* 1. ***TP N°6: Colecciones***

|  |
| --- |
| La siguiente guía cubre los contenidos vistos en las clases teóricas  **8. Colecciones** |

***Ejercicio 1***

Indicar cuáles de las siguientes asignaciones son válidas y justificar:

|  |
| --- |
| import java.util.ArrayList;  import java.util.List;  public class Ej1 {    public static void main(String[] args) {  List<Integer> list1 = new ArrayList();  List<Integer> list2 = new ArrayList<Integer>();  List<Number> list3 = new ArrayList<Integer>();  List list4 = new ArrayList<Integer>();  List<? extends Number> list5 = new ArrayList<? extends Number>();  List<?> list6 = new ArrayList();  }    } |

***Ejercicio 2***

El siguiente fragmento de código intenta crear una lista de números enteros y luego eliminar aquellos que son pares. Explicar el error obtenido en tiempo de ejecución. Proponer una solución.

|  |
| --- |
| import java.util.ArrayList;  import java.util.List;  public class Ej2 {  public static void main(String[] args) {  List<Integer> list = new ArrayList<>();  list.add(1);  list.add(6);  list.add(8);  list.add(10);  for (Integer i : list) {  if (i % 2 == 0) {  list.remove(i);  }  }  }  } |

***Ejercicio 3***

Un iterador cíclico es un iterador que, a partir de una colección, permite recorrerla infinitamente. Al igual que un iterador convencional permite recorrer todos los elementos de la colección, pero la diferencia es que, una vez consumido todos los elementos de la colección, vuelve a recorrer los mismos elementos.

Se desea implementar un iterador cíclico que, ante cada invocación a next, retorne un par de elementos de la colección.

El método next lanza una NoSuchElementException cuando una invocación a hasNext hubiera retornado false.

Implementar todo lo necesario para que, con el siguiente programa:

|  |
| --- |
| import java.util.Arrays;  import java.util.HashSet;  import java.util.List;  import java.util.Set;  public class PairCyclicIteratorTester {  public static void main(String[] args) {  List<String> list = Arrays.asList("hola", "que", "tal", "todo", "bien");  PairCyclicIterator<String> listIterator = new PairCyclicIterator<>(list);  for(int i = 0; listIterator.hasNext() && i < 4; i++) {  System.out.println(listIterator.next());  }  System.out.println("----------");  Set<Integer> set = new HashSet<>();  PairCyclicIterator<Integer> setIterator = new PairCyclicIterator<>(set);  System.out.println(setIterator.hasNext());  setIterator.next();  }  } |

se obtenga la siguiente salida

|  |
| --- |
| # hola + que #  # tal + todo #  # bien + hola #  # que + tal #  ----------  false  Exception in thread "main" java.util.NoSuchElementException |

***Ejercicio 4***

Implementar la clase Bank que almacena un conjunto de cuentas (para ello utilizar la familia de clases BankAccount del ***Ejercicio 7*** del ***TP Nº3****)*.

El banco deberá almacenar el conjunto de cuentas que le pertenecen. Se deben ofrecer métodos para operar con las cuentas, como se muestra en el siguiente fragmento de código:

|  |
| --- |
| Bank bank = new Bank();  BankAccount c1 = new CheckingAccount(1234, 5000);  BankAccount c2 = new CheckingAccount(3462, 5000);  bank.addAccount(c1);  bank.addAccount(c2);  System.out.println(bank.accountSize());  System.out.println(bank.totalAmount());  c1.deposit(100);  c2.deposit(200);  bank.removeAccount(c2);  System.out.println(bank.accountSize());  System.out.println(bank.totalAmount()); |

que produce la siguiente salida:

|  |
| --- |
| 2  0.0  1  100.0 |

***Ejercicio 5***

Modificar la implementación del ***Ejercicio 2*** del ***TP Nº4*** para que en lugar de componer dos funciones permita componer N funciones, como se muestra en el siguiente ejemplo:

|  |
| --- |
| public class FunctionTester {  public static void main(String[] args) {  // armamos la funcion (2x)^2 como la composicion de 2x con x^2  CompositeFunction f1 = new CompositeFunction();  f1.addFunction(new LinearFunction(2, 0)); // y = 2x  f1.addFunction(new QuadraticFunction(1, 0, 0)); // y = x^2  System.out.println(f1.evaluate(1)); // 4  System.out.println(f1.evaluate(2)); // 16  // armamos la funcion 3 \* sin(2x) componiendo 2\*x con sin(x) y con 3\*x  CompositeFunction f2 = new CompositeFunction();  f2.addFunction(new LinearFunction(2, 0)); // y = 2\*x  f2.addFunction(new SineFunction()); // y = sin(x)  f2.addFunction(new LinearFunction(3, 0)); // y = 3\*x  System.out.println(f2.evaluate(0)); // 0  System.out.println(f2.evaluate(Math.PI / 4)); // 3  // si no especificamos ninguna funcion, se lanza una excepcion.  CompositeFunction f3 = new CompositeFunction();  System.out.println(f3.evaluate(1)); // IllegalStateException  }  } |

***Ejercicio 6***

Dada la siguiente clase Person:

|  |
| --- |
| public class Person {  private final String firstName, lastName;  private final LocalDate bornDate;  public Person(String firstName, String lastName, LocalDate bornDate) {  this.firstName = firstName;  this.lastName = lastName;  this.bornDate = bornDate;  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (!(o instanceof Person))  return false;  Person person = (Person) o;  if (!firstName.equals(person.firstName))  return false;  return lastName.equals(person.lastName);  }  @Override  public int hashCode() {  return Objects.hash(firstName, lastName);  }  @Override  public String toString() {  return "Person{" +  "firstName='" + firstName + '\'' +  ", lastName='" + lastName + '\'' +  ", bornDate=" + bornDate +  '}';  }  public String getFirstName() {  return firstName;  }  public String getLastName() {  return lastName;  }  } |

implementar la siguiente interfaz PersonCollection:

|  |
| --- |
| import java.util.List;  public interface PersonCollection {    void addPerson(Person aPerson);    List<Person> findByLastName(String lastName);    Person findByName(String firstName, String lastName);    } |

***Ejercicio 7***

Implementar un método que dada una colección de objetos retorne el mayor (recorriéndola elemento a elemento, esto es, sin utilizar Collections.max() o similar). ¿El método debería ser de instancia o de clase? Utilizar tipos genéricos para exigir en tiempo de compilación que los objetos de la colección recibida implementen la interfaz Comparable. ¿El tipo genérico debe estar en la clase, en el método o en ambos?

***Ejercicio 8***

Diseñar la interfaz Bag que define una colección de elementos no ordenados que admite repeticiones. Dado el siguiente programa de prueba, deducir la definición de la interfaz Bag y realizar la implementación BagImpl.

|  |
| --- |
| public class BagTester {  public static void main(String[] args) {  Bag<String> bag = new BagImpl<>();  bag.add("Hola");  bag.add("Hola");  System.out.println(bag.count("Hola")); // 2  bag.add("Chau");  System.out.println(bag.size()); // 3  System.out.println(bag.sizeDistinct()); // 2  System.out.println(bag.contains("Chau")); // true  bag.remove("Hola");  System.out.println(bag.count("Hola")); // 1  bag.remove("Hola");  System.out.println(bag.contains("Hola")); // false  bag.remove("Hola"); // NoSuchElementException  }  } |

***Ejercicio 9***

Se desea implementar un conjunto genérico de elementos, en donde a cada elemento se le agrega un horario (horas y minutos), para luego poder consultar aquellos objetos que pertenezcan a un determinado rango horario. Realizar una implementación de la siguiente interfaz:

|  |
| --- |
| import java.util.Set;  /\*\*  \* Conjunto no ordenado de elementos, en donde a cada elemento se le asigna  \* un horario al momento de la inserción, en formato horas y minutos.  \*  \* @param <T> Tipo de los objetos almacenados.  \*/  public interface TimeSet<T> {  /\*\*  \* Agrega un elemento al conjunto, junto con un horario especificado.  \* Si el elemento a agregar ya existía, se modifica el horario que tenía  \* anteriormente.  \*  \* @throws IllegalArgumentException Si el horario no es valido.  \*/  void add(T elem, int hours, int minutes);  /\*\*  \* Elimina un objeto del conjunto.  \*  \* @param elem Elemento a ser eliminado.  \*/  void remove(T elem);  /\*\*  \* Obtiene la cantidad de elementos del conjunto.  \*/  int size();  /\*\*  \* Determina si un objeto pertenece o no al conjunto.  \*/  boolean contains(T elem);  /\*\*  \* Obtiene todos los objetos que pertenezcan al rango  \* horario especificado.  \*  \* @throws IllegalArgumentException Si el limite superior es menor al  \* inferior, o los horarios no son validos.  \*/  Set<T> retrieve(int hoursFrom, int minutesFrom, int hoursTo, int minutesTo);    } |

Implementar todo lo necesario para que, con el siguiente programa,

|  |
| --- |
| public class TimeSetTester {  public static void main(String[] args) {  TimeSet<String> timeSet = new TimeSetImpl<>();  timeSet.add("Taller POO", 17, 0);  timeSet.add("Taller PI", 13, 0);  timeSet.add("Clase POD", 18, 30);  System.out.println(timeSet.size());  System.out.println(timeSet.contains("Taller POO"));  System.out.println(timeSet.retrieve(12, 0, 17, 0));  timeSet.remove("Clase POD");  System.out.println(timeSet.size());  System.out.println(timeSet.retrieve(17, 30, 18, 30));  }  } |

se obtenga la siguiente salida

|  |
| --- |
| 3  true  [Taller PI, Taller POO]  2  [] |

***Ejercicio 10***

Se cuenta con la interfaz MultiSortedCollection<T>que representa a una colección de elementos **sin repetidos** que mantiene a sus elementos ordenados según varios criterios simultáneos, en base a comparadores.

|  |
| --- |
| import java.util.Comparator;  public interface MultiSortedCollection<T> {  /\*\*  \* Agrega un comparador a la colección.  \*/  void add(Comparator<T> comp);  /\*\*  \* Agrega un elemento a la colección.  \* Arroja IllegalStateException si no se agrego al menos un comparador.  \*/  void add(T elem);  /\*\*  \* Elimina un elemento de la colección. Si el mismo no existe, no hace nada.  \*/  void remove(T elem);  /\*\*  \* Devuelve un objeto iterable en base al comparador pedido.  \* Arroja IllegalArgumentException si el comparador no pertenece a la colección.  \*/  Iterable<T> iterable(Comparator<T> comp);  } |

**Se busca que la implementación sea eficiente en cuanto al tiempo de acceso a los elementos, por encima del uso de memoria**.

Implementar todo lo necesario para que, con el siguiente programa,

|  |
| --- |
| import java.util.Comparator;  public class MultiSortedCollectionTester {  public static void main(String[] args) {  MultiSortedCollection<Person> m = new MultiSortedCollectionImpl<>();  Comparator<Person> nameComparator = (o1, o2) -> o1.getName().compareTo(o2.getName());  m.add(nameComparator);  Comparator<Person> ageComparator = (o1, o2) -> o1.getAge() - o2.getAge();  m.add(ageComparator);  m.add(new Person("Ana", 15));  m.add(new Person("Juan", 20));  m.add(new Person("Pedro", 10));  for(Person person: m.iterable(nameComparator)) {  System.out.println(person);  }  System.out.println();  for(Person person: m.iterable(ageComparator)) {  System.out.println(person);  }  System.out.println();  m.remove(new Person("Ana", 15));  for(Person person: m.iterable(ageComparator)) {  System.out.println(person);  }  }  } |

se obtenga la siguiente salida

|  |
| --- |
| Ana # 15  Juan # 20  Pedro # 10  Pedro # 10  Ana # 15  Juan # 20  Pedro # 10  Juan # 20 |