Nombres: Carlos Riveira y Santiago de Prada
Grupo: 2262

Memoria Practica 5

Apartado 1: Listas ordenadas

- Para poder realizar este ejercicio hemos creado una clase: SortedList<T extends Comparator<T>>, la cúal extiende de ArrayList<T>. En ella, hemos creado: un constructor, el cúal recibe una lista y debe ordenarla respecto al orden natural. En caso de empate usará los distintos comparadores introducidos para decidir el orden.

  Para poder ordenar la lista hemos creado el Método sort(), el cúal usando una lista auxiliar reordena la lista. Más en profundidad, nuestro objetivo es ordenar la lista de menor a mayor, para ello, buscamos en cada iteración del bucle el elemento más pequeño de la lista y lo añadimos a la auxiliar. Para encontrar el elemento menor usamos un for y una variable auxiliar. Mediante el for vamos comparando el elemento menor con cada elemento del array, si uno de ellos es menor sustituimos la variable auxiliar y seguimos. Por cada iteración del while, añadimos el elemento guardado en la variable auxiliar a la lista auxiliar y lo eliminamos de nuestra lista. Finalmente copiamos la lista auxiliar ordenada en nuestro SortedList y finalizamos.

  Para poder manejar los distintos comparadores tenemos un array, el cuál los guarda y son usados a la hora de comparar los objetos de la lista en caso de que con el comparador por defecto sean iguales. Para poder añadir nuevos comparadores hemos creado el método addCriterion(Comparator <? super T> comparador) , el cúal añade un comparador a la lista de comparadores.

Apartado 2: Plantillas simples de generación de texto

- En este apartado se nos pedía definir la clase Template<T extends Comparator<T>>, que servirá para definir plantillas de generación de texto, que será parametrizables con un tipo.

  Para resolver este ejercicio, la clase Template consta de un SortedList para almacenar los objetos, una lista para almacenar los lambdas y un Map<ArrayList<Integer>, String>. Primero hemos creado una función add, que va añadiendo los lambdas a la lista. Aparte, vamos rellenando de información el mapa, el cual relaciona cada string con su Key, un arrayList<Integer>, este ArrayList contiene en su primera posición el índice donde empieza la lista de lambdas para ese determinado string y en su segunda posición el número de lambdas de esa lista. De esta forma podemos localizar los lambdas que rellenarán los huecos del String al que van asociados en el mapa.

  Aparte, tuvimos que realizar el método emit(), el cual permite crear un mapa relacionando cada objeto con sus cadenas de texto. Para ello realizamos un emit para cada objeto que lo que hace es sustituir los huecos del String con la información descrita por las funciones lambda. En está función es donde es

indispensable el mapa que relaciona los strings con la posición y tamaño de su lista de lambdas.

Finalmente, debíamos realizar el método withSortingCriteria, el cúal simplemente recibe un comparador y llama al método addCriterion del primer apartado.

Apartado 3: Plantillas condicionales e iteradoras

- Nos piden completar la clase Template<T extends Comparator<T>>, añadiéndole 2 métodos nuevos. Por una parte, addWhen, con el fin de que las plantillas se emitan en caso de que los objetos cumplan la condición pasada como argumento, que será una lambda de tipo Predicate<T>, ya que retornará un boolean para evaluar si los objeto cumple con la lambda.
  Para poder realizar el ejercicio nos hemos declarado un nuevo atributo en la Template de tipo Map que tiempo de ejecución se comportará como un HashMap. Ese mapa tendrá como clave el texto, el cual tendrá como valor la lambda Predicate <T> que tendrán que cumplir los objetos para ser emitidos. Hemos llamado condiciones al mapa. El cual se va construyendo con llamadas a addWhen, y por lo general para lo demás se comporta de forma muy parecida al método add de esa misma clase.

  Por otra parte, tenemos que añadir el método addforEach, el cual recibe una lambda que retornará una colección de otro tipo genérico nuevo que manejará este método, lo hemos llamado R, además de los lambdas con su plantilla para rellenar los huecos.
  Para todos los elementos de esa colección se deberá emitir la plantilla nueva, en el hueco de la anterior plantilla.

  Finalmente, para probar el test nos hemos tenido que crear la clase Mascot y añadirla como atributo de la clase Person, la cual tendrá un ArrayList de Mascots.
  Esas mascotas son el tipo R de clase addforEach, donde la lambda que emite la colección emite las mascotas de una Persona, para la cual en hueco determinada de la plantilla, se emitirá otra nueva plantilla para cada mascota.

Apartado 4: Añadiendo estrategias

- El objetivo de este apartado es crear funcionalidades que permitan alterar las cadenas de texto, aparte de reemplazar los huecos. En este apartado hemos creado un interfaz Strategy<T>, la cual define un método textModifier(String, string, T t); Después, hemos creado tres clases: TimeStamper<T>, UpperCaser<T> y FilePersister<T>. Todas ellas implementan Strategy<T>. Cada una de ellas hace: añade la fecha al inicio del string, convierte el string a letras mayúsculas y crea e imprime cada string en su propio fichero.

  Para poder utilizar esta funcionalidad, hemos tenido que adaptar el emit() añadiendo un bucle for que fuese recorriendo las estrategias, previamente introducidas a través del método withOptions, y realizando cambios al String antes de introducirlo en el map.

Diagrama de clases