INSTITUTO SAN ISIDRO

# Curso de Java a distancia

## Clase 24: Clases genéricas

Java permite crear clases que administren distintos tipos de datos.

Se utilizan mucho para la administración de colecciones de datos (pilas, colas, listas, árboles etc.)

Las clases genéticas nos evitan duplicar clases que administran tipos de datos distintos pero implementan algoritmos similares, son una herramienta fundamental para reutilizar código.

Vimos en conceptos anteriores como implementar estructuras dinámicas de tipo pila y dependiendo del problema definíamos una pila de enteros o de String o de cualquier otro tipo de datos.

Si no existiera el concepto de genéricos en Java deberíamos implementar una clase por cada tipo de dato que administra una pila.

**Problema**

Implementar una clase Pila que administre cualquier tipo de datos mediante el concepto de genéricos.

Crear luego 4 objetos de la clase Pila y almacenar en la primer pila objetos de la clase Persona, en la segunda objetos de la clase Carta, en la tercera objetos de la clase String y finalmente en la cuarta objetos de la clase Integer.

**Programa:**

```
public class Persona {
   private String nombre;
   private int edad;

   public Persona(String nombre, int edad) {
      this.nombre = nombre;
      this.edad = edad;
   }

   public void imprimir() {
      System.out.println(nombre + " " + edad);
   }

}
public class Carta {
   private int numero;
   private String palo;

   Carta(int numero, String palo) {
      this.numero = numero;
```

```
        this.palo = palo;
    }

    public void imprimir() {
        System.out.println(numero + " " + palo);
    }

}
public class Pila<E> {
    class Nodo {
        public E info;
        public Nodo sig;
    }

    private Nodo raiz;

    public void insertar(E x) {
        Nodo nuevo;
        nuevo = new Nodo();
        nuevo.info = x;
        if (raiz == null) {
            nuevo.sig = null;
            raiz = nuevo;
        } else {
            nuevo.sig = raiz;
            raiz = nuevo;
        }
    }

    public E extraer() {
        if (raiz != null) {
            E informacion = raiz.info;
            raiz = raiz.sig;
            return informacion;
        } else {
            return null;
        }
    }

    public int cantidad() {
        int cant = 0;
        Nodo reco = raiz;
        while (reco != null) {
            reco = reco.sig;
            cant++;
        }
        return cant;
    }

}

public class PruebaGenericos {
    public static void main(String[] args) {
        Pila<Persona> pila1 = new Pila<Persona>();
        pila1.insertar(new Persona("Juan", 33));
        pila1.insertar(new Persona("Ana", 45));
        pila1.insertar(new Persona("Carlos", 33));
        System.out.println("Extraemos el primer elemento de la pila de personas:");
        Persona ultimaPersona = pila1.extraer();
        ultimaPersona.imprimir();
```

```
        Pila<Carta> pila2 = new Pila<Carta>();
        pila2.insertar(new Carta(1, "Corazón"));
        pila2.insertar(new Carta(2, "Corazón"));
        pila2.insertar(new Carta(1, "Trebol"));
        pila2.insertar(new Carta(2, "Trebol"));
        System.out.println("Extraemos el primer elemento de la pila de cartas:");
        Carta ultimaCarta = pila2.extraer();
        ultimaCarta.imprimir();

        Pila<String> pila3 = new Pila<String>();
        pila3.insertar("cadena 1");
        pila3.insertar("cadena 2");
        pila3.insertar("cadena 3");
        pila3.insertar("cadena 4");
        System.out.println("Extraemos el primer elemento de la pila de String:");
        String ultimoString = pila3.extraer();
        System.out.println(ultimoString);

        Pila<Integer> pila4 = new Pila<Integer>();
        pila4.insertar(1);
        pila4.insertar(2);
        pila4.insertar(3);
        pila4.insertar(4);
        System.out.println("Extraemos el primer elemento de la pila de Integer:");
        Integer entero = pila4.extraer();
        System.out.println(entero);

    }
}
}
```

Las clases Persona y Carta no tienen nada nuevo con respecto a temas vistos anteriormente.

Para plantear una clase genética debemos indicar luego del nombre de la clase entre los símbolos menor y mayor un nombre, por convención se utiliza el caracter 'E' (Element):

```
public class Pila<E> {
```

El nombre 'E' almacena el tipo de dato que administrará la pila y se lo inicializa cuando se crea un objeto de la clase Pila:

```
        Pila<Persona> pila1 = new Pila<Persona>();
        ...
        Pila<Carta> pila2 = new Pila<Carta>();
        ...
        Pila<String> pila3 = new Pila<String>();
        ...
        Pila<Integer> pila4 = new Pila<Integer>();
        ...
```

Como vemos cuando creamos el objeto pila1 le antecedemos entre los símbolos menor y mayor el tipo de dato que insertaremos en la pila1, en nuestro caso indicamos la clase Persona.

De forma similar creamos los objetos pila2, pila3 y pila4 indicando en cada uno de los casos otros nombres de clases.

Un punto importante a tener en cuenta que en Java no podemos crear un objeto de una clase genérica de un tipo de dato primitivo, por ejemplo se genera un error de compilación si intentamos hacer:

```
        Pila<int> pila5 = new Pila<int>();
```

El algoritmo de la clase Pila no solo cambia en su declaración, sino en cada lugar que hacemos referencia al tipo genérico.

La información del nodo será del tipo genérico 'E':

```
class Nodo {
   public E info;
   public Nodo sig;
}
```

El método insertar recibe un parámetro x de tipo 'E', el algoritmo propiamente no varía:

```
public void insertar(E x) {
   Nodo nuevo;
   nuevo = new Nodo();
   nuevo.info = x;
   if (raiz == null) {
      nuevo.sig = null;
      raiz = nuevo;
   } else {
      nuevo.sig = raiz;
      raiz = nuevo;
   }
}
```

El método extraer retorna un tipo de dato genérico:

```
public E extraer() {
   if (raiz != null) {
      E informacion = raiz.info;
      raiz = raiz.sig;
      return informacion;
   } else {
      return null;
   }
}
```

# Colecciones: Java API

Hemos visto en conceptos anteriores como administrar distintas estructuras de datos estáticas (vectores, matrices) y dinámicas (listas y árboles)

Aprendimos a crear clases en Java para administrar listas tipo pila, cola y genéricas. Desarrollamos todos los algoritmos internos para su administración utilizando punteros.

Veremos ahora que el API de Java nos provee un conjunto de clases e interfaces que nos facilitan la creación de pilas, colas, listas genéricas etc.

En muchas situaciones el empleo de esta librería de clases e interfaces nos reducen el tiempo de desarrollo de un programa.

Para trabajar con estas clases e interfaces debemos importarlas del paquete 'java.util' donde se encuentran las mismas.

Todas estas clases e interfaces están implementadas con el concepto de genéricos para poder almacenar cualquier tipo de datos.

Las colecciones fundamentales que podemos hacer uso en nuestros proyectos son:

- Stack : Implementa el concepto de una pila (LIFO - Last In First Out - Ultimo en entrar primero en salir)
- Queue : Implementa el concepto de una cola (FIFO - First In First Out - Primero en entrar primero en salir)
- PriorityQueue : Implementa el concepto de una cola por prioridad (por ejemplo si son números los organiza en la cola de menor a mayor)
- ArrayList : Implementa el concepto de un arreglo dinámico que puede crecer o decrecer.
- LinkedList : Implementa el concepto de una lista genérica.
- HashSet, TreeSet y LinkedHashSet : Implementa el concepto de listas sin valores repetidos.

# Colecciones: Stack

Veremos con una serie de ejemplo los métodos disponibles en la clase Stack.

También se las llama listas LIFO (Last In First Out - último en entrar primero en salir)

**Problema**

Definir un objeto de la clase Stack y llamar a los métodos principales.

**Programa:**

```
import java.util.Stack;

public class PruebaStack {

    public static void main(String[] args) {
        Stack<String> pila1 = new Stack<String>();
        System.out.println("Insertamos tres elementos en la pila: juan, ana y luis");
        pila1.push("juan");
        pila1.push("ana");
        pila1.push("luis");
        System.out.println("Cantidad de elementos en la pila:" + pila1.size());
        System.out.println("Extraemos un elemento de la pila:" + pila1.pop());
        System.out.println("Cantidad de elementos en la pila:" + pila1.size());
        System.out.println("Consultamos el primer elemento de la pila sin extraerlo:" +
pila1.peek());
        System.out.println("Cantidad de elementos en la pila:" + pila1.size());
        System.out.println("Extraemos uno a un cada elemento de la pila mientras no este
vacía:");
        while (!pila1.isEmpty())
            System.out.print(pila1.pop() + "-");
        System.out.println();

        Stack<Integer> pila2 = new Stack<Integer>();
        pila2.push(70);
        pila2.push(120);
        pila2.push(6);
        System.out.println("Imprimimos la pila de enteros");
        for (Integer elemento : pila2)
            System.out.print(elemento + "-");
        System.out.println();
        System.out.println("Borramos toda la pila");
        pila2.clear();
```

```
        System.out.println("Cantidad de elementos en la pila de enteros:" + pila2.size());
    }
}
```

Para poder hacer uso de la clase Stack debemos importarla del paquete 'java.util' ya que no se encuentra en el paquete 'java.lang' que se importa en forma automática:

```
import java.util.Stack;
```

Creamos un objeto de la clase Stack indicando que se almacenarán objetos de la clase String:

```
        Stack<String> pila1 = new Stack<String>();
```

Para agregar elementos en la pila lo hacemos a través del método push:

```
        pila1.push("juan");
        pila1.push("ana");
        pila1.push("luis");
```

Para conocer la cantidad de elementos almacenados en la pila llamamos al método size:

```
        System.out.println("Cantidad de elementos en la pila:" + pila1.size());
```

Para extraer un elementos de la pila disponemos del método pop:

```
        System.out.println("Extraemos un elemento de la pila:" + pila1.pop());
```

En cambio si queremos conocer el valor que hay en el primer elemento de la pila pero sin eliminarlo hacemos uso del método peek:

```
    System.out.println("Consultamos el primer elemento de la pila sin extraerlo:" + pila1.peek());
```

El método isEmpty retorna true si la lista está vacía o false en caso contrario:

```
        while (!pila1.isEmpty())
            System.out.print(pila1.pop() + "-");
```

Podemos utilizar un for para recorrer todos los elementos de la colección de tipo Stack con la siguiente sintaxis (no se eliminan los elementos de la pila):

```
        System.out.println("Imprimimos la pila de enteros");
        for (Integer elemento : pila2)
            System.out.print(elemento + "-");
```

Finalmente podemos eliminar todos los elementos de la pila mediante el método clear:

```
        pila2.clear();
```

Cuando vimos como implementar desde cero una pila en Java desarrollamos un problema de aplicación. Lo volveremos a codificar pero ahora utilizando la clase Stack en lugar de la clase Pila implementada desde cero.

**Problema**

Todo compilador o intérprete de un lenguaje tiene un módulo dedicado a analizar si una expresión está correctamente codificada, es decir que los paréntesis estén abiertos y cerrados en un orden lógico y bien balanceados.

Se debe desarrollar una clase que tenga las siguientes responsabilidades (clase Formula):

- Ingresar una fórmula que contenga paréntesis, corchetes y llaves.
- Validar que los ( ) [] y {} estén correctamente balanceados.

Para la solución de este problema la clase formula empleará la clase Stack.

Veamos como nos puede ayudar el empleo de una pila para solucionar este problema.
Primero cargaremos la fórmula en un JTextField.

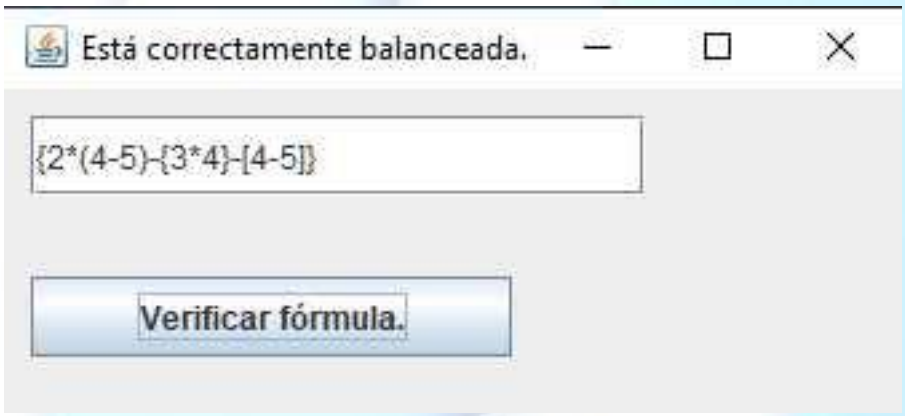
Ejemplo de fórmula: (2+[3-12]*{8/3})

El algoritmo de validación es el siguiente:

Analizamos caracter a caracter la presencia de los paréntesis, corchetes y llaves.
Si vienen símbolos de apertura los almacenamos en la pila.
Si vienen símbolos de cerrado extraemos de la pila y verificamos si está el mismo símbolo pero de apertura: en caso negativo podemos inferir que la fórmula no está correctamente balanceada.
Si al finalizar el análisis del último caracter de la fórmula la pila está vacía podemos concluir que está correctamente balanceada.

Ejemplos de fórmulas no balanceadas:

}(2+[3-12]*{8/3})
Incorrecta: llega una } de cerrado y la pila está vacía.
{[2+4}]
Incorrecta: llega una llave } y en el tope de la pila hay un corchete [.
{[2+4]
Incorrecta: al finalizar el análisis del último caracter en la pila queda pendiente una llave {.




**Programa:**

```
import javax.swing.*;
import java.awt.event.*;
import java.util.Stack;

public class Formula extends JFrame implements ActionListener {
    private JTextField tf1;
    private JButton boton1;

    public Formula() {
        setLayout(null);
        tf1 = new JTextField("{2*(4-5)-{3*4}-[4-5]}");
        tf1.setBounds(10, 10, 230, 30);
        add(tf1);
        boton1 = new JButton("Verificar fórmula.");
        boton1.setBounds(10, 70, 180, 30);
        add(boton1);
        boton1.addActionListener(this);
    }
```

```
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == boton1) {
            if (balanceada()) {
                setTitle("Está correctamente balanceada.");
            } else {
                setTitle("No está correctamente balanceada.");
            }
        }
    }

    public boolean balanceada() {
        Stack<Character> pila1 = new Stack<Character>();
        String cadena = tf1.getText();
        for (int f = 0; f < cadena.length(); f++)
            if (cadena.charAt(f) == '(' || cadena.charAt(f) == '[' || cadena.charAt(f) == '{')
                pila1.push(cadena.charAt(f));
            else if (cadena.charAt(f) == ')' || cadena.charAt(f) == ']' || cadena.charAt(f) == '}') {
                if (pila1.isEmpty())
                    return false;
                else if (cadena.charAt(f) == ')' && pila1.pop() != '(')
                    return false;
                else if (cadena.charAt(f) == ']' && pila1.pop() != '[')
                    return false;
                else if (cadena.charAt(f) == '}' && pila1.pop() != '{')
                    return false;
            }
        if (pila1.isEmpty())
            return true;
        else
            return false;
    }

    public static void main(String[] ar) {
        Formula formula1 = new Formula();
        formula1.setBounds(0, 0, 400, 160);
        formula1.setVisible(true);
        formula1.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

}
```

Lo primero que hacemos es importar la clase Stack del paquete 'java.util':

```
import java.util.Stack;
```

En el método balanceada creamos un objeto de la clase 'Stack' e indicamos que almacenamos objetos de la clase Character (recordemos que con clases genéricas no podemos pasar tipos de datos primitivos como char):

```
        Stack<Character> pila1 = new Stack<Character>();
```

Dentro de un for analizamos caracter a caracter la formula ingresada por el operador:

```
        for (int f = 0; f < cadena.length(); f++)
```

Si se trata de un caracter de paréntesis, corchetes o llaves de apertura procedemos a insertarlo en la pila:

```
            if (cadena.charAt(f) == '(' || cadena.charAt(f) == '[' || cadena.charAt(f) == '{')
                pila1.push(cadena.charAt(f));
```

Si se trata de un paréntesis, corchetes o llaves de cerrado verificamos primero que la pila no esté vacía. Si la pila está vacía podemos decir que la formula no tiene los símbolos correctamente balanceados:

```
if (pila1.isEmpty())
   return false;
```

Si la pila no está vacía debemos controlar que el elemento de la pila coincida con el mismo elemento de cerrado, si no coincide no esta balanceada la formula:

```
else if (cadena.charAt(f) == ')' && pila1.pop() != '(')
   return false;
else if (cadena.charAt(f) == ']' && pila1.pop() != '[')
   return false;
else if (cadena.charAt(f) == '}' && pila1.pop() != '{')
   return false;
```

Cuando se sale del for si la pila no está vacía también significa que no está correctamente balanceados los paréntesis, corchetes y llaves:

```
if (pila1.isEmpty())
   return true;
else
   return false;
```

Con esto terminamos la codificación del mismo problema resuelto en conceptos anteriores, pero utilizando la clase Stack en lugar de implementar nosotros mismos la clase Pila.

Como programadores debemos conocer tanto las librería que nos proporciona el lenguaje para la administración de estructuras dinámicas como los algoritmos que las implementan a esas estructuras que nos pueden ser muy útiles cuando las clases del API de Java no se adaptan a nuestras necesidades.

Muchas gracias hasta la próxima clase.

Alsina 16 [B1642FNB] San Isidro | Pcia. De Buenos Aires |Argentina |

TEL.: [011] 4742-1532 o [011] 4742-1665 |

www.institutosanisidro.com.ar info@institutosanisidro.com.ar