PREGUNTAS:

1. La forma canoníca en C++ incluye el constructor copia, el destructor, el operador asignación, el constructor por defecto y cualquier otro constructor parametrizado. F
2. En JAVA la forma canoníca incluye constructor, equals, hashCode, toString y clone. F
3. Tato la herencia protegida como la privada permiten a una clase derivada acceder a las propiedades privadas de la clase base. F
4. Independientemente del tipo de herencia la clase base siempre podrá acceder a lo público, protegido y default heredado pero no a lo privado. V
5. Una clase abstracta se caracteriza por declarar todos sus métodos de instancia como abstractos. F
6. En JAVA la clase deberá contener al menos un método abstracto para ser declarada como tal en C++ no. F
7. La siguiente sentencia class S {public final void f(){}} constituye una interfaz en JAVA. F
8. La siguiente sentencia interfaz S {void f();} constituye una interfaz en C++. F
9. Desde un método de una clase derivada nunca puede invocar un método implementad o con idéntica signatura de su clase. F
10. Desde un método de una clase derivada nunca puede invocarse un método implementado en esta con el mismo nombre que en la clase base. F
11. En Java los métodos de instancia con polimorfismo puro pero no abstracto tienen enlace dinámico. V
12. Una operación de clase solo puede acceder directamente a atributos de clase. V
13. Una operación de instancia puede acceder directamente a atributos de clase y de instancia. V
14. Una operación de clase no es una función miembro de la clase F.
15. En la misma clase podemos definir constructores de con distinta visibilidad. V
16. El modificador static es correcto para JAVA pero no para C++ cuando lo usamos con constructores. F
17. Un buen diseño OO se caracteriza por un alto acoplamiento y una baja cohesión. F
18. Mediante la herencia de implementación heredamos la implementación de los métodos de la clase base pero no la interfaz de esta. F
19. La genericidad es un tipo de polimorfismo. V
20. El polimorfismo es un tipo de genericidad. F
21. En JAVA y en C++ todas las clases son polimórficas. F
22. El downcasting en JAVA y en C++ es siempre dinámico. F
23. Si la conversión, downcasting, es fuera de la jerarquía de herencia JAVA dará error de ejecución. F
24. El downcasting implica deshacer el principio de sustitución. V
25. En JAVA si no se captura una excepción lanzada por un método da error la compilación. F
26. La instrucción throw en JAVA solo permite lanzar objetos que son de la clase throwable o clases derivada de esta. V
27. Uno de los objetivos del tratamiento de errores mediante excepciones es el manejo de errores del resto del código. V
28. Si no se captura una excepción por un método, el programa no advierte que ha ocurrido error y conti núa su ejecución que llamo a este. F
29. En JAVA. Siempre es obligatorio especificar que excepciones verificadas (checked exceptions) lanza un método mediante una clausula throws tras la lista de argumentos. V
30. Si se produce una excepción el método que la provoca se cancela y se continúa la ejecución en el método que llamo a este. F
31. Si se produce una excepción en un constructor el objeto se construirá con los valores por defecto. F
32. Todas las excepciones son checked exception salvo las runtime que son uncheked exception. V
33. La cláusula throws de un método incluirá todas las excepciones uncheked exception que puedan producirse en este y no estén dentro del bloque try catch que les capture. F
34. El orden de las excepciones en los bloques catch no es relevante. F
35. Podemos poner un bloque finally sin poner bloques catch. V
36. El bloque finally solo se ejecutará si se produce alguna excepción en el bloque try al que este asociado. F
37. IllegalArgumentException, ArrayIndexOutOfBoundsException, ClassCastException y IOExceptio son excepciones del tipo runtimeException y por tanto no es necesario capturarlas ni indicar throws en el método en el que se provoquen. F
38. La genericidad se considera una característica opcional de los lenguajes. V
39. La genericidad se considera una característica opcional de los lenguajes OO. V
40. No se puede derivar una clase genérica de una clase no genérica. F
41. En JAVA no podemos crear interfaces genéricas. F

42. En los métodos genéricos solo podremos usar los métodos definidos en Object. V
43. En la genericidad restringida solo podremos usar los métodos de Object al implementar los métodos. F
44. La API de reflexión de JAVA incluye métodos para obtener la signatura de todos los métodos. V
45. La reflexión permite que un programa obtenga información sobre sí mismo en tiempo de ejecución. V
46. Para usar reflexión en JAVA hemos de conocer el nombre de las clases den tiempo de compilación. F
47. En JAVA el concepto de meta-clase se presenta con la clase Class. V
48. Con el uso de la reflexión solo podemos invocar métodos de instancia. F
49. La reflexión solo es útil para trabajar con componentes JavaBeans. F
50. La reflexión es demasiado compleja para usarla en aplicaciones de propósito general. F
51. La reflexión reduce el rendimiento de las aplicaciones. F
52. La reflexión no puede usarse en aplicaciones certificadas con estándar 100% Pure Java. F
53. Mediante reflexión podemos saber cuáles son las clases derivadas de una clase dada. F
54. Mediante reflexión no podemos saber cuál es el método que se está ejecutando en un determinado momento. V
55. Podemos usar reflexión para encontrar un método heredado (solo hacia arriba) y reducir código condicional. V
56. La refactorización debe hacerse siempre apoyándonos en un conjunto de tests completo y robusto. V
57. Una clase con gran número de métodos y atributos es candidata a ser refactorizada. V
58. Los métodos grandes (con muchas instrucciones) son estructuras que sugieren la posibilidad de una refactorización. V
59. En la refactorización se permite que cambie la estructura interna de un sistema software aunque varié su comportamiento externo. F
60. Un ejemplo de refactorización seria mover un método arriba o abajo en la jerarquía de herencia. V
61. Los frameworks no contienen implementación alguna, únicamente un conjunto de interfaces que deber ser implementados por el usuario del framework. F
62. Un framework invoca mediante enlace dinámico a nuestra implementación de interfaces propi os de framework. V
63. Hibernate es un framework para congelar en memoria el estado de nuestra aplicación. F
64. JDBC es un framework de JAVA que usan los fabricantes de sistemas de gestión de bases de datos para ofrecer un acceso estandarizado a las bases de datos. V
65. El usuario de un framework implementa al componente declarado de los interfaces de framework mediante herencia de implementación. F
66. El usuario de un framework implementa el comportamiento declarado en los interfaces del framework mediante herencia de interfaz. V
67. Un framework es un conjunto de clases cuyos métodos invocamos para que realicen unas tareas a modo de caja negra. F
68. En JAVA el acceso a bases de datos se hace con librerías propietarias de SGBD cuyos interfaces no tienen ningún estándar. F
69. Para poder utilizar un framework, es necesario crear clases que implementen todas las interfaces declaradas en el framework. V
70. Una librería de clases proporciona una funcionalidad completa, es decir, no requiere que el us uario implemente o herede nada. V
71. El polimorfismo es una forma de reflexión. F
72. En el proceso de diseño de un sistema de software se debería intentar aumentar el acoplamiento y la cohesión. F
73. Cuando diseñamos sistema OO las interfaces de las clases que diseñamos deberían estar abiertas a la extensión y cerradas a la modificación. V
74. Todo espacio de nombre define su propio ámbito, distinto de cualquier otro. V
75. Una colaboración describe como un grupo de objetos trabaja conjuntamente para realizar una tarea. V
76. En el diseño mediante tarjetas CRC utilizamos una tarjeta para cada clase. V
77. Una tarjeta CRC contiene el nombre de una clase, su lista de responsabilidades y su lista de colaboradores. V
78. La robustez de un sistema software es un parámetro de calidad intrínseco. F
79. El principio abierto-cerrado indica que un componente software debe estar abierto a su extensión y cerrado a su modificación. V
80. Con el diseño OO es perfectamente posible y deseable hacer uso de variables globales. F
81. En el diseño por contrato son dos componentes fundamentes las pre y pos condiciones. V

82. Los métodos definidos en una clase derivada nunca pueden acceder a las propiedades privadas de una clase base. V
83. Una clase abstracta se caracteriza por no tener definido ningún constructor. F
84. La siguiente clase: class S {public: S(); S(const S &s); virtual ~S();}; constituye una interfaz en C++. F
85. Desde un método de una clase derivada nunca puede invocarse a un método implementado con la idéntica signatura de una de sus clases base. F
86. Los métodos virtuales son métodos abstractos. F
87. Los métodos abstractos son métodos con enlace dinámico. V
88. Los constructores de las clases abstractas son métodos con enlace dinámico. F
89. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase. F
90. Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica. F
91. Un método tiene polimorfismo puro cuando devuelve una variable polimórfica. F
92. En C++ no podemos hacer sobrecarga de operadores para tipos predefinidos. V
93. En C++, si no se define un constructor sin argumentos explícitamente, el compilador proporciona uno por defecto. F
94. En la misma clase, podemos definir constructores con distinta visibilidad. V
95. Si no se captura una excepción lanzada por un método, el programa no advierte que ha ocurrido algún error y continúa su ejecución normalmente. F
96. La instrucción throw permite lanzar como excepción cualquier tipo de dato. V
97. Las funciones genéricas no se pueden sobrecargar. F
98. Dada una clase genérica, se pueden derivar de ella clases no genéricas. V
99. De una clase abstracta no se pueden crear instancias, excepto si se declara explícitamente algún constructor. F
100. Una clase interfaz no puede tener atributos de instancia. Una clase abstracta sí puede tenerlos. V
101. La herencia de interfaz se implementa mediante herencia pública. V
102. La herencia pública permite a los métodos definidos en una clase derivada acceder a las propiedades privadas de la clase base. F
103. Una clase abstracta se caracteriza por declarar al menos un método abstracto. V
104. La siguiente clase: class S {public: Object +o;}; constituye una clase interfaz en C++. F
105. Los métodos abstractos siempre tienen enlace dinámico. V
106. Los constructores siempre son métodos virtuales. F
107. Un método tiene polimorfismo puro cuando tiene como argumentos al menos una variable polimórfica. V
108. Un atributo declarado con visibilidad protegida en una clase A es accesible desde clases definidas en el mismo espacio de nombres donde se definió A. F
109. Una de las características básicas de una lengua orientada a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes. V
110. Una operación de clase sólo puede ser invocada mediante objetos constantes. F
111. En C++, si no se define ningún constructor, el compilador proporciona por defecto uno sin argumentos. V
112. Una clase interfaz no puede tener instancias. V
113. Una clase abstracta siempre tiene como clase base una clase interfaz. F
114. No se puede definir un bloque catch sin su correspondiente bloque try. V
115. Tras la ejecución de un bloque catch, termina la ejecución del programa. F
116. Una variable polimórfica puede hacer referencia a diferentes tipos de objetos en diferentes instantes de tiempo. V
117. El downcasting estático siempre es seguro. F
118. El principio de sustitución implica una coerción entre tipos de una misma jerarquía de clases. V
119. La sobrecarga basada en ámbito permite definir el mismo método en dos clases diferentes. V
120. Una operación de clase no puede tener enlace dinámico. V
121. La herencia protegida permite a los métodos de la clase derivada acceder a las propiedades privadas de la clase base. F
122. La siguiente clase en C++: class S {public: virtual ~S()=0;}; define una interfaz. V
123. Los métodos virtuales siempre tienen enlace dinámico. F
124. Los constructores siempre tienen enlace dinámico. F
125. En C++, un atributo de la clase debe declararse dentro de la clase con el modificador static. V
126. Un método tiene polimorfismo puro cuando devuelve una variable polimórfica. F
127. Dada la siguiente definición de clase en C++: class TClase {public: TClase (int dim); private: int var1;}; La instrucción TClase c1; no da error de compilación e invoca al constructor por defecto. F

128. Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes. V
129. Hablamos de encapsulación cuando agrupamos datos junto con las operaciones que pueden realizarse sobre esos datos. V
130. En C++ los constructores se pueden declarar como métodos virtuales. F
131. En C++, es obligatorio especificar qué excepciones lanza una función mediante una cláusula throw tras la declaración de la función. F
132. Una clase interfaz no debe tener atributos de instancia. Una clase abstracta sí puede tenerlos. V
133. Un mensaje tiene cero o más argumentos. Por el contrario, una llamada a procedimiento/función tiene uno o más argumentos. F
134. La interpretación de un mismo mensaje puede variar en función del receptor del mismo y/o del tipo de información adicional que lo acompaña. V
135. En una agregación, la existencia de un objeto-parte depende de la existencia del objeto-todo que lo contiene. F
136. Una forma de mejorar el diseño de un sistema es reducir su acoplamiento y aumentar su cohesión. V
137. Los iniciadores en C++ tienen el formato nombreAtributo(valor) [o nombre_atributo(valor), otra pregunta] y se colocan entre la lista de argumentos y el cuerpo de cualquier método, permitiendo asignar valores a los atributos de instancia de la clase en la que se define dicho método. F
138. Un atributo estático ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de cualquier clase derivada de ella. F
139. La herencia múltiple se produce cuando de una misma clase base se heredan varias clases derivadas. F
140. Un atributo protegido en la clase base es también protegido en cualquier clase que derive de dicha base, independientemente del tipo de herencia utilizado. F
141. Cuando se crea un objeto de la clase D de que deriva de una clase B, el orden de ejecución de los constructores es siempre B() D(). V
142. Una clase abstracta es una clase que no permite instancias de ella. V
143. Una interfaz es una clase abstracta con al menos un método de instancia abstracto. F
144. Un método de clase (estático) no puede tener enlace dinámico. V
145. Un método virtual en C++ siempre tiene enlace estático. F
146. La llamada a un método sobrescrito se resuelve en tiempo de compilación. F
147. Los métodos definidos en una clase genérica son a su vez genéricos. V
148. La instrucción throw (en C++) sólo permite lanzar objetos de la clase exception o de clases derivadas de ella. F
149. En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación de instancia de los objetos pasados como parámetro. F
150. En una composición de un objeto componente puede formar parte de más de un compuesto. F
151. Los destructores en C++ pueden aceptar cualquier número de parámetros. F
152. Los TAD's representan una perspectiva orientada a datos, mientas que los objetos reflejan una perspectiva orientada a servicios. V
153. Un atributo privado en la clase base no es directamente accesible en la clase derivada, independientemente del tipo de herencia utilizado. V
154. Un constructor de copia acepta cualquier tipo de modificador (static, const, public, etc). F
155. Una clase es una especificación abstracta de una estructura de datos y de las operaciones que se pueden realizar con ella. F
156. Una operación constante sólo puede ser invocada por un objeto constante. F
157. El puntero this es un puntero constante al objeto que recibe el mensaje. V
158. Implementar la forma canónica ortodoxa de una clase es una condición necesaria (aunque no suficiente para controlar que el valor de un atributo de clase que cuenta el número de instancias de dicha clase esté siempre en un estado consistente. F
159. La existencia de una relación todo-parte entre dos clases implica necesariamente que el objeto todo maneja la creación/destrucción de los objetos parte. F
160. La forma canónica de la clase está formada por el constructor, el constructor de copia, el destructor y el operador de asignación. V
161. Si la clase no proporciona un constructor sin parámetros, el compilador en C++ genera uno de oficio. F
162. Tanto composición como herencia son mecanismos de reutilización del software. V
163. Un atributo de clase debe declararse dentro de la clase con el modificador const. F

164. Un atributo de clase público puede ser accedido desde fuera de la clase a través de un objeto de la clase, un puntero o referencia al mismo o mediante el nombre de la clase seguido del operador de ámbito. V
165. Una clase derivada puede añadir nuevos métodos/atributos propios de la clase derivada, pero no modificar los métodos heredados de la clase base. F
166. Una interfaz es la definición de un protocolo para cierto comportamiento, sin especificar la implementación de dicho comportamiento. V
167. Una colaboración describe como un grupo de objetos trabaja conjuntamente para realizar una tarea. V
168. En el diseño mediante tarjetas CRC, utilizamos una tarjeta por cada clase. V
169. La sobreescritura es una forma de polimorfismo. V
170. El principio de segregación de interfaz indica que el código cliente no debe ser forzado a depender de interfaces que no utiliza. V
171. La inversión de control en los frameworks es posible gracias al enlace dinámico de métodos. V
172. El usuario de un framework implementa el comportamiento declarado en los interfaces del framework mediante herencia de implementación. F
173. La instanciación mediante reflexión de un objeto de la clase Rectángulo del paquete pack se realiza así: Class.forName("pack", "Rectangulo"); F **((FORMA CORRECTA: class.forName("pack.Rectangulo");))**
174. En Java el enlace por defecto de métodos de instancia es estático. F
175. En Java no se pueden derivar clases genéricas de otras clases genéricas. F
176. En Java las sentencias de un bloque 'finally' solamente se ejecutan cuando se ha producido una excepción y no la hemos capturado en un bloque 'catch'. F
177. En el paradigma orientado a objetos, un objeto siempre es instancia de alguna clase. V
178. El método invocado por un objeto en respuesta a un mensaje viene siempre determinado, entre otras cosas por la clase del objeto receptor en tiempo de compilación. F
179. En el paradigma orientado a objetos, un programa es un conjunto de objetos que se comunican mediante el paso de mensajes. V
180. El siguiente código en Java define una interfaz: interface S {} V
181. Sea un método llamado glue(), sin argumentos, implementado en una superclase y sobrescrito en una de sus subclases. Siempre podremos invocar a la implementación del método en la superclase desde la implementación del método en la subclase usando la instrucción super.glue(); V
182. Los métodos abstractos siempre tienen enlace dinámico. V
183. Dado un objeto, mediante reflexión no se puede obtener la lista de todos los métodos declarados en su clase, tanto públicos como privados. F
184. Cuando diseñamos sistemas orientados a objetos las interfaces de las clases que diseñamos deberían estar abiertas a la extensión y cerradas a la modificación. V
185. La refactorización nunca produce cambios en las interfaces de las clases.F
186. Las sentencias 'switch' son un caso de código sospechoso (código con mal olor). V
187. El código duplicado es un caso de código sospechoso en el que se aconseja el uso de técnicas de refactorización para eliminarlo. V
188. La existencia de una sólida colección de pruebas unitarias es una precondición fundamental para la refactorización. V
189. "Los métodos que usan referencias a clases base deben ser capaces de usar objetos de clases derivadas sin saberlo" es una posible formulación del principio de inversión de dependencias. F
190. El enlace de la invocación a un método sobrescrito se produce en tiempo de ejecución en función del tipo del receptor del mensaje. V
191. this es un ejemplo de variable polimórfica en Java. V
192. En Java el downcasting siempre se realiza en tiempo de ejecución. V
193. En Java, un atributo de clase debe declararse dentro de la clase con el modificador static. V
194. En Java, gracias a la sobrecarga de operadores podemos crear nuevos operadores en el lenguaje. F
195. Si en una clase no se declara, implícita o explícitamente, un constructor por defecto, no se pueden crear instancias de esa clase. V
196. Una de las características básicas de unos lenguajes orientados a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes. V
197. La instrucción throw en Java sólo permite lanzar objetos que son instancias de la clase java.lang.Throwable o de clases derivadas de ésta. V
198. En Java, la instrucción throw no se puede usar dentro de un bloque catch. F
199. Los métodos genéricos no se pueden sobrecargar ni sobrescribir. F

200. Una clase abstracta siempre tiene como clase base una clase interfaz. F
201. De una clase abstracta no se pueden crear instancias, excepto si se declara explícitamente algún constructor. F
202. C++ sólo permite heredar cuando la clase hija es un subtipo de la clase padre (herencia como implementación de la generalización). F
203. El constructor de copia permite argumentos tanto por referencia como por valor. F
204. El estado de un objeto es el conjunto de valores de los atributos y métodos que han sido invocados sobre él. F
205. En las jerarquías de herencia en C++, si la clase base define un operador de asignación y la clase derivada no lo redefine, al invocar a dicho operador con objetos de la clase derivada se invocará al código de la clase base. F
206. La herencia es más flexible en cuanto a posibles cambios en la naturaleza de los objetos que la composición F
207. La herencia privada en C++ es un tipo de herencia insegura porque no preserva el principio de encapsulación. F
208. La relación de herencia es una relación de clases no persistente. F
209. Un objeto se caracteriza por poseer un estado, un comportamiento y una identidad. V
210. Una clase abstracta siempre tiene que tener alguna clase que derive de ella. F
211. Tanto la herencia protegida como la privada permiten a una clase derivada acceder a las propiedades privadas de la clase base F-
212. Una clase abstracta se caracteriza por no tener atributos. F
213. La siguiente clase: class S {public: virtual ~S()=0; virtual void f()=0;}; constituye una interfaz en C++. V
214. Desde un método de una clase derivada nunca puede invocarse un método implementado con idéntica signatura de una de sus clases base. F
215. Los métodos con enlace dinámico son abstractos. F
216. Los constructores de las clases abstractas son siempre métodos abstractos. F
217. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase. F
218. Un método sobrecargado es aquel que tiene más de una implementación, diferenciando cada una por el ámbito en el que se declara, o por el número, orden y tipo de argumentos que admite. V
219. Un método abstracto es un método con polimorfismo puro. F
220. Todo espacio de nombres define su propio ámbito, distinto de cualquier otro ámbito. V
221. En la sobrecarga de operadores binarios para objetos de una determinada clase, si se sobrecarga como función miembro, el operando de la izquierda es siempre un objeto de la clase. V
222. La genericidad se considera una característica opcional de los lenguajes orientados a objetos. V
223. Hablamos de encapsulación cuando diferenciamos entre interfaz e implementación. V
224. Una operación de clase no es una función miembro de la clase. F
225. Los constructores siempre deben tener visibilidad pública. F
226. En C++, si no se captura una excepción lanzada por un método, se produce un error de compilación. F
227. En C++, la cláusula throw() tras la declaración de una función indica q ésta no lanza ninguna excepción. V
228. Dada una clase genérica, no puede ser utilizada como clase base en herencia múltiple. F
229. De una clase interfaz no se pueden crear instancias. De una clase abstracta sí. F
230. Cuando usamos la varianza estamos haciendo un uso inseguro de la herencia de implementación. V
231. En la sobrecarga de operadores como función miembro, el operando de la izquierda puede ser un objeto de la clase o cualquier otro tipo de objeto, mientras que en las funciones amigas siempre es un objeto de la clase. F
232. Cuando creamos un objeto en C++ mediante una variable automática el constructor se autoinvoca. También se autoinvoca el destructor del mismo al salir del ámbito de la función donde se creó. V
233. Si para una clase genérica llamada Pila<T> declaramos las siguientes funciones: virtual void apilar(T* pt); virtual void apilar(T t); En caso de sobrescritura ya que el tipo devuelto es el mismo en ambos métodos. F
234. De una clase abstracta se pueden crear referencias a objetos. V
235. En un atributo de clase se reserva espacio en memoria para una copia de él por cada objeto de su clase creado. F
236. Declarar un dato miembro de una clase como private indica que sólo puede acceder a ese atributo desde las funciones miembro de la clase. F
237. Una clase abstracta siempre tiene que tener alguna clase que derive de ella. F
240. El polimorfismo debido a la sobrecarga de funciones siempre se da en relaciones de herencia. F
241. A los atributos de instancia si son constantes se les asigna su valor inicial fuera de la clase. F

242. Toda sentencia que aparece después del punto del programa en el que ocurre una excepción, en ningún caso se ejecuta. F
243. Si utilizamos los mecanismos de manejo de excepciones disminuye la eficiencia del programa incluso si no se llega a lanzar nunca una excepción. V
244. Cuando se captura una excepción y ésta pertenece a una jerarquía de clases, el primer bloque catch debe comenzar con la clase del nivel más alto de la jerarquía. F
245. Hablamos de shadowing cuando el método a invocar se decide en tiempo de compilación. V
246. A diferencia de otros lenguajes de programación en C++ la sobreesctritura en relaciones de herencia se debe indiciar de forma explícita en la clase padre. V
245. La encapsulación es un mecanismo que permite separar de forma estricta interfaz e implementación. V
246. La interpretación de un mismo mensaje puede variar en función del receptor del mismo y/o del tipo de información adicional que lo acompaña. V
247. Un atributo de clase público puede ser accedido desde fuera de la clase a través de un objeto d e la clase, un puntero o referencia al mismo o mediante el nombre de la clase seguido del operador de ámbito. V
248. Es posibles definir un constructor de copia invocando en su cuerpo al operador de asignación. V
249. El recolector de basura es un mecanismo de liberación de recursos presente en todos los lenguajes OO. F
250. Para que se pueda realizar una herencia múltiple en C++, es necesario que no coincida ninguno de los nombre de atributo entre las clases bases involucradas. F
251. La signatura de tipo de un método incluye el tipo devuelto por el método. V
252. En el principio de sustitución implica una coerción entre tipos de una misma jerarquía de clases. V
253. En C++, un destructor no puede ser virtual. F
254. El puntero this no es una variable polimórfica porque es constante y no se puede cambiar su valor. F
255. Las instrucciones para el manejo de excepciones nos permite mezclar el código que describe el funcionamiento normal de un programa con el código encargado del tratamiento de errores. F
1. La herencia de interfaz se implementa mediante herencia pública. V
2. Una interfaz no puede tener atributos de instancia. Una clase abstracta si puede tenerlos. V
3. No se puede definir un bloque catch sin su correspondiente bloque try. V
4. Una variable polimórfica puede hacer referencia a diferentes tipos de objetos en diferentes instantes de tiempo. V
5. El downcasting siempre es seguro. F
6. La sobrecarga basada en ámbito permite definir el mismo método en dos clases diferentes. V
7. En el diseño mediante tarjetas CRC, utilizamos una tarjeta por cada jerarquía de herencia. F
8. Un espacio de nombres es un ámbito con nombre. V
9. Un sistema de tipos de un lenguaje asocia a cada tipo una expresión. V
10.Hacer que el código sea más fácil de entender no es un motivo suficiente para refactorizarlo. F
11.En Java, los tipos genéricos sólo se pueden aplicar a clases e interfaces. F
12.En Java, una clase genérica puede ser parametrizada empleado más de un tipo. V
13.Sean dos clases Base e Hija. La clase Hija hereda de Base. En Java, cuando asignamos un objeto de la clase Hija a una referencia a Base haciendo conversión de tipo explícita estamos haciendo object slicing. F
14.Una de las principales fuentes de problemas cuando utilizamos herencia múltiple es que las clases bases hereden de un ancestro común. V
15. Los lenguajes de programación soportan el reemplazo y el refinamiento como métodos de sobrescritura, pero no hay ningún lenguaje que proporcione ambas técnicas (JAVA solo soporta reemplazo y C++ sólo soporta refinamiento). F
16.Una interfaz puede implementar otra interfaz. F
17.En java es obligatorio indicar que un método de una clase derivada sobrescribe un método de la clase base con la misma signatura. F
18. En la sobrecarga basada en ámbito los métodos pueden diferir únicamente en el tipo devuelto. V
19.En la herencia pública la clase derivada podrá acceder a los atributos privados de la clase base de la que hereda. F
20. Una clase abstracta se caracteriza por no tener ningún constructor. F
21.El cambio de una condicional por el uso de polimorfismo es un ejemplo de refactorización. V
22.Un atributo siempre tiene visibilidad pública. F
23.El principio de segregación de interfaz indica que el código cliente no debe ser forzado a depender de interfaces que no utilice. V
24.El usuario de un framework implementa el comportamiento declarado en los interfaces del framework mediante herencia de implementación F

25. El proceso de diseño de un sistema software se debería intentar aumentar la cohesión y reducir el acoplamiento. V
26.Cuando diseñamos sistemas orientados a objetos las interfaces de las clases que diseñamos deberían estar cerradas a la extensión y abiertas a la modificación. F
27.La existencia de una sólida colección de pruebas unitarias es una precondición fundamental para la refactorización. V
28.Existe un catálogo de refactorizaciones comunes de forma que el programador no se ve obligado a usar su propio criterio y metodología para refactorizar código. V
29.Arraylist es una implementación en el Java Collection Framework de la interfaz List. V
30.Con el uso de reflexión solo podemos invocar métodos de instancia. F
31.La siguiente clase class S{ public Object obj; } constituye una interfaz en Java. F
32.Desde un método de una clase derivada solamente puede invocarse un método implementado con idéntica signatura de una de sus superclases si el método en la clase base tiene enlace dinámico. F
33.Los métodos con enlace dinámico son métodos abstractos. F
34.Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica. F
35.La genericidad se considera una característica opcional de los lenguajes orientados a objetos. V
36.Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes. V
37.En java, podemos definir constructores con distinta visibilidad en la misma clase. V
38.Los métodos genéricos no se puede sobre sobrecargar ni sobrescribir. F
39.Una interfaz no tiene instancia. Por ejemplo, dada la interfaz Comprable en Java, no podemos hacer new Comparable(). V
40.Una interfaz en Java obliga a que las clases no abstractas que la implementan definan todos los métodos que la interfaz declara. V