INTRODUCCIÓN A LA PROGRAMACIÓN

Semana 8

ING. MARTIN POLIOTTO
Docente a cargo del módulo

Septiembre 2020

DIPLOMATURA EN
NUEVAS TECNOLOGÍAS

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA SEU | UTN Facultad Regional Córdoba | Ministerio de Promoción del Empleo y de la Economía Familiar | Ministerio de Ciencia y Tecnología | CBA Entre Todos | Oficina de Montevideo

Semana 08

1. Arreglos dinámicos

1.1 Clase ArrayList

La clase ArrayList en Java, es una clase que permite almacenar datos en memoria de forma similar a los arreglos, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los arreglos

Esta clase representa una **lista** implementada sobre un arreglo (los nodos de las lista están almacenados dentro de un arreglo en lugar de estar "sueltos" en la memoria). Esta forma de implementación supone alguna ganancia de eficiencia en el uso de la memoria si se sabe que los nodos serán relativamente pocos y puede hacerse alguna estimación de esa cantidad.

Los principales métodos para trabajar con los ArrayList son los siguientes:

```
// Declaración de un ArrayList de "String". Puede ser de cualquier otro Elemento
//u Objeto (float, Boolean, Object, etc)

ArrayList<String> nombreArrayList = new ArrayList<String>();

// Añade el elemento al ArrayList

nombreArrayList.add("Elemento");

// Añade el elemento al ArrayList en la posición 'n'

nombreArrayList.add(n, "Elemento 2");

// Devuelve el número de elementos del ArrayList

nombreArrayList.size();

// Devuelve el elemento que está en la posición '2' del ArrayList

nombreArrayList.get(2);

// Comprueba se existe del elemento ('Elemento') que se le pasa como valor

nombreArrayList.contains("Elemento");

// Devuelve la posición de la primera ocurrencia ('Elemento') en el ArrayList

nombreArrayList.indexOf("Elemento");

// Devuelve la posición de la última ocurrencia ('Elemento') en el ArrayList

nombreArrayList.lastIndexOf("Elemento");

// Borra el elemento de la posición '5' del ArrayList

nombreArrayList.remove(5);

// Borra la primera ocurrencia del 'Elemento' que se le pasa como parametro.

nombreArrayList.remove("Elemento");

//Borra todos los elementos de ArrayList

nombreArrayList.clear();

// Devuelve True si el ArrayList esta vacio. Sino Devuelve False

nombreArrayList.isEmpty();

// Copiar un ArrayList
```

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

2. Calificadores **static** y **final**

2.1 Elementos de clase: **static**

Cuando un atributo o método de una clase se marca con la palabra reservada ***static*** este miembro pasa a ser de clase y no de instancia (por ejemplo public static void main() utilizado en la clase Ejecutable). Esto lo convierte en un miembro compartido por todas las instancias de la clase.

Los miembros estáticos de una clase se cargan en memoria antes que se cree cualquier instancia de la clase, y todas las instancias que se creen luego, acceden exactamente al mismo elemento: no hay una copia de un atributo estático en cada instancia, sino que todas las instancias "ven" la misma variable (cada instancia posee un puntero a la única copia que hay de ese miembro).

El siguiente ejemplo muestra la definición de una clase Test que permite contar la cantidad de objetos creados:

```
public class Test {
  //atributo de clase:
  private static int contador = 0;
  //atributo de instancia:
  private String valor;
  public Test(String valor) {
    this.valor = valor;
```

```
        contador++; //Cada vez que se crea un objeto Test, se increment el contador
    }
    public String getValor() {
        return valor;
    }
    public static int getContador() {
        return contador;
    }
}
```

Clase Ejecutable:

```
public class Ejecutable {
    public static void main(String[] args) {
        Test a, b, c;
        a = new Test("Ok");
        b = new Test("Nok");
        c = new Test("Ok");
        int x = Test.getContador();
        System.out.println("Cantidad de instancias creadas:" + x);
    }
}
```

- Si ejecutamos el proyecto el programa mostrará un mensaje indicando que 3 instancias fueron creadas.
- Es importante tener presente que todos los miembros no estáticos tienen acceso a los miembros estáticos. Al inverso

no es posible. Es decir, no se puede acceder a un atributo de instancia desde un contexto estático

- Un uso frecuente de métodos estáticos son los definidos en las clases de utilidad o soporte, donde no necesitamos crear un objeto de la clase para poder utilizar sus servicios. Recuerde el uso de los métodos de la clase Math.

2.2 Declarar constantes: **final**

Un atributo puede ser marcado también como final, lo cual en esencia lo convierte en una constante. Una vez asignado un valor inicial a una constante, el mismo no puede ser modificado durante el resto del programa (cualquier intento de hacerlo provocará un error de compilación).

Por ejemplo:

```
public class Test{
    public static final String POR_DEFECTO = "Nok";
}
```

Además, los calificadores **final** y **static** pueden combinarse: un atributo marcado con ambos calificadores será una constante (por ser final), y también será compartida la misma copia de esa constante por todas las instancias de la clase (por ser static). En casos como estos, en que el atributo es compartido y constante, se estila también declararlo publica en lugar de private: al fin y al cabo, el Principio de Ocultamiento busca evitar que se manipule de manera incorrecta el valor de un atributo de la clase, pero eso no podrá ocurrir si ese atributo es único para todas las instancias y marcado como constante.

2.2.1 Consideraciones sobre **final**

Tener presente que si:

- Un **atributo** se marca como *final* este podrá asumir un valor y luego no se podrá modificar durante la ejecución del programa. Por convención para definir un calificador de una constante no se utiliza notación de camello, si no que se indica mediante mayúsculas. Si el nombre de la variable consta de más de una palabra, entonces se utiliza el carácter de subrayado (_)
- Un **método** si indica como *final* este no podrá ser redefinido en las clases hijas.
- Una **clase** se marca como *final* entonces no se podrá crear otra clase que herede de esta última.

3. Apéndice: Componentes **JComboBox** y **JTable**

3.1 JComboBox

Clase que permite mostrar una lista de elementos como un combo de selección, ideal para gran cantidad de opciones de selección única.

3.1.1 Como rellenar un JComboBox

Una de las propiedades más importantes de este tipo de componentes es la lista de ítems o elementos a seleccionar. Para poder cargar esta lista de valores existen varias opciones. A continuación se muestran las frecuentes:

- Utilizando un modelo: **DefaultComboBoxModel**:
  Podemos asociar al componente un objeto que se encargue de gestionar los datos (**modelo**) que se mostrarán en la lista.

  Ejemplo:

  ```
  //Luego de inicializar los componentes:
  //Creamos un arreglo con los valores:
  String valores[] ={"Seleccione", "Opción 1", "Opción 2"};
  jcMiCombo.setModel(new DefaultComboBoxModel<>(valores));
  ```

  Otra alternativa sería crear una clase propia que se comporte como un modelo, extendido de la clase *AbstractListModel*, y luego instanciar y asociar con el combo.

- Utilizando el método **addItem**()
  Esta segunda forma es la más simple y fácil de entender, ya que tan solo debemos utilizar el método additem() que nos

provee el componente, tal como se muestra a continuación:

```
jcMiCombo.addItem("Seleccione");
jcMiCombo.addItem("Opcion 1");
jcMiCombo.addItem("Opcion 2");
jcMiCombo.setSelectedIndex(0);
```

Adicionalmente el uso del método **addItem**() nos da mayor libertad para agregar elementos de forma dinámica, por ejemplo en un ciclo cuando obtenemos datos de una base de datos.

Notar que por defecto la lista de elementos es de tipo String, pero en realidad puede contener cualquier tipo de objetos.

3.1.2 Item seleccionado

Asociado con el ítem que se ha seleccionado existen dos métodos de utilidad:

- **getSelectedIndex()**: que retorna el índice del elemento en la lista. Siempre comenzando en 0 (cero) para la primer posición. Si no existe ningún ítem seleccionado, este método devolverá -1.
- **getSelectedItem**(): retorna el elemento que se encuentra actualmente seleccionado. El elemento siempre es de tipo *Object*, por lo que es necesario hacer una conversión de tipo antes de usarlo.

3.1.3 Evento **ActionPerformed**

Como la mayoría de los componentes el JComboBox tiene un evento de acción asociado que permite reconocer cuando selecciona un ítem de la lista.

Por ejemplo:

```
private void jcMicomboActionPerformed(java.awt.event.ActionEvent evt){
    JOptionPane.showMessageDialog(this, "Item seleccionado: " +
jcMicombo.getSelectedItem());
}//Muestra en una ventana modal el valor del elemento seleccionado.
```

3.2 JTable

Un **JTable** es un componente visual de java que nos permite dibujar una tabla, de forma que en cada fila/columna de la tabla podamos poner el dato que queramos; un nombre, un apellido, una edad, un número, etc (muy útil para mostrar datos de una base datos).

Como muchos componentes de java, se ha seguido una separación modelo-vista . La vista es el componente visual que vemos en pantalla, el modelo es una clase que contiene los datos que luego se verán en pantalla. El modelo de datos únicamente contiene los datos, no sabe nada de quién va a visualizar los datos ni cómo.

Para diseñar la tabla el Netbeans permite modelar gráficamente mediante la opción: *Contenido de la tabla* haciendo click derecho sobre el componente en la vista de diseño. Para cada columna es

posible definir si será editable, si se podrá redimensionar o no e indicar el tipo de componente a visualizar.

3.2.1 Como rellenar un JTable

Siguiendo con el mismo criterio que con el JComboBox, podemos crear un modelo de datos utilizando un modelo por defecto, tal como se muestra a continuación:

```
DefaultTableModelo modelo = new DefaultTableModel();
JTable tabla = newJTable(modelo);
```

A partir de ahora todo se maneja con el modelo. En cuanto se añadan, borren o cambien datos del modelo, el JTable se enterará y actualizará automáticamente. El DefaultTableModel tiene todos los métodos necesarios para modificar datos en su interior, añadir filas o columnas y darle a cada columna el nombre que se quiere visualizar como encabezado.

El siguiente ejemplo permite cargar una tabla con datos de infracciones modeladas mediante objetos Multa:

```
//datos a cargar:

Multa[] filas = new Multa[3];

multas[0] = new Multa(1, 1, 300);

multas[1] = new Multa(2, 9, 1000);

multas[2] = new Multa(3, 9, 1000);

//Crear un modelo por defecto:

DefaultTableModel modelo = new DefaultTableModel();

//Establecer los encabezados de la tabla:

modelo.setColumnIdentifiers(new String[]{"Acta", "Código", "Monto"});
```

```
//Cargar las filas de la tabla con los objetos Multa:
 for (Multa multa : multas) {
        modelo.addRow(new  Object[]{multa.getActa(),  multa.getCodigo(),
 multa.calcularMonto()});
 }
//Finalmente asignar el modelo al JTable.
jTable1.setModel(modelo);
```