Tarea Práctica

Saul Ivan Rivas Vega

Ejercicios Teóricos

Ejercicio 1.

La respuesta es (d) Imprime en pantalla 654321.
La razón de esto es que es correcta la creación del arreglo *a* y de la variable *i,* ahora con i tomando el valor del último índice en el arreglo *i*, que corresponde a su tamaño menos uno, se recorren todos los elementos del final al inicio es decir del 6 al 1 siendo esto los valores de los elementos en el arreglo. Otra observación es que se hace uso de *System.out.print* y esto hace que se impriman los números en una sola línea en contraste de haber usado *System.out.println* que habría hecho que se imprimieran cada número en una línea.

Ejercicio 2.

La respuesta es (d) El programa no compila.
La razón de esto es que en la línea 8 se introduce una instrucción de imprimir pero justo entre la estructura de un *try-catch* lo cual el compilador interpreta que no hay un *catch* para ese *try* incluso si este se encuentra justo debajo de la instrucción de impresión. Otra observación es que en la línea 9 en el *catch e* es de tipo *Excepcion* el cual no está definido, el que está definido es *Exception,* lo cual de igual manera causa un error de compilación.

Ejercicio 3.

La respuesta es (c) Imprime Resultado de la operación: 14.
La razón de esto es la prioridad de operadores en la evaluación de la expresión:

(5 + 4) * 3 / 2 + 1

Primero los operadores en el grupo de mayor prioridad en esta expresión son:
() Paréntesis
Posteriormente:
* Multiplicación
/ División
Y finalmente:
+ Suma

Al evaluar el paréntesis tiene como resultado parcial 9, lo que significa que la expresión se encuentra como:
9 * 3 / 2 + 1
Ahora para el segundo grupo serán evaluados de izquierda a derecha:
27 / 2 + 1
En el paso de la división como los valores primitivos no fueron definidos como float entonces se interpretan como enteros y la división entera de 27 / 2 es 13:
13 +1
Finalmente al evaluar la suma el resultado es:
14

Ejercicio 4.

La respuesta son los incisos (a), (b) y (d).
(b) es CORRECTA puesto que una clase F puede implementar interfaces y en este caso B y D lo son, además (a) es CORRECTA también porque puede implementar más de una interfaz.
(d) es CORRECTA puesto que E es una clase F puede extenderla, heredando así las propiedades y métodos correspondientes.
(c) es FALSA pues implica que F hereda de A y E, siendo ambas clases sería herencia múltiple algo que no está permitido en Java.
(e) es FALSA porque F intenta implementar una interfaz y una clase abstracta, siendo esta última la que no estaría permitido implementar pues no es una interfaz, sin embargo lo que podría modificarse para que pudiera tener elementos de B y C sería que implemente B y extendiera C.

Ejercicio 5.

La respuesta es (c) true, false y Error en tiempo de ejecución.
En a[3] == b[2][1] como los indices estan en el rango correcto solo se están comparando los valores de los enteros en esas posiciones, en este caso 98 y 98 siendo iguales y por lo tanto evaluando a *true.*
En b[1] == b[2] se están comparando las referencias de los arreglos en esas posiciones y como son referencias distintas evalúa a false.
En b[0][3] == a[3] el problema yace en que en el primer arreglo de *b* es decir b[0] no existe un elemento en la posición 3, que sería el cuarto elemento y como solo hay 3 elementos al tratar de evaluar la igualdad lanza un error pero en tiempo de ejecución.

Ejercicio 6.

Describa a detalle qué es lo que sucede.

Primero se define una interfaz con un solo método *saluda* llamada *IFunciones*, posteriormente se define una clase concreta llamada FuncionesImpl la cual implementa a la clase *IFunciones* y por lo tanto define el contenido del método *saluda* siendo este una impresión del argumento nombre diciendo que viene de la implementación de la Interfaz.

Posteriormente se define una clase abstracta con solo un método abstracto *saluda* llamada AbstractFunciones, también se define una clase concreta llamada *Funciones* la cual tiene la implementación de *saluda* similar a FuncionesImpl sin embargo cambia en la impresión.

Pasando a la clase *Reactivo5* se definen dos métodos cuya implementación mandan a llamar el método *saluda*. En el main se realiza una instanciación de la clase *Reactivo5* y se manda a llamar el método que recibe un objeto de tipo *IFunciones* y otro de *Funciones* de tal forma que se pueda imprimir un valor llamando a sus respectivos métodos de *saluda*.

Indique cuál es la diferencia entre las líneas 2 y 6 de la clase Reactivo5;

La diferencia clave es el tipo de objeto que se recibe como argumento, puesto que uno tomara el del tipo que sea *IFunciones* y otro de *AbstractFunciones* lo cual indica que puede hacerse un casting implícito para necesariamente a los objetos de tipo *FuncionesImpl*, de igual forma para el otro método pero para objetos de tipo *Funciones*.

hay alguna utilidad práctica en ellas (¿Qué se logra?).

Se logra un casting a la clase o interfaz que definió el “contrato” del comportamiento de los objetos, así se puede mantener esas firmas en los métodos para cualquier clase concreta que implementa la interfaz o extienda a la clase abstracta, manteniendo el control de las instancias que se usan en el método.

Ejercicio 7.

La respuesta son los incisos (a), (b) y (e).

Las restricciones para los identificadores en Java impiden que empiecen con un número o con algún símbolo que no sea “_” o “$”, o de un número, haciendo los incisos (c) y (d) FALSOS.

Los incisos (a), (b) y (e) son CORRECTOS porque cumplen las reglas de los identificadores y no usan algún símbolo que no sea “_” o “$”.

Ejercicio 8.

Una gran diferencia es que *String* es inmutable a diferencia de *StringBuffer* que sí lo es, esto quiere decir que un objeto de tipo String no puede modificar su valor, sin embargo lo que si se puede hacer es que la referencia pase a ser a una nueva cadena siendo alguna transformación de la original. Ejemplo:
Modificar una cadena “Hola” para que tenga como valor “Hola Saul”.

Usando *String*:
String probando = “Hola”;
probando = probando +” Saul”;
Aquí en la expresión probando +” Saul”, se está generando una nueva referencia a la cadena “Hola Saul”, y se la estamos asignando a probando, cambiamos la referencia completa a una nueva cadena, la cadena original no fue modificada solo se creó una nueva a partir de ella.

Usando *StringBuffer*:
StringBuffer probando = new StringBuffer("Hola");
probando.append(“ Saul”);
Aquí directamente se hace uso del método “append” de la clase *StringBuffer* para modificar el valor en probando, eso lo hace mutable, no se crea una nueva.

Por la propiedad de la mutabilidad en *StringBuffer* es que es recomendada en aplicaciones donde se modifiquen constantemente cadenas y sea más eficiente.