Evaluación síncrona 3 de Programación Orientada a Objetos. Fecha: Jueves, 3 de diciembre de 2020.

Nombre del alumno: Armando Emmanuel Correa Amorelli Aciertos: \_\_\_\_\_\_\_\_\_

*Sección I*.- Codifica lo que se te pide en cada pregunta, de acuerdo a la información de este diagrama. Para cada instrucción que no se pueda realizar indica porqué no se puede (hay tres que no). Cuando se pida guardar un dato en una variable, declare la variable intuyendo el tipo de dato. *Valor: 50 puntos*.

<interface>

**Map** <K, V>

get(key) : V

put(key, value)

remove(key)

size()

*Collection <E>*

add(element)

clear()

remove(object)

size() : int

*Map <K, V>*

get(key) : V

put(key, value)

remove(key) : V

keySet(): Set<K>

*List <E>*

get(index) : E

indexOf(object) : int

remove(object)

size() : int

*Queue <E>*

offer(element)

poll(): E

peek(): E

*Set <E>*

add(elem) : boolean

contains(elem): boolean

ArrayList<E>

TreeMap<K, V>

HashMap<K, V>

LinkedList<E>

Vector<E>

HashSet<E>

Stack<E>

PriorityQueue<E>

TreeSet<E>

ArrayDeque<E>

1. Crea una tabla *importantDates* que asocie fechas (*Date*) con su nombre. Agrega a la tabla los datos que se muestran. El tipo de tabla elegido debe mantener ordenados los datos de acuerdo a la clave.

|  |  |
| --- | --- |
| **Value** | **Key** |
| 5-may-1862 | Batalla Puebla |
| 25-dic-2017 | Esta Navidad |
| 16-sep-1810 | Independencia |

public static void main(String[] args) throws Exception {

Date d1 = new Date(1862, 5, 5);

HashMap<String, Date> importantDates = new HashMap<>();

importantDates.put("Batalla Puebla", d1);

importantDates.put("Esta Navidad", new Date(2017, 12, 25));

importantDates.put("Independencia", new Date(1810, 9, 16));

}

1. Elimina la fila con clave “Batalla Puebla” y guarda en *v* el valor de “Esta navidad”.

importantDates.remove("Batalla Puebla");

Date v = importantDates.get("Esta Navidad");

1. Con una línea de código, guarda todas las claves en *keys*.

Set<String> keys = importantDates.keySet();

1. Añade la fecha de hoy a *keys*.

NO se puede, nos arroja una excepcion del tipo UnsupportedOperationException

1. Sea *b* el resultado de preguntar si “Revolución” pertenece a *keys*.

boolean b = keys.contains("Revolucion");

1. Sea *i* el resultado de preguntar la posición de “Esta Navidad” en *keys*.

No se puede, al ser un map, no te puede retornar un indice que signifique su posición.

1. Crea una cola *timeQueue* que almacene horas (*DateTime*). El tipo de cola a utilizar es aquel que entrega el dato más antiguo y utiliza un arreglo subyacente. Haz que la hora actual se forme en la cola.

ArrayDeque <DateTime> timeQueue = new ArrayDeque<>();

timeQueue.add(new DateTime());

1. Crea una lista *queueList* capaz de almacenar colas como la del punto anterior. El tipo de lista a utilizar es aquel que no usa arreglos subyacentes. Añada *timeQueue* a esta lista.

LinkedList<ArrayDeque> queueList = new PriorityQueue<>();

queueList.add(timeQueue);

1. Suponiendo que *queueList* ya está poblada, extrae el elemento más antiguo contenido en la 3er. cola de *queueList*, y guárdalo en *t*.

DateTime t = queueList.get(2).peak();

1. Crea un vector *vec* que permita almacenar números reales de 64 bits.

Vector<Double> vec = new Vector<>();

1. ¿Qué realiza de forma implícita el compilador para poder añadir este valor?

vec.add(156.34);

aumenta el tamaño de vec y crea enlaces entre sus elementos.

*Sección II*.- Crea la clase *Pair* que haga posible lo siguiente: *Valor: 15 puntos.*

Pair<DateTime, String> pair = **new** Pair<DateTime, String>(**new** DateTime(), "Right now");

DateTime t = pair.getFirst();

String s = pair.getSecond();

*Sección III*.- Implementa un método genérico *equals* que devuelva el número de elementos de una colección que sean iguales a alguno de los dos argumentos recibidos. Así se usaría el método *Valor: 15 puntos.*

int eq;

eq = *equals*("Hola", "Adiós", stringSet);

eq = *equals*( 4.6, Math.***PI***, doubleVector);

eq = *equals*( ~~1500~~, new Date(), dateQueue); // No válido

eq = *equals*( 4.6, ~~new Date()~~, doubleVector); // No válido

*Sección IV*.- Reconstruya el método *factorial* para que genere una excepción si el argumento es negativo. Nótese en el ejemplo de abajo que el proceso invocador está *obligado a manejar* la excepción lanzada. También crea la clase *NegativeIntegerException*. *Valor: 8 + 12 puntos.*

public static int factorial(int N) {

int f = 1;

for(int i = 2; i <= N; i ++) f \*= i;

return f;

}

try {

System.*out*.println(*factorial*(5)); // Imprime: 120

System.*out*.println(*factorial*(0)); // Imprime: 1

System.*out*.println(*factorial*(-5));

} catch(NegativeIntegerException ex) {

System.*out*.println(ex); // Imprime: Negative integer found: -5

}

*Sección V*.- ¿Qué imprime el siguiente bloque *try*/*catch*, si el método *binomialCoefficient* (abajo) invoca al método *factorial* reconstruido en la sección anterior? *Valor: 10 puntos.*

try {

System.*out*.println(*binomialCoefficient*(3, 3));

System.*out*.println(*binomialCoefficient*(3, 7));

System.*out*.println(*binomialCoefficient*(3, 2));

} catch(NegativeIntegerException ex) {

System.*out*.println(ex);

}

static int *binomialCoefficient*(int N, int k) throws NegativeIntegerException {

return *factorial*(N) / (*factorial*(k) \* *factorial*(N - k));

}