PSP - Anexo ArrayList

Descargar estos apuntes

Índice

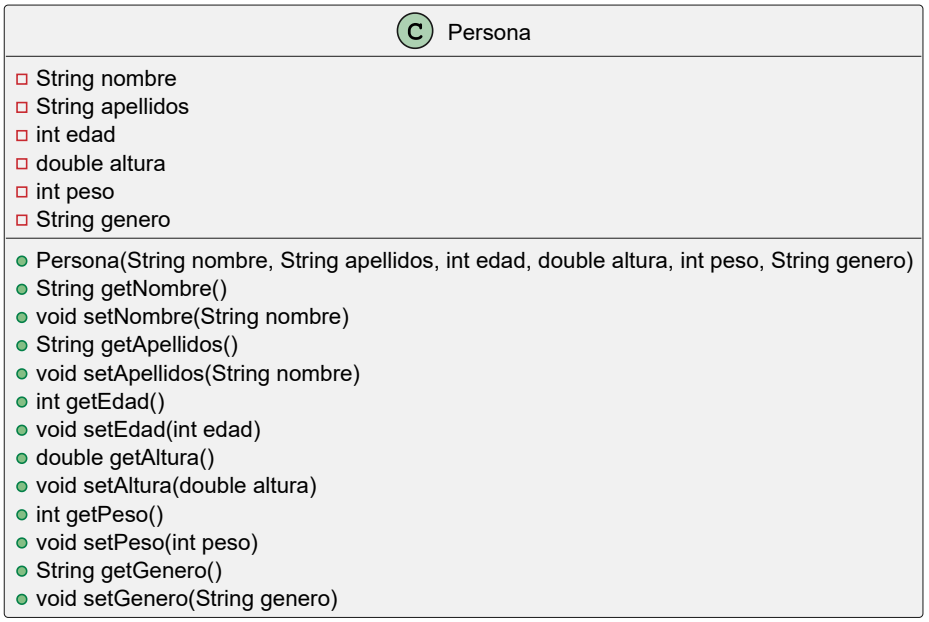
Anexo - ArrayList hoja de referencia de los alumnos

Autoría

Esto es un extracto del trabajo *Reto I (Challenge I)* realizado por mis alumnos como parte del módulo de PSP.
He tomado partes de los diferentes trabajos entregados para complementar la información a la que podréis acceder durante los exámenes.

Gracias a todos.

Para los ejemplos vamos a trabajar con la siguiente clase



A. Definición y creación

Una colección representa un grupo de objetos. Esto objetos son conocidos como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. En Java, se emplea la interfaz genérica `Collection` para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección.

Partiendo de la interfaz genérica Collection extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

La clase ArrayList en Java permite almacenar datos en memoria de forma similar a los Arrays con la ventaja de que el numero de elementos que almacena lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los Arrays. Los elementos pueden añadirse o eliminarse según necesidad.

A.1. Constructores de ArrayList

ArrayList proporciona 3 constructores que definen la capacidad inicial de la colección o si la inicializamos a partir de los valores de otra colección.

```
// Crea una instancia de ArrayList vacía.
ArrayList<Persona> listaPersonas = new ArrayList<>();

// ArrayList(int initialCapacity). Crea una instancia de ArrayList con una capacidad inicial especificada.
ArrayList<Persona> listaPersonas = new ArrayList<>(10);

// ArrayList(Collection c). Crea una instancia de ArrayList a partir de otra colección de datos.
// Los datos se añaden en el orden en el que el Iterator de la otra colección los recorra.
ArrayList<Persona> listaPersonas = new ArrayList<>(c);
```

B. Métodos y propiedades generales

Partiendo de una serie de objetos, vamos a ver el resultado que obtendríamos con la ejecución de estos métodos

```
Persona p1 = new Persona("Manuel", "García", 44, 1.74d, 80, "Hombre");
Persona p2 = new Persona("Juan", "Martínez", 65, 1.84d, 82, "Hombre");
Persona p3 = new Persona("Nombre3", "Apellido3", 52, 1.70d, 66, "Hombre");
Persona p4 = new Persona("Nombre4", "Apellido4", 23, 1.96d, 98, "Mujer");
```

B.1. Creación de un ArrayList

```
// Crear un ArrayList de Personas
ArrayList<Persona> listaPersonas = new ArrayList<>();
```

B.2. Añadir y eliminar elementos

```
// Añadir un objeto Persona al final del ArrayList
listaPersonas.add(p1);

// Añade el elemento al ArrayList en la posición 'n+1'.
// Cuidado, no podemos insertar en posiciones que no existen.
listaPersonas.add(1, p1);

// Borra el elemento de la posición 'n+1' del ArrayList
// Cuidado, no podemos eliminar de posiciones que no existen.
listaPersonas.remove(2);

// Borra el primer objeto pasado encontrado en el ArrayList que se le pasa como parámetro.
listaPersonas.remove(p1);
```

> **Duplicados**
>
> Si ya existe un elemento en el ArrayList, esta colección no controla la existencia de duplicados, por lo que tendremos el mismo objeto en dos posiciones diferentes.
> Podemos evitarlo comprobando previamente si ya existe ese objeto, como veremos a continuación.

B.3. Comprobar si un elemento ya existe

```
// Comprueba si existe del objeto que se le pasa como parámetro
listaPersonas.contains(p4);

// Devuelve la posición del primer objeto pasado encontrado en el ArrayList
// Si no lo encuentra, devuelve -1
listaPersonas.indexOf(p1);

// Devuelve la posición del último objeto pasado en el ArrayList
// Si no lo encuentra, devuelve -1
listaPersonas.lastIndexOf(p1);
```

B.4. Acceder a un elemento del ArrayList

```
// Devuelve el elemento que esta en la posición 'n+1' del ArrayList
listaPersonas.get(2);
```

B.5. Otras funciones de utilidad

```
// Devuelve el numero de elementos del ArrayList
listaPersonas.size();

//Borra todos los elementos de ArrayList
listaPersonas.clear();

// Devuelve true si el ArrayList esta vacío. Sino Devuelve false
listaPersonas.isEmpty();

// Pasa el ArrayList a un Array
Object[] array = listaPersonas.toArray();
```

C. Añadir datos a un ArrayList

> **Orden de los elementos en un ArrayList**
>
> Cuando añadimos elementos a un ArrayList, el orden de inserción se conserva.
>
> Una Lista, por definición, siempre mantiene el orden de los elementos. Esto no es solo para ArrayList, sino para todo tipo de listas como LinkedList, Vector, y el resto de clases que implementan el interfaz `java.util.List` .

C.1. Añadir elementos desde el constructor

A la hora de crear el ArrayList, podemos añadirle datos, usando la sintaxis del doble corchete o bien con la construcción List.of o Arrays.asList

```
// Crea una nueva lista y a la vez la inicializa con valores
ArrayList<Persona> lista1 = new ArrayList<>() {{
    add(p1);
    add(p2);
}};

// En este caso indicamos los valores como si de parámetros se tratase
// De esta forma podemos añadir hasta un máximo de 10 elementos
ArrayList<Persona> lista2 = new ArrayList<>(
      List.of(p1, p2, p3)
);

// De forma similar al caso anterior, con una construcción a partir de un Array
ArrayList<Persona> lista3 = new ArrayList<>(
      Arrays.asList(p1, p2, p3)
);
```

C.2. Añadir elementos desde otras colecciones

Podemos inicializar un ArrayList, como hemos visto en los ejemplos anteriores, desde varios tipos de colecciones (listas) que poseen características similares.

```
// Partiendo del código anterior, creamos un nuevo ArrayList a partir de lista2
ArrayList<Persona> lista4 = new ArrayList<>(lista3);

// Añade todos los elementos de la lista que indiquemos como argumento al final del ArrayList
lista4.addAll(lista3);

// Hace lo mismo, pero lo inserta en la posición indicada (debe existir al menos la posición anterior)
lista4.addAll(2, lista3);
```

C.3. Añadir / eliminar elementos desde código

```
// Añadir un elemento al final de la lista
lista4.add(p4);
// Añade un elemento en la posición indicada. El elemento que ocupaba esa posición y todos los que había detrás,
// se mueven una posición a la derecha.
// En el ejemplo, inserta p1 al principio de la lista.
lista4.add(0,p1);   // Inserta p1 al principio de la lista.

// Eliminar un elemento. Si existe lo elimina y devuelve true, si no devuelve false
boolean existe = lista4.remove(p1);
// Eliminar un elemento por índice. Si la clave existe devuelve el valor asociado, si no devuelve null
Persona eliminada = lista4.remove(2);
// Elimina del ArrayList todos aquellos elementos que coinciden con los indicados en la lista que pasamos como argumento
lista4.removeAll(lista3);
// Elimina del ArrayList todos aquellos elementos que cumplen con el predicado (la condición) descrita como
// argumento en el método e indicada como expresión lambda
lista4.removeIf(p->p.getEdad()<18);   // Elimina los menores de 18
```

D. Recorrer la colección

Vamos a preparar un ArrayList para recorrerlo y usarlo en los siguientes apartados

```
// ArrayList creation
ArrayList<Persona> grupoPersonas = new ArrayList<>() {{
  add(new Persona("Nombre1", "Apellido1", 35, 1.66d, 71, "Mujer"));
  add(new Persona("Nombre2", "Apellido2", 40, 1.84d, 88, "Mujer"));
  add(new Persona("Nombre3", "Apellido3", 52, 1.70d, 66, "Hombre"));
  add(new Persona("Nombre4", "Apellido4", 23, 1.96d, 98, "Mujer"));
  add(new Persona("Nombre5", "Apellido5", 16, 1.55d, 60, "Hombre"));
  add(new Persona("Nombre6", "Apellido6", 20, 1.75d, 74, "Hombre"));
}};
```

D.1. Usando un bucle for

Con el bucle for iteramos de forma natural accediendo a los elementos por índice

```
for (int i = 0; i < grupoPersonas.size(); i++) {
  System.out.println(grupoPersonas.get(i));
}
```

D.2. Usando un bucle foreach de Java

Otra forma de recorrer el ArrayList es con un bucle similar al foreach de C#, aunque con el formato de un bucle for, pero en este caso indicando for(elemento : colección)

```
  // Obtenemos un objeto de tipo Persona en cada iteración del bucle
  for (Persona p: grupoPersonas) {
      System.out.println(p);
  }
```

D.3. Usando Iterator

El interface Iterator de Java permite movernos por una colección y acceder a sus elementos

> **java.lang.Iterator**
>
> Todas las colecciones de Java incluyen un método iterator() que devuelve una instancia de Iterator para recorrer la colección.
>
> Iterator tiene 4 métodos:
>
> - **hasNext()** - devuelve true si hay un elemento más en la lista
> - **next()** - devuelve el siguiente elemento de la lista
> - **remove()** - elimina el último elemento de la lista que hemos obtenido con next()
> - **forEachRemaining()** - realiza la acción indicada con cada uno de los elementos que quedan por recorrer de la lista

Vamos a ver un ejemplo con los valores de la lista

```
  Iterator<Persona> iterator = grupoPersonas.iterator();
  while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

D.4. Usando el método forEach con expresiones lambda

En este caso aprovechamos el método foreach de las colecciones para poder realizar una acción concreta sobre cada uno de los elementos de la misma.

De forma similar al bucle foreach, indicamos un elemento de la lista y la acción a realizar sobre el mismo

```
// Para cada persona (p) -> Acción a realizar
grupoPersonas.forEach(p -> System.out.println(p));
```

D.5 Eliminando / Modificando elementos mientras se itera sobre la colección

Mientras se está recorriendo una colección, no con todos los tipos de bucles se puede modificar (añadir/eliminar elementos) de la colección. Vamos a ver el comportamiento de cada uno de ellos.

D.5.1 Con un bucle for

En este caso no tendríamos problemas. Al acceder por índice, podemos añadir o eliminar elementos mientras se recorre la colección.

```
for (int i = 0; i < grupoPersonas.size(); i++) {
  if(grupoPersonas.get(i).getNombre().equals("Jordi")) {
    grupoPersonas.remove(i);
    // Decrementamos el índice para no saltarnos el siguiente elemento al eliminar el actual
    // También evitaríamos un error de índice fuera de rango si el elemento a eliminar es el último de la lista
    i--;

  }
}
```

D.5.2 Con un bucle foreach de Java

> **No modificable mientras se recorre**
>
> Si intentamos eliminar un elemento mientras lo recorremos con foreach, provocaremos una `java.util.ConcurrentModificationException`.
> Por lo tanto, sólo debemos usar este bucle si queremos leer sus elementos sin modificar la estructura de la colección.

```
  for (Persona p : grupoPersonas) {
    if (p.getPeso()>100) {
      grupoPersonas.remove(p);
    }
    System.out.println("Eliminada: " + p);
}
```

D.5.3 Con Iterator

Siempre que usemos el método remove de Iterator para eliminar elementos de la colección mientras la recorremos, podremos hacerlo sin que se genere ninguna excepción.

```
Iterator<Persona> iterator = grupoPersonas.iterator();
while (iterator.hasNext()) {
  Persona p = iterator.next();
  if (p.getAltura() < 100) {
    // Si borramos usando iterator.remove funciona
    iterator.remove();
  }
}
```

D.5.4 Con el método forEach y expresiones lambda

No modificable mientras se recorre

Si intentamos eliminar un elemento mientras lo recorremos con foreach, provocaremos una `java.util.ConcurrentModificationException`.
Por lo tanto, sólo debemos usar este bucle si queremos leer sus elementos sin modificar la estructura de la colección.

```
// Elimina si encuentra un elemento concreto
grupoPersonas.forEach((p) -> {if (p.getNombre().equals("Jordi")) grupoPersonas.remove(p);});
```

Otra cosa es que intentemos hacer cambios en los valores de la colección, por ejemplo, intercambiar los apellidos con los nombres de cada persona. En este caso, no se modifica la estructura de la colección, por lo que no se genera ninguna excepción.

```
// Intercambia los apellidos <-> nombres de todas las personas
grupoPersonas.forEach((p) -> {
    String aux = p.getApellidos();
    p.setApellidos(p.getNombre());
    p.setNombre(aux);
  }
);
```

E. Búsqueda de elementos

Para buscar elementos en un ArrayList hay distintas formas de hacerlo. Desde los propios métodos que nos ofrece la clase hasta el uso de API Stream. Vamos a describir cada uno de ellos

E.1. Búsqueda usando los métodos de la clase

La clase ArrayList nos ofrece diferentes alternativas para buscar y/o saber si un elemento está presente en la colección. Así, podemos usar los métodos

```
// Comprobar si está el objeto en la colección
boolean existe = grupoPersonas.contains(p1);
// O simplemente obtener su posición, si lo encuentra
grupoPersonas.indexOf(p1);
grupoPersonas.lastIndexOf(p1);
```

E.2. Búsqueda por el valor de una propiedad

A diferencia del caso anterior, si queremos buscar un objeto que contenga un valor concreto en un campo, debemos recorrer la colección hasta encontrarlo. Para eso, una de las alternativas es usar alguno de los bucles vistos anteriormente.

```
Iterator<Persona> it = grupoPersonas.iterator();
while (it.hasNext()) {
  Persona p = it.next();
  if (p.getNombre().equals("Jorge")) {
    // Se puede obtener el elemento o bien modificarlo
    System.out.println("Encontrado: " + p);
    break;
  }
}
```

E.3. Búsqueda usando expresiones lambda

Mediante expresiones lambda, podemos incluir una condicional que nos haga el filtrado de elementos que deseemos

```
// Puede haber más de un elemento que cumpla el criterio de búsqueda
grupoPersonas.forEach(p -> {
  if (p.getNombre().equals("Jorge")) {
    // Se puede obtener el elemento o bien modificarlo
    System.out.println("Encontrado: " + p);
  }
});
```

E.4. Búsqueda usando API Stream

En este tipo de acciones es donde ya podemos empezar a ver la potencia que ofrece el API Stream para el manejo y gestión de las colecciones.
Podemos emplear varios métodos, como filter, findAny, findFirst, allMatch, anyMatch, count, distinct. Como veremos en el siguiente apartado, esos resultados los podemos guardar en forma de subcolección

```
// Puede haber más de un elemento que cumpla el criterio de búsqueda

// Obtener un subarray con los elementos que cumplan el criterio
grupoPersonas.stream()
        .filter(p -> p.getAltura() > 170)
        .collect(Collectors.toList());

// O bien recorrer la lista de entradas obtenidas
for (Persona p : grupoPersonas.stream()
        .filter(p -> p.getAltura() > 170)
        .collect(Collectors.toList())) {
    System.out.println(p);        // Muestra solo la persona
}

// Saber cuántos cumplen el criterio de búsqueda
grupoPersonas.stream().filter(p -> p.getAltura() > 170).count();

// Obtener el primero que cumpla el criterio, si es que hay alguno
Optional<Persona> first = grupoPersonas.stream()
        .filter(p -> p.getAltura() > 170)
        .findFirst();

// Obtener cualquiera que cumpla el criterio, o null si no hay ninguno
Persona any = grupoPersonas.stream()
        .filter(p -> p.getAltura() > 170)
        .findAny()
        .orElse(null);
```

F. Obtención de subcolecciones

Lo podemos considerar un tipo especial de búsqueda en el que el objetivo es conseguir una colección con los elementos que cumplan un determinado criterio.

Así, la forma de buscar es idéntica a la del apartado anterior, pero en este caso lo que obtendremos de esa búsqueda será una nueva lista con un subconjunto de elementos de la original

F.1. Subcolecciones usando bucles

Para obtener los elementos podemos aplicar lo aprendido en el anterior punto. Buscaremos elemento a elemento y, cuando se cumpla una condición especificada, añadiremos los elementos encontrados a una nueva colección.

```
// Personas cuyo nombre empieza por "M"
ArrayList<Persona> personas = new ArrayList<>();
for (Persona p : grupoPersonas) {
  if (p.getNombre().startsWith("M")) {
    personas.add(p);
  }
}
```

F.2. Subcolecciones usando expresiones lambda

La idea es similar al punto anterior

```
// Personas cuyo nombre empieza por "M"
ArrayList<Persona> personas2 = new ArrayList<>();
grupoPersonas.forEach(p -> {
  if (p.getNombre().startsWith("M")) {
    personas2.add(p);
  }
});
```

F.3. Subcolecciones usando API Stream

Podemos obtener directamente una subcolección mediante el filtrado, haciendo subconjuntos y guardando el resultado.

Con API Stream podemos guardar el resultado usando diferentes formas de .collect, que darán como resultados distintos tipos de colecciones.

```java
// ArrayList de Personas cuyo nombre empieza por "M"
ArrayList<Persona> personas3 = (ArrayList<Persona>) grupoPersonas.stream()
        .filter(p -> p.getNombre().startsWith("M"))
        .collect(Collectors.toList());

// Conjunto de Personas cuyo nombre empieza por "M"
Set<Persona> personas4 = grupoPersonas.stream()
        .filter(p -> p.getNombre().startsWith("M"))
        .collect(Collectors.toSet());

// Mapa de Personas cuyo nombre empieza por "M"
// En la construcción del mapa, hay que elegir un campo que sirva de clave
Map<String, Persona> personas5 = grupoPersonas.stream()
        .filter(p -> p.getNombre().startsWith("M"))
        .collect(Collectors.toMap(Persona::getApellidos, Function.identity()));
```

G. Ordenación de elementos

G.1. Ordenar usando métodos de Collection

Si lo que queremos es tener el conjunto ordenados, la forma más fácil es ordenarla usando el método sort() de Collection.

```
// Ordena según el método compareTo sobrescrito al implementar el interfaz Comparable
// Si no se ha implementado, ordena según el orden natural
Collections.sort(grupoPersonas);
// Ordena por un campo cualquiera que indiquemos
Collections.sort(grupoPersonas, Comparator.comparing(Persona::getApellidos));
// Con Comparator tenemos disponibles varios comparadores (naturalOrder, reverseOrder, nullsFirst, ....)
```

Interfaz Comparable

Para que la primera forma de sort funcione, la clase Persona debe implementar el interfaz Comparable y sobrescribir su método `compareTo` para definir la forma de ordenar los objetos de tipo Persona.

Veremos más adelante que tenemos formas de definir el comparador usando expresiones lambda o APi Stream, permitiendo mayor flexibilidad a la hora de comparar elementos.

```
// Un ejemplo, si queremos ordenar a las personas por edad
  @Override
  public int compareTo(Object o) {
      return ((Integer)this.getEdad()).compareTo((Integer)((Persona)o).getEdad());
  }
```

G.2. Ordenar con expresiones lambda

De esta forma no es necesario implementar el interfaz Comparable, ya que podemos indicar la comparación que queremos hacer como parámetro del método sort.

Así podemos tener distintas formas de ordenar, según nos convenga.

```
// Indicamos la comparación que queremos hacer. Podemos usar compareTo
Collections.sort(grupoPersonas, (e1, e2) -> ((Integer)e1.getEdad()).compareTo(e2.getEdad()));
System.out.println(grupoPersonas);

// o bien podemos definir nuestro propio comparador
Collections.sort(grupoPersonas, (p1, p2) -> {
  String nombreCompleto1 = p1.getApellidos() + ", " + p1.getNombre();
  String nombreCompleto2 = p2.getApellidos() + ", " + p2.getNombre();
  return nombreCompleto1.compareTo(nombreCompleto2);
});
```

G.3. Ordenar con API Stream

Con API Stream usamos también el método sorted para indicar qué comparación se debe realizar. Tenemos varias opciones en función del tipo de dato que pasemos.

Como en el caso anterior, el más flexible es aquel en el que indicamos, mediante una expresión lambda, qué comparación realizar.

```
// Guardamos el resultado en una nueva lista con los elementos ordenados
ArrayList<Persona> grupoPersonasOrdenado = (ArrayList<Persona>) grupoPersonas.stream()
        // En este caso ordenamos por apellido, ascendente.
        .sorted((p1, p2) -> p1.getApellidos().compareTo(p2.getApellidos()))
        // Si queremos hacerlo descendente, ponemos .sorted((e2,e1) -> ......
        //.sorted((e2, e1) -> e1.getApellidos().compareTo(e2.getApellidos()))
        .collect(Collectors.toList());
```

Encadenar métodos

Al final, el uso de API Stream nos permite en una misma sentencia, buscar los elementos que queramos, ordenarlos y generar una subcolección con los resultados.

Es lo más parecido que vamos a encontrar a una consulta SQL para los datos de una colección cualquiera.

Aunque su sintaxis no es muy clara, si aprendemos a usarla correctamente, podremos realizar operaciones instantáneas, sin lugar a bugs, con muy poco código.