el código compila y funciona correctamente.

2. **Dada la siguiente clase `Box` que NO redefine el método `equals`:**

```
public class Box {
    private int value;
    public Box(int value) { this.value = value; }
}
```

¿Cuál es el resultado de compilar y ejecutar el siguiente código?

```
Box x = new Box(7);
Box y = new Box(7);

if (x == y) System.out.println("Identical");
else System.out.println("NOT Identical");

if (x.equals(y)) System.out.println("Equal");
else System.out.println("NOT Equal");
```

A. Imprime "`Identical`" y "`Equal`"

B. Imprime "`NOT Identical`" y "`Equal`"

C. Imprime "`Identical`" y "`NOT Equal`"

D. Imprime "`NOT Identical`" y "`NOT Equal`"

3. ***Comparado con* la visibilidad `public`, `protected` y `private` de Java, el especificador de *visibilidad por defecto* (*package*) es...**

A. Menos restrictivo que `public`.

B. Más restrictivo que `public`, pero menos restrictivo que `protected`.

C. Más restrictivo que `protected`, pero menos restrictivo que `private`.

D. Más restrictivo que `private`.

4. **Dada el interfaz `OperationInterface` y el enumerado `Operation`, indica cuál de las siguientes sentencias es cierta...**

```
interface OperationInterface {
    double calculate(double first, double second);
}

public enum Operation implements OperationInterface {
    PLUS { public double calculate(double x, double y){return x + y;} },
    MINUS { public double calculate(double x, double y){return x - y;} },
    TIMES { public double calculate(double x, double y){return x * y;} },
    DIVIDE { public double calculate(double x, double y){return x / y;} };
}
```

A. El código no es correcto porque un enumerado no puede implementar interfaces.

B. El código no es correcto porque un enumerado puede implementar interfaces, pero solo si estos interfaces no incluyen en su declaración ningún método abstracto que haya que implementar.

C. El código no es correcto, solo debería haber un única implementación de `calculate` dentro del enumerado. Esta operación tendría que tener una instrucción `switch` para decidir qué operación en concreto hacer.

D. El código es correcto.

5. **Dado el registro de Java definido a continuación, señala la falsa:**

```
public record Box(int value) { }
```

A. `Box` incluye un constructor público `Box(int value)`.

B. `Box` incluye implementaciones para los métodos `equals` y `hashCode` basados en el atributo `value`.

C. `Box` incluyen un método de lectura `getValue()` y un método de escritura `setValue(int value)` sobre el atributo `value`.

D. `Box` incluye una implementación de `toString()` que devuelve una representación en `String` de todos los atributos del registro.

6. **¿Cuál de las siguientes sentencias sobre clases abstractas es cierta?**

A. Una clase abstracta debe tener definido al menos un método abstracto.

B. Una clase abstracta puede contener métodos NO abstractos.

C. Una clase abstracta no puede tener definidos constructores porque no se pueden crear instancias de la misma usando el operador `new`.

D. Una clase abstracta puede tener definidos atributos, pero serán implícitamente `public`, `static` y `final` (constantes de clase).

7. **¿Cuál de estas afirmaciones sobre la sobrecarga paramétrica es falsa?**

A. Es muy utilizada en los constructores para ofrecer distintas maneras de crear objetos.

B. Es posible sobrecargar métodos heredados de superclases (por ejemplo, cuando en la superclase existe un método con dos parámetros y la subclase lo sobrecarga añadiendo más parámetros).

C. Se considera que es un tipo de polimorfismo aparente (*ad hoc*).

D. Sobrecargar un método requiere el uso del la anotación `@Overload` para que compile.

8. **¿Soporta Java *duck typing* (tipado del pato)?**

   A. Sí, porque Java es un lenguaje con tipado dinámico.

   B. Sí, a través del uso de clases abstractas.

   C. Depende del código en concreto, a veces sí y a veces no.

   D. No, porque Java es un lenguaje con tipado estático.

9. **`Comparable` es un interfaz del API de Java que se define tal y como se muestra a continuación. La clase `SomeClass` alberga un parámetro de tipo `T` genérico e implementa `Comparable` para hacer uso del método `compareTo` y comparar el valor interno con otro que se pasa por parámetro. ¿Cómo modificarías la linea `1` de la declaración de la clase `SomeClass` para que implemente correctamente el interfaz `Comparable`?**

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

```
public class SomeClass { // 1
    public T value;

    @Override
    public int compareTo(T o) { /* ... */ }
}
```

   A. `public class SomeClass implements Comparable`

   B. `public class SomeClass<T>implements Comparable`

   C. `public class SomeClass implements Comparable<T>`

   D. `public class SomeClass<T>implements Comparable<T>`

10. **Dado el siguiente código ¿cuál es el resultado de compilarlo y ejecutarlo?**

```
interface Flyable { void fly(); }

abstract class Bird implements Flyable {
    public void fly() { System.out.println("Default fly"); }
}

class Chicken extends Bird {
    public void fly() { System.out.print("Cannot fly"); };
}

class BirdsFarm {
    public static void main(String[] args) {
        Bird b = new Chicken();
        Flyable f = b;
        f.fly();
    }
}
```

   A. *Muestra* `"Default fly"`.

   B. *Muestra* `"Cannot fly"`.

   C. *Error de compilación porque no podemos meter un objeto creado como* `Chicken` *en una variable de tipo* `Flyable`, *ya que* `Chicken` *no implementa* `Flyable`.

   D. *Error de ejecución porque al llamar al método* `fly()` *de* `Flyable`, Java comprueba que este *método está sin implementar y lanza una excepción.*

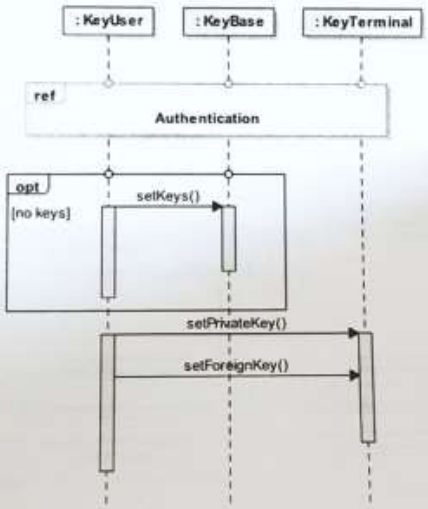
11. **En UML, cuando el nombre de un método aparece subrayado, significa que ese método es...**

A. público
B. estático
C. final
D. abstracto

12. **En la asignatura hemos visto tres cosas llamadas "*Composición*". Indica la falsa.**

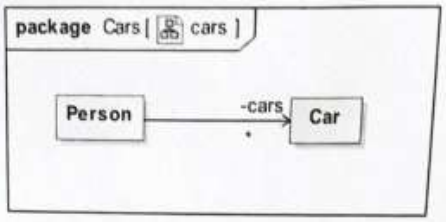
A. La *composición* orientada a objetos que define relaciones TIENE_UN y ocurre cuando un objeto contiene otros objetos.
B. La *composición* UML que define relaciones todo-parte con una idea de pertenencia fuerte.
C. El principio de *composición* que indica que es mejor tener un objeto compuesto de otros objetos pequeños y no un objeto compuesto de otro objeto grande.
D. El patrón *composición* que se utiliza para componer objetos en estructuras de árbol que representan jerarquías todo-parte.

13. **Dado el siguiente diagrama de secuencia ¿Cuál de las siguientes afirmaciones es falsa?**



A. La caja identificada como `KeyUSer` representa a un objeto (no una clase) asociado a una línea discontinua que representa su línea de vida
B. El fragmento con el operador `ref` indica que son interacciones definidas en otro lugar
C. El fragmento con el operador `opt` indica una interacción opcional que sólo ocurre si la *guardia* es cierta
D. `setPrivateKey()` y `setForeignKey()` son operaciones definidas en la clase `KeyUser`

14. ¿Cuál es la implementación correcta del siguiente diagrama de clases UML?



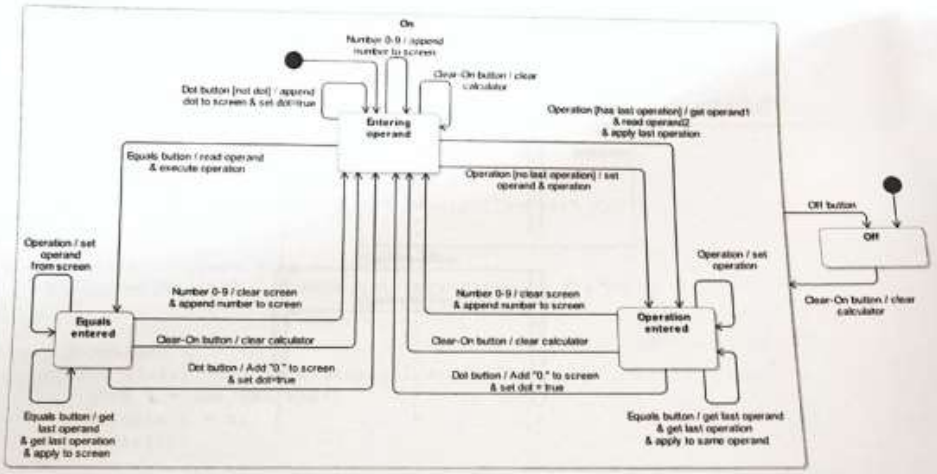
A.
```
class Person { private List<Car> cars; }
class Car { }
```

B.
```
class Person { private List<Car> cars; }
class Car { private Person owner; }
```

C.
```
class Person { List<Car> cars; }
class Car { }
```

D.
```
class Person { private List<Car> cars; }
class Car { private List<Person> owners; }
```

15. Dada la siguiente imagen que muestra un diagrama de estados de una calculadora, indica cuál de las siguientes afirmaciones es falsa



A. Cuando se inicia la calculadora está está en el estado `Off`.

B. En cualquier momento y estado, si ocurre el evento `Off button` la calculadora pasará al estado `Off`.

C. Si la calculadora está en `Off` y sucede el evento `Clear-On button` la calculadora pasará al estado `Entering operand`.

D. Transitaremos del estado `Entering Operand` al estado `Equals entered` si ocurre cualquiera de estos tres eventos: `Equals button`, `read operand` y `execute operation`.

16. **El Principio de Responsabilidad Única indica que... (señala la falsa)**

A. Cada objeto debe tener una responsabilidad única que esté enteramente encapsulada en la clase.

B. La clase deberá tener sólo una razón para cambiar.

C. El objetivo es crear clases con baja cohesión.

D. Hay que evitar las *clases Dios* que lo hacen todo en un programa.

17. **¿Qué principio SOLID incumplen las nuevas clases selladas (*sealed classes*) de Java?**

A. Principio de Responsabilidad Única.

B. Principio Abierto-Cerrado.

C. Principio de Sustitución de Liskov.

D. Principio de Inversión de la Dependencia.

E. Principio de Segregación de Interfaces.

18. **El siguiente código, en el cual a la clase `Client`, se le *inyecta* una dependencia mediante la clase `Service`. Y sabiendo que en realidad `Service` es un interfaz que será implementado por distintas clases implementadoras ¿Qué principio de diseño SOLID estamos siguiendo?**

```
public class Client {
    // Internal reference to the service used by this client
    private Service service;
    // Constructor injection
    Client(Service service) {
        // Save the reference to the passed-in service inside this client
        this.service = service;
    }
    // Setter injection
    public void setService(Service service) {
        this.service = service;
    }
    // Method within this client that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

A. Principio de Responsabilidad Única.

B. Principio Abierto-Cerrado.

C. Principio de Sustitución de Liskov.

D. Principio de Inversión de la Dependencia.

E. Principio de Segregación de Interfaces.

19. **"Una precondición sólo puede sustituirse por otra precondición más débil y una post-condición sólo puede sustituirse por otra postcondición más fuerte". Esta afirmación corresponde al...**

A. Principio de Subcontratación.

B. Principio de Inversión de la Dependencia.

C. Principio de Responsabilidad Única.

D. Principio Abierto-Cerrado.

20. Dadas las clases `Person` y `ManageClient` que vemos a continuación. Qué *principio de diseño* está incumpliendo `ManageClient` en el método `clientFullName()`

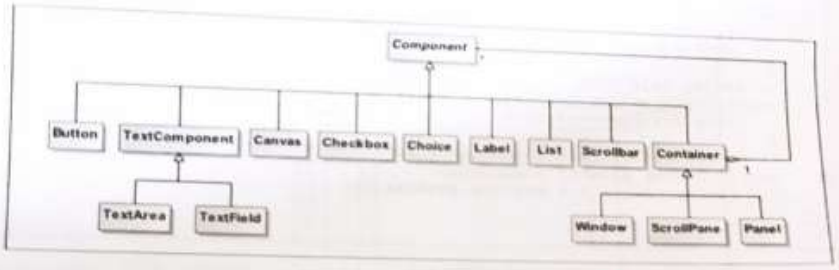
```
public class Person {
    public String name;
    public String middleName;
    public String Surname;
}
```

```
public class ManageClient {
    private final Person person;
    public ManageClient(Person person) {
        this.person = person;
    }

    public int clientFullName() {
        return person.name + " " + person.middleName + " " + person.surname;
    }
}
```

A. Principio de Sustitución de Liskov
B. Principio Abierto-Cerrado
C. Principio de Hollywood
(D) Principio *Tell, Don't Ask*

21. La librería Abstract Window Toolkit (AWT) de Java tiene la siguiente estructura ¿Qué patrón de diseño representa?



A. Inmutable B. Instancia única C. Estrategia D. Estado (E) Composición F. Iterador G. Observador H. Adaptador I. Fachada J. Método Factoría K. Constructor L. Método Plantilla

22. **En el *push model* del patrón observer... (señala la falsa)**

A. El sujeto manda a los observadores información detallada acerca de lo que ha cambiado.
B. La comunicación es más eficiente ya que no se fuerza a descubrir a los observadores qué es lo que ha cambiado.
C. Sujeto y observadores no son tan independientes como en el modelo *pull*.
(D) Es más fácil integrar nuevos observadores con necesidades completamente diferentes que en el modelo *pull*.

23. **Cuál de las siguientes frases describe correctamente el funcionamiento del patrón Iterador.**

A. Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

B. Permite a un objeto modificar su conducta al cambiar su estado interno.

C. Permite definir una dependencia "uno a muchos" entre objetos de tal forma que, cuando el objeto cambie de estado, todos sus objetos dependientes sean notificados y actualizados automáticamente.

D. Provee de un interfaz unificado para un conjunto de clases en un subsistema que hace que dicho subsistema sea más fácil de utilizar.

24. **Dado el siguiente código, qué principio de diseño estamos incumpliendo...**

```
public class SongList {
    private List<Song> songs = new ArrayList<>();
    private int playingSong;

    public void insertSong(int index, Song song) {
        if (index < 0 || index >= songs.size())
            throw new IllegalArgumentException();
        songs.add(index, song);
    }
    public void removeSong(int index) {
        if (index < 0 || index >= songs.size())
            throw new IllegalArgumentException();
        songs.remove(index);
    }
    public void selectPlayingSong(int index) {
        if (index < 0 || index >= songs.size())
            throw new IllegalArgumentException();
        playingSong = index;
    }
}
```

A. DRY

B. KISS

C. YAGNI

D. No incumple ningún principio de diseño.

25. **Dados los siguientes códigos referentes a patrones de diseño, indica cuál representa el patrón Constructor.**

A.
```
public enum Number {ACE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,JACK,QUEEN,KING}
public enum Suit {SPADES, HEARTS, CLUBS, DIAMONDS}
public record Card(Number number, Suit suit) { }
```

B.
```
public final class Integer extends Number {
    private int value;
    public Integer(int value) { this.value = value; }
    public int intValue() { return value; }
}
```

C.
```
for(Enumeration e = collection.getEnumeration(); e.hasMoreElements(); )
    System.out.println(e.nextElement());
```

D.
```
CurrencyConverter cc1 = CurrencyConverter.incomingCurrency("USD")
                            .outgoingCurrency("EUR")
                            .build();
cc1.convert(50.00);
```