En el diagrama UML, el comportamiento por defecto del método AbstractPieza.isValidad() es devolver falso. Por tanto, esta implementación del método en la clase AbstractPieza es correcta:

Public boolean isValidad(){ return false; }

**FALSO**

En Java, si un método no abstracto f() definido en una superclase se sobrescribe es una de sus subclases, se puede invocar el método de la superclase desde la sublase mediante la instrucción super.f().

**VERDADERO**

Dado el diagrama de clases de la figura 1 el siguiente código de Java contiene un error de compilación (suponiendo que ninguno de los métodos invocados declara lanzar excepciones):

AbstractPieza a = new AbstractPieza();

a.setCasilla(null);

System.out.println(a.getValor());

**VERDADERO**

Dado el diagrama de clases de la figura 1 el siguiente código en Java contiene un error de compilación (Suponiendo que ninguno de los métodos invocados declara lanzar excepciones):

AbstractPieza a = new superPieza();

a.setCasilla(null);

a.usaSuperpoderere(); // esta no tira

**VERDADERO**

El bloque finally de Java se ejecuta incluso si se lanza una excepción dentro de su bloque try correspondiente

**VERDADERO**

Todas las clases que representan excepciones en Java tienen a la clase Object como una de sus superclases

**VERDADERO**

En Java, un objeto de clase Class es creado en tiempo de ejecución cada vez que se crea un nuevo objeto en el programa.

**FALSO**

Los constructores no pueden invocarse mediante reflexión en java, pero cualquier otro tipo de método si puede ser invocado.

**FALSO**

El paso de mensajes es una característica opcional de los lenguajes orientados a objetos

**FALSO**

La refactorización sustituir condicional con polimorfismo contribuye positivamente a cumplir el principio Abierto/Cerrado (Open-Closed Principle).

**VERDADERO**

Una interfaz puede implementar una interfaz

**FALSO**

Una clase abstracta no puede tener constructores

**FALSO**

En Java solo las clases pueden ser genéricas, no así las interfaces

**FALSO, pueden ser genéricas tanto las clases como la interfaces**

Para usar reflexión en Java debemos conocer en tiempo de compilación el nombre de las clases que queremos manipular

**FALSO**

La inversión de control den los frameworks es posible gracias al enlace dinámico de métodos

**VERDADERO**

La refactorización nunca produce cambios en las interfaces de la clase

**FALSO**

Cuando se aplica correctamente, el principio de responsabilidad única (Single Responsibility Principe) conduce a diseñar con mayor acoplamiento

**FALSO**

La forma canonica de una clase varia dependiendo del lenguaje de programación que se utilice

**VERDADERO**

En Java, al implementar una composición, la copia defensiva nos ayuda a prevenir que existan referencias a los objetos ‘parte’ que sean externas al objeto ‘todo’

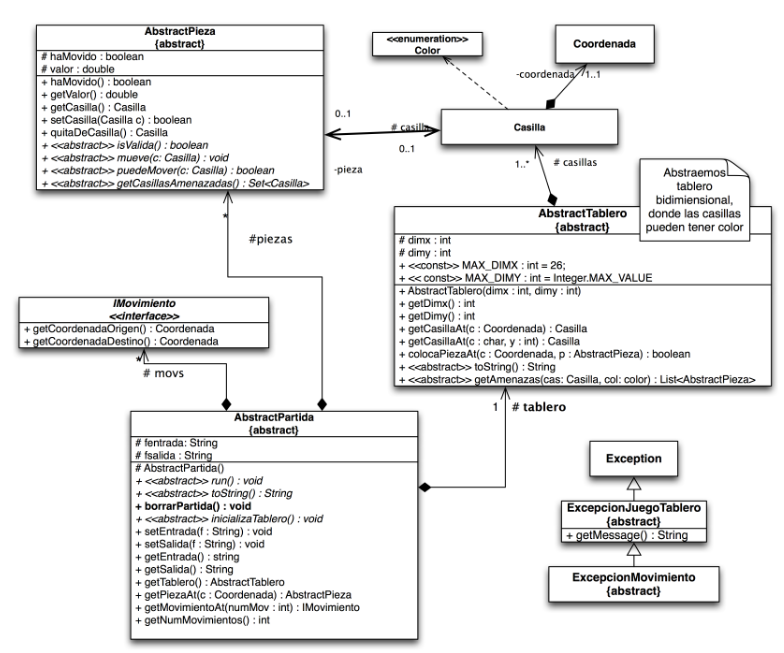
**VERDADERO**

Los lenguajes de programación soportan el reemplazo o refinamiento como una forma de sobrecarga o sobrescritura, pero no hay ningun lenguaje que proporcione ambas técnicas (por ejemplo, Java solo soporta reemplazo y C++ solo soporta refinamiento)

**FALSO**

Un lenguaje puede combinar tipado estático en algunas construcciones del lenguaje y tipado dinamico en otras

**VERDADERO**



En el diagrama de clases, la relación de AbstractTablero a Casilla es unaria, No así la relación entre Casilla y AbstractPieza

**FALSO**

La clase Color del Diagrama UML es una interface de Java

**FALSO**

En el diagrama UML, el método mueve() de la clase AbstractPieza tiene enlace dinamico

**VERDADERO**

A partir del diagrama UML, podemos deducir que diferentes objetos AbstractTablero pueden compartir objetos de tipo Casilla a través de la relación Casillas.

**FALSO**

De las subclases de AbstractPieza del diagrama UML constituyen una herencia disjunta y completa sólo si un objeto de esta jerarquía solo puede ser un PiezaNormal o SuperPieza y nada más, pero no los dos a la vez

**VERDADERO**

La clase ExcepcionJuegoTablero en el diagrama UML tiene una relación de dependencia con AbstractPartida porque usa información proporcionada por esta última

**FALSO, la clase Excepcion Juego Tablero es independiente ya que se hereda (extends) de Exception**

Dado el Diagrama UML, el siguiente constructor de copia de AbstractTablero realiza una copia profunda de un objeto AbstractTablero, (Suponemos que casillas esta implementado como un List <Casilla>):

Public AbstractTablero(AbstractTablero otro){

Super(otro); //FALLA

Dimx = otro.dimx; dimy = otro.dimy; // SE TIENE QUE ACCEDER CON GETTERS

Casillas = new ArrayList <Casilla>();

For (Casilla casilla : otro.casillas)

Casillas.add(casilla);// ESTO SE TENDRIA Q HACER USANDO EL CONSTRUCTOR DE COPIA

}

**FALSO**

Esta implementación del método AbstractTablero.Inicializa() del diagrama UML no compilara:

Public static void inicializa(){

dimx = 0;

dimy = 0;

casillas = null;

}

**FALSO**

En el ejemplo de código siguiente, la instrucción que pretende insertar una PiezaNormal en la lista produce un error de compilación:

¿?? Extends AbstractPieza plist = new ArrayList<Piezaormal>();

…add(new PiezaNormal());

**FALSO**

La clase AbstractPieza del diagrama de clases, en caso de disponer de un constructor por defecto, la podríamos instanciar usando reflexión con el código Java.

forName(“AbstractPieza”).newInstance()

**FALSO**