




1. Un atributo de clase 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 la clase derivada. 
2. En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación de instancia de los objetos pasados como parámetro. 
3. En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación del objeto receptor del mensaje.
4. En el cuerpo de una operación de clase se puede acceder únicamente a atributos/operaciones de clase.
5. En C++ el constructor copia permite argumentos tanto por referencia como por valor.
6. En C++ el constructor copia se invoca de forma implícita cuando devolvemos un valor o pasamos un objeto como argumento por valor.
7. Las declaraciones `A a(b);` y `A a = b;` (siendo `b` un objeto de tipo `A`) invocan ambas al constructor de copia de la clase `A`.
8. En C++ el constructor copia solo pertenece a la forma canónica de la clase si esta procesa memoria dinámica.
9. La forma canónica en JAVA incluye el constructor, constructor copia, equals, hashCode, toString y el método clone.
10. En JAVA y en C++ solo podremos especificar como visibilidad para un atributo/operación son: `private`, `public` y `protected`.
11. El método `clone()` que proporciona JAVA por defecto, realiza una 'Deep copy' que copia bit a bit el contenido del objeto pasado como parámetro en el nuevo objeto.
12. Tanto JAVA como C++ proporcionan un constructor copia que realiza una Shallow copy del objeto pasado como parametro.
13. Tanto JAVA como C++ proporcionan un método clone que devuelve una copia Shallow copy del objeto que invoca el método.
14. El método clone que se nos proporciona por defecto en Java invoca al clone de la clase base, así como, el método constructor copia que C++ proporciona por defecto.
15. C++ siempre proporciona un constructor por defecto (sin parámetros) de oficio, JAVA no.
16. C++ siempre proporciona un constructor por defecto (sin parámetros) mientras que no definamos ningún otro constructor.
17. JAVA siempre proporciona un constructor por defecto (sin parámetros) de oficio, mientras que no definamos ningún otro constructor.
18. El orden de ejecución de los constructores en la herencia es primero la derivada y luego la base.

19. Un atributo privado en la clase base no es directamente accesible en las derivadas sea cual sea el tipo de herencia, tanto en JAVA como en C++.
20. Un atributo public en base es accesible en main desde un objeto derivado sea cual sea el tipo de herencia.
21. En JAVA la herencia solo puede ser pública, mientras que en C++ solo puede ser privada o protegida.
22. La herencia pública implica herencia de implementación y de interfaz mientras que la herencia privada o protegida implica herencia de implementación pero no de interfaz.
23. La herencia de implementación siempre implica herencia de interfaz.
24. La existencia de una relación todo-parte entre dos clases implica necesariamente que el objeto todo maneja la creación/destrucción de los objetos de tipo parte.
25. Las relaciones todo-parte (tiene-un) son la agregación y la composición.
26. Las relaciones todo-parte son relaciones persistentes como la herencia.
27. En la composición el objeto todo maneja la creación/destrucción de los objetos de tipo parte al contrario que la agregación que no la maneja.
28. La única relación no persistente es la relación de uso, ya que no se materializa mediante referencias.
29. Todo son relaciones de clase excepto la relación de uso que es relación de objeto.
30. La agregación es una relación mucho más restrictiva que la herencia.
31. La agregación y la asociación pueden ser bidireccionales.
32. En JAVA un atributo de clase público puede ser accedido desde fuera de la clase a través de una referencia de la clase o mediante el nombre de la clase.
33. En C++ 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.
34. Implementar la forma canónica 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.
35. Un constructor en JAVA, acepta cualquier tipo de modificador (static, final, public, etc).
36. En C++ los métodos virtuales puros, pueden tener implementación pero siempre harán abstracta a la clase que los contenga.
37. En JAVA los métodos abstractos pueden tener implementación pero siempre harán abstracta a la clase que los contenga.



38. La sentencia Derivada `d = d2.clone();` es correcta si `d` y `d2` son de tipo Derivada. 
39. Una clase abstracta siempre tiene que tener alguna clase que derive de ella.
40. En una composición un objeto componente puede formar parte de más de un objeto compuesto.
41. En una composición un objeto compuesto puede poseer varios componentes del mismo tipo al contrario que en la agregación.
42. Ni la asociación ni la agregación manejan la creación/destrucción de objetos de tipo parte.
43. Los destructores se pueden invocar como si fueran métodos y acepta cualquier número de parámetros.
44. Una clase derivada puede añadir nuevos métodos/atributos propios de la clase derivada pero no modificar los métodos heredados de la clase base.
45. El puntero `this` es un puntero que no puede cambiar de valor y que contiene la dirección del objeto receptor del mensaje, además existe en cualquier método definido dentro de la clase.
46. La interpretación de un mismo mensaje puede variar en función del receptor del mensaje y de la información adicional que lo acompañe.
47. La relación de uso es una relación no persistente.
48. En C++ es posible definir un constructor de copia invocando únicamente al operador asignación.
49. El recolector de basura es un mecanismo de liberación de recursos presente en todos los lenguajes OO.
50. Para que se pueda realizar una herencia múltiple en C++, es necesario que no coincida ninguno de los nombres de atributo entre las clases base involucradas.
51. La signatura de tipo incluye el tipo devuelto por el método, el tipo el número y el orden de los parámetros y el nombre del método.
52. El principio de sustitución implica una coerción entre tipos de una misma jerarquía de herencia.
53. El puntero `this` no es una variable polimórfica porque es constante y no se puede cambiar su valor.
54. No se puede derivar de una clase no genérica a una genérica.
55. No se puede derivar una clase genérica a otra no genérica.

56. Las instrucciones para el manejo de excepciones nos permiten mezclar el código que describe el funcionamiento normal de un programa con el código encargado del tratamiento de errores.
57. No pueden haber asociados varios bloques catch a un bloque try.
58. En C++ no se pueden definir sobrecargas de operadores fuera de la clase a no ser que sean funciones amigas.
59. Definir dos métodos que se llamen igual pero que tengan distinto tipo devuelto se llama sobrecarga.
60. Para sobrecargar dos métodos dentro del mismo ámbito es necesario que difieran en el número, tipo y orden de sus argumentos.
61. En la sobrecarga basada en ámbito los métodos deben diferir en el tipo, orden o número de argumentos.
62. En la sobrecarga nunca se podrá cambiar el tipo devuelto por un método.
63. Los métodos de sobrecarga son el refinamiento y el reemplazo.
64. En JAVA toda clase es potencialmente polimórfica en C++ solo si posee algún método virtual.
65. Ser abstracto implica enlace dinámico.
66. Un método con enlace dinámico siempre será abstracto.
67. En JAVA se puede elegir entre enlazar un método de forma estática, no poniendo virtual delante del método.
68. Los miembros de instancia final de una clase se pueden inicializar en el cuerpo de los constructores.
69. JAVA inicializa todos los atributos de una clase, incluyendo los atributos constantes.
70. Si A tiene una relación de uso con B, no contendrá datos de tipo B pero algunos de sus métodos recibirán objetos de tipo B como parámetros, accederán a sus variables privadas o usarán algún método de la clase B.
71. En algunos lenguajes una clase es una instancia de otra clase, llamada metaclass.
72. El diagrama de clases define las clases, sus propiedades y como se relacionan unas con otras, proporcionando una vista estática de los elementos que conforman el software.
73. Los componentes de una tarjeta CRC son el nombre de la clase, sus responsabilidades y los colaboradores con los que esta clase interactúa.
74. Los bloques try catch no se pueden anidar.

75. Si en el bloque try de un catch, que captura una checked exception, es imposible que se produzca dicha excepción el código no compilará, indicando que el bloque catch es inalcanzable.
76. Si en la sentencia throws de un método especificamos una excepción que es imposible que se lance en el cuerpo de dicho método el código no compilará, indicando que el método nunca lanzara la excepción.
77. Cuando un método sobreesscribe a otro, este podrá especificar lista de excepciones en su declaración mediante la sentencia throws.
78. Cuando un método sobreesscribe a otro, si indica una excepción de tipo checked dicha excepción (o una compatible e) deberá aparecer en la lista throws del método sobrescrito.
79. Cuando un método sobreesscribe a otro, el nuevo método podrá cambiar la visibilidad del método heredado sin restricciones.
80. En JAVA solo existe la sobrescritura, la redefinición y la ocultación son características de C++.
81. Cuando una clase sobrescribe el comportamiento de un método heredado estaremos haciendo herencia de interfaz.
82. Una clase que sobrescriba todos los métodos heredados de la clase base, realiza herencia de interfaz.
83. Cuando una clase sobrescribe algunos de los métodos heredados de la clase base, estaremos únicamente ante una herencia de implementación.
84. La herencia privada y protegida de C++ son herencia únicamente de interfaz.
85. Los constructores no se heredan.
86. Los constructores siempre se refinan ya sea de forma explícita o implícita.
87. Para invocar al método constructor de la clase base de forma explícita usaremos la sentencia super.Base(parámetros).
88. No podemos invocar desde un constructor de la clase a otro constructor de la misma clase.
89. Tanto en JAVA como en C++ los constructores se pueden invocar como si fueran métodos, objeto.constructor(parámetros).
90. C++: En la herencia primero se destruyen los objetos bases y luego las derivadas, es decir, primero se ejecutan los destructores de la base y luego los de las derivadas.
91. Un objeto derivado al que se accede a través de una referencia a clase base sólo se puede manipular usando haciendo uso de la interfaz de la clase base.
92. C++ soporta herencia múltiple de implementación y de interfaz, sin embargo, JAVA solo soporta herencia múltiple de interfaz.

93. Dos son los problemas que nos podemos encontrar en la herencia múltiple en C++, la colisión de nombres y la duplicación de propiedades.
94. La colisión de nombres se podrá resolver mediante el operador de ámbito.
95. La duplicación de propiedades se resuelve usando el atributo virtual.
96. La herencia de interfaz garantiza siempre el principio de sustitución.
97. Mientras que la especialización y la especificación son un uso seguro de la herencia, la generalización la restricción y la varianza no lo son.
98. La herencia privada en C++ implementa un tipo de herencia de construcción que si preserva el principio de sustitución
99. La herencia de construcción o herencia de implementación pura, no cumple el principio de sustitución.
100. La herencia ralentiza la velocidad de ejecución.
101. La regla del cambio y la regla del polimorfismo nos servirán para diferenciar si estamos ante una relación IS-A o una relación HAS-A.
102. La composición es una técnica generalmente más sencilla que la herencia y más flexible, resistente en cuanto a cambios.
103. En la sobrecarga en signatura de tipos, en los parámetros, no se considerarán tipos distintos, si el parámetro en el nuevo método es derivado del parámetro del sobrescrito.
104. Al sobrescribir un método en clase derivada, podemos cambiar el tipo de retorno del método a un subtipo del especificado en la clase base.
105. Si usamos sobrecarga basada en signatura de tipos, los métodos tienen enlace estático (static) y los tipos de los parámetros son diferentes pero relacionados por herencia, si invocamos al método usando una referencia de tipo Base, como parámetro, pero que apunta a un derivado, se invocará al método que recibe una Base en su definición.
106. JAVA permite la sobrecarga de operadores, C++ no.
107. En la sobrecarga de operadores, no se puede cambiar la precedencia, asociatividad o aridad de estos.
108. Para sobrecargar un operador como método de clase (función miembro), el operando de la izquierda deberá ser del tipo de la clase.
109. En JAVA el enlazado de métodos es siempre dinámico, aunque un método definido como final, se comportaría como estático.
110. Tanto la redifinición como el shadowing tienen enlazado estático.
111. Podemos utilizar el atributo final para indicar que no podemos heredar de dicha clase.

112. En JAVA toda variable es potencialmente polimórfica, en C++ solo aquellas que poseen métodos virtuales.
113. Las variables this y super son variables polimórficas.
114. El Downcasting es el polimorfismo inverso.
115. C++ y JAVA soportan downcasting estatico y dinámico.
116. En el downcasting dinámico la comprobación de la conversión se hará en tiempo de compilación.
117. El downcasting en JAVA es siempre seguro.
118. Un método polimórfico o con polimorfismo puro es aquel que no recibe variables potencialmente polimórficas.
119. ArrayList<Integer> e; y ArrayList<String> s; devolverán el mismo tipo, si reciben el mensaje getClass().
120. ArrayList<Object> a = new ArrayList<String> (); y Object [] a = new String [3]; son declaraciones correctas.
121. La implementación de un framework sigue el principio de Hollywood: “no nos llames, nosotros te llamaremos”.
122. Cualquier operador puede ser sobrecargado como función miembro.
123. Los miembros de instancia final (constantes) de una clase, se pueden inicializar en el constructor de dicha clase.
124. Los miembros de clase final (constantes) de una clase, se pueden inicializar en el constructor de dicha clase.
125. Los miembros públicos de instancia de una clase son accesibles en cualquier función o método usando el nombre de la clase y el operador de ámbito en C++ y el nombre de la clase y el operador punto en JAVA.
126. Los miembros protected solo son accesibles por los métodos de esa clase.
127. En JAVA y C++ los miembros private de una clase son accesibles únicamente por los métodos de esa clase.
128. Si en una relación todo-parte las partes pueden cambiar de objeto todo se aproximará más a ser una agregación que una composición.
129. De una clase abstracta no se podrán crear objetos pero si punteros o referencias.
130. El modelo de herencia en C++ es merge.
131. El atributo virtual se hereda.

132. En C++ el atributo virtual sobre un método cambia su enlazado a dinámico y además se hereda.
133. Tres son los tipos de polimorfismo en jerarquías de herencia ocultación, redefinición y sobrecarga.
134. En la sobrescritura en el método definido en la derivada debe tener la misma signature que el de la base.
135. En la redefinición (sin virtual en base) puede cambiar únicamente la parte derecha de la signature.
136. En C++ la sobrecarga de operadores es conmutativa.
137. Las operaciones consultoras obtienen información sobre el estado del objeto (en C++ deben ser declaradas const), mientras que las ordenes modifican el estado del objeto (en C++ no podrán ser declaradas const).
138. C++ y JAVA son lenguajes débilmente tipados.
139. `List<Fruit> flist = new ArrayList<Apple>();` Es una asignación correcta siempre que Apple sea derivado de Fruit.
140. `List<?> flist = new ArrayList<Apple>();` Es una asignación correcta.
141. Si accedemos a una colección mediante una referencia comodín podremos consultar y añadir elementos a dicha colección.
142. JUnit es una librería.
143. En JAVA podremos crear objetos de tipo Collection, List, Set, SortedSet, Map y SortedMap.
144. Si usamos el ArrayList y el TreeSet estaremos usando el JCF, como un framework.
145. Implementar la interfaz comparator y llamar al método sort implica usar el JCF como un framework.
146. Para poder usar las librerías de del JDK, en algún caso, nos veremos obligados a sobrescribir algún método abstracto.
147. Los toolkits son frameworks.
148. JDBC es una librería.
149. Los desarrolladores usaremos las librerías de los fabricantes que implementan en framework JDBC.
- 150.



