Estructura de datos dinámicas lineales en Java

Colección
Dossier Académico
ULEAM

Tecnologías de la Información y la comunicación (TIC)

Robert Wilfrido Moreira Centeno
Edwin René Guamán Quinche
Oscar Armando González López
Mike Paolo Machuca Ávalos

Ediciones Uleam

Este libro ha sido evaluado bajo el sistema de pares académicos y mediante la modalidad de doble ciego.

Universidad Laica Eloy Alfaro de Manabí
Ciudadela universitaria vía circunvalación (Manta)
www.uleam.edu.ec

Estructuras de datos dinámicas lineales en java

Consejo Editorial: Universidad Laica Eloy Alfaro de Manabí
Director Editorial: Fidel Chiriboga
Diseño de cubierta: José Márquez
Estilo, corrección y edición: Alexis Cuzme (DEPU)

ISBN: 978-9942-775-27-6

Edición: Primera. Julio 2018

Departamento de Edición y Publicación Universitaria (DEPU)
Ediciones Uleam
2 623 026 Ext. 255
www.depu.uleam.blogspot.com
Manta - Manabí - Ecuador

A mi familia, a mi madre María Centeno y mi padre Wilfrido Moreira a quienes por sus esfuerzos respondo con gratitud en cada obra y acción profesional y personal que realizo, a mis hermanos Gabriel Marcelo, María Jimena y a mi pequeño sobrino César Abdiel.

Robert

A mi Dios que es mi fortaleza y a mi amiga Karina.

René

A mi familia, a mi esposa y a mis hijos quienes han sido parte fundamental para escribir este libro, ellos son quienes me dieron grandes enseñanzas y los principales protagonistas de este "sueño alcanzado".

Óscar

A mi esposa, quien ha estado a mi lado y ha creído siempre en mí, a pesar de todo. A mis hijos, quienes son mi inspiración para seguir adelante.

Mike

RESUMEN

La programación es una técnica muy interesante y dentro de las existentes la que cuenta con una vertiginosa evolución, aunque su desarrollo se enmarcó inicialmente en un *begin* y *end*, ahora el alcance de los trabajos que se realizan en ella se expande debido a las características modulares en los lenguajes y herramientas actuales.

En cualquier contexto particular de desarrollo, las variables juegan un rol primordial en la etapa de desarrollo e implementación de un programa, pues dan lugar a conceptos como las constantes, arreglos estáticos, arreglos dinámicos, estructuras de datos dinámicas que son la base de algoritmos en sistemas expertos, todo lo anterior se constituye en fundamentos para entender conceptos vigentes como el *machine learning*.

Los autores de esta obra han vivido la experiencia educativa y por ende conocen algunas de las dificultades que surgen cuando se entra en el campo de la tecnología, este libro constituye un buen inicio para tan interesante disciplina como lo es la programación, puesto que se abarca desde el uso básico de las variables hasta las operaciones comunes numéricas y ahonda en las estructuras de datos.

Luego de hacer referencia a las estructuras básicas de almacenamiento se explica como se origina el concepto de nodo y de él se derivan las famosas estructuras de datos dinámicas, se muestran 2 formas de implementación que dan el mismo, pero que el desarrollo de cada una de ellas muestra lo robusto que se puede ser en el sentido de poco código al momento de programar.

Palabras clave: estructura de datos, Java, programación.

CONVENCIONES TIPOGRÁFICAS

Se ha adoptado un conjunto de convenciones tipográficas con el objetivo de mejorar la lectura de los conceptos y comprender los ejercicios planteados.

Los estilos empleados en el texto son:

- El texto general del libro se utiliza la letra Times New Roman.
- Las palabras reservadas se destacan con color morado. Ejemplo: new, for, if, etc.
- Los nombres de los programas, clases, métodos se destacan con tipo de letra Courier (11 puntos). Ejemplo: `Estudiante`, `mostraMensaje`, `IPersona`, etc
- Toda palabra que no sea en español se destaca con cursiva. Ejemplo: *String Development Kit*, *swing*

Los estilos empleados en los ejemplos son:

- Se han omitido las tildes y eñes en los identificadores de los ejemplos propuestos al ser Java un lenguaje escrito en inglés. Ejemplo: anio, dias, numero, etc.
- Los ejemplos normalmente están completos y por lo tanto se pueden escribir y probar. Se los ha encerrado en cajas y están numerados. El texto de los ejemplos emplea un tipo de letra de paso fijo Courier. Ejemplo:

Ejemplo 1 Estilo del código Java

```
1  public class Computadora {
2        String marca;
3        double precio;
4
5        public void prender( ){
6              // bloque de código del método
7              String mensaje = "Se prendió";
8              System.out.println(mensaje);
9        }
10 }
```

- Los números de línea permiten hacer referencia a una instrucción concreta del código.
- Los ejemplos parciales tienen el mismo formato que un ejemplo completo, pero no se colocará su título.

```
1  int numero = 9;
2  String mensaje = "Se prendió";
```

- La salida que genera un código de ejemplo o un programa se encierra en un cuadro y están numerados (el texto de las salidas emplea un tipo de letra de paso fijo Courier). Ejemplo:

```
1  Ingrese el primer número
2  7
3  Ingrese el segundo número
4  8
5  Ingrese el tercer número
6  6
7  El número mayor es = 8
```

Los estilos empleados en los cuadros de sintaxis y recomendaciones:

- Las sintaxis de cada elemento van encerradas en un cuadro sin numeración. Ejemplo:

tipo_dato [ ] *identificador_arreglo*;
o
tipo_dato *identificador_arreglo* [ ];

- Las recomendaciones o sugerencias se mostrarán a través de un cuadro sin numeración. Ejemplo:

Recomendación
Se recomienda que al definir una clase, se establezca el inicio y final de la clase a través de la llave abierta y cerrada { }, los programadores que están aprendiendo este lenguaje de programación, suelen olvidar cerrar la clase, esto ayudará a evitar errores al momento de compilar sus aplicaciones.

Contenido

INTRODUCCIÓN

El desarrollo de sistemas informáticos es una disciplina que inicia desde las nociones básicas de las matemáticas, en donde el estudio de tendencias numéricas para ejercicios algebraicos o estadísticos se realiza a través del uso de variables, denominadas frecuentemente *x* e *y*.

El concepto de variable aterriza en su propia forma en los diversos lenguajes de programación, se constituyen en el elemento subyacente fundamental de la programación, puesto que se usan para almacenar dominios sencillos y conocidos como lo son texto, números, fechas e inclusive elementos más complejos como conexiones a bases de datos y objetos *JSON* y *XML* que viajan dentro de los protocolos de red de una máquina para su normal proceso de trabajo.

Un poco antes del uso de las variables como objetos contenedores de registros, tablas y bases de datos, se desarrolló el concepto de "estructura de datos" las mismas que tenían su representación más sencilla en variables que almacenaban únicamente un valor, luego en los arreglos, y finalmente una forma en la que una variable almacenaba a una variable del mismo tipo en sí misma y posibilitaba la implementación de métodos para operar sobre ella como insertar, borrar y modificar, dando lugar al concepto de "estructura de datos dinámica", que de acuerdo a cada operación particular define a los tipos de datos abstractos colas, pilas, listas, árboles y grafos.

Este libro sirve de referencia para esas operaciones que caracterizan en particular a cada tipo de dato abstracto, se comparan las mismas operaciones en su forma más larga usando el tipo de dato abstracto nodo, y en su forma más simplificada usando las clases preconstruidas de Java.

1 ESTRUCTURAS DE DATOS DINÁMICAS EN EL PARADIGMA DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

En este capítulo se aborda el concepto de almacenamiento básico de un programa, como el de la variable, la misma abordada desde la perspectiva de la Programación Orientada a Objetos (POO) se llama «clase, envoltorio o *Wrapper*» y en un sentido más complejo pasará a llamarse Estructura de Datos Dinámica cuya definición de métodos particulares la diferencia una de otra, es decir pilas, colas o listas.

Objetivos:

- Conocer la definición de un tipo de dato primitivo y envoltorios en Java.
- Identificar como se definen estructura de datos dinámicas a partir de la auto referenciación.

1.1 VARIABLES Y REFERENCIAS

- Java es un lenguaje de programación de cuarta generación debido a su característica de desarrollo orientada a objetos, el elemento central de todo el código fuente se denomina clase y de este surgen denominaciones como referencias e instancias. Java aún conserva unas características de los lenguajes de tercera generación, denominada tipos de datos primitivos los cuales son 8: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`.

Figura 1 Formas de nombrar a las estructuras de almacenamiento en los lenguajes de programación de tercera y cuarta generación.

Una variable o tipo de dato primitivo es simplemente un nombre con el que se accede a un valor que está guardado en alguna posición de la memoria principal del computador (memoria RAM), en tanto que una clase es una definición general de un conjunto de referencias e instancias con la cual no solamente se accederá a un valor, sino también a un conjunto de propiedades y métodos, que en conjunto se denominan como "comportamiento" en el contexto del paradigma de la Programación Orientada a Objetos.

La declaración de una variable implicará a nivel de aplicación un uso limitado de la misma, debido a que servirá exclusivamente para las operaciones lógicas y matemáticas que le correspondan en el contexto de un programa. Usar una instancia o referencia otorga la posibilidad de usar esta en función de niveles más avanzados a través de un comportamiento que ya está definido previamente en el argot de clases de Java o fue definido por un desarrollador.

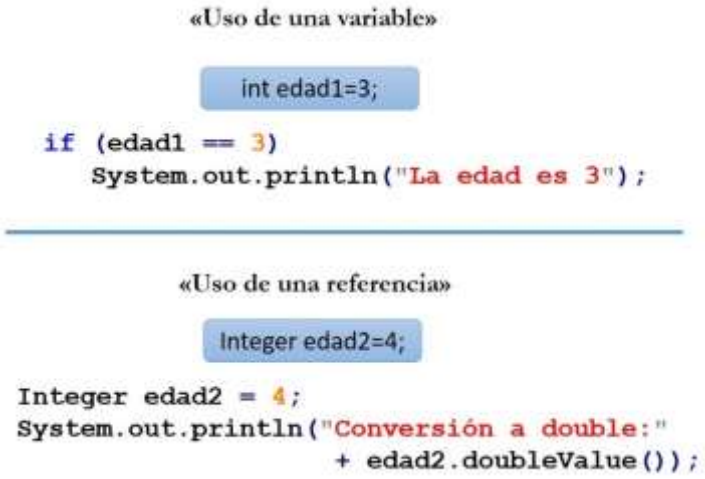

Figura 2 Ejemplo del uso de una variable y una referencia

Se puede observar el uso del método `doubleValue`() para la referencia `edad2`, un método que es parte del comportamiento de todas las instancias de tipo *Integer*; en tanto que para el caso de `edad1` este método es imposible de usar, dado que `edad1` solo está creado como tipo de variable nombre para acceder al valor 3 almacenado en algún lugar de la memoria principal.

Se indica que para la explicación anterior se usó una clase envoltorio, también denominada *Wrapper*, son clases que se crearon en Java para cada dato primitivo, así

por ejemplo el tipo de dato primitivo `double` tiene al *wrapper Double*, el tipo de dato primitivo short tiene al *Wrapper Short*, etc.

1.2 IMPORTANCIA DE LA PROGRAMACIÓN ORIENTADA A OBJETOS EN LA DEFINICIÓN DE TIPOS DE DATOS ABSTRACTOS

Antes de definir un tipo de dato abstracto, es relevante destacar el papel de la Programación Orientada a Objetos al respecto, siendo que la misma consta de 3 pilares fundamentales como lo son la «herencia», el «polimorfismo» y el «encapsulamiento», algunos autores definen un primer pilar sobre todos ellos, el cual es la «abstracción», pero este último es un proceso inherente a todos los otros 3 elementos que anteceden.

1.2.1 LA ABSTRACCIÓN

Todo concepto que se quiera recordar y usar debe ser abstraído, la abstracción es un proceso inherente a cada ser viviente con capacidad de tener recuerdos, la lógica humana evalúa, mide y da importancia a una situación o conocimiento, después de lo cual abstrae una situación determinada, este es un proceso presente desde que nacemos, un bebé evalúa el dolor y lo representa o abstrae como llanto, mide el hambre y lo representa o abstrae como llanto, sin embargo, cuando el niño crezca un poco más al sentir hambre muy posiblemente ya no la abstraerá o representará como llanto sino como muchos gritos, la abstracción entonces depende de la perspectiva y grado de madurez mental.

1.2.2 EL ENCAPSULAMIENTO

Encapsular tal como sugiere la semántica de la palabra es aislar porciones de una realidad, con tal de aplicar el principio de «divide y triunfarás», a nivel de programación implica que el profesional de la información evalué un «comportamiento» y plasma ese comportamiento en un sistema, en Java la palabra reservada que permite el encapsulamiento es `class`**.**

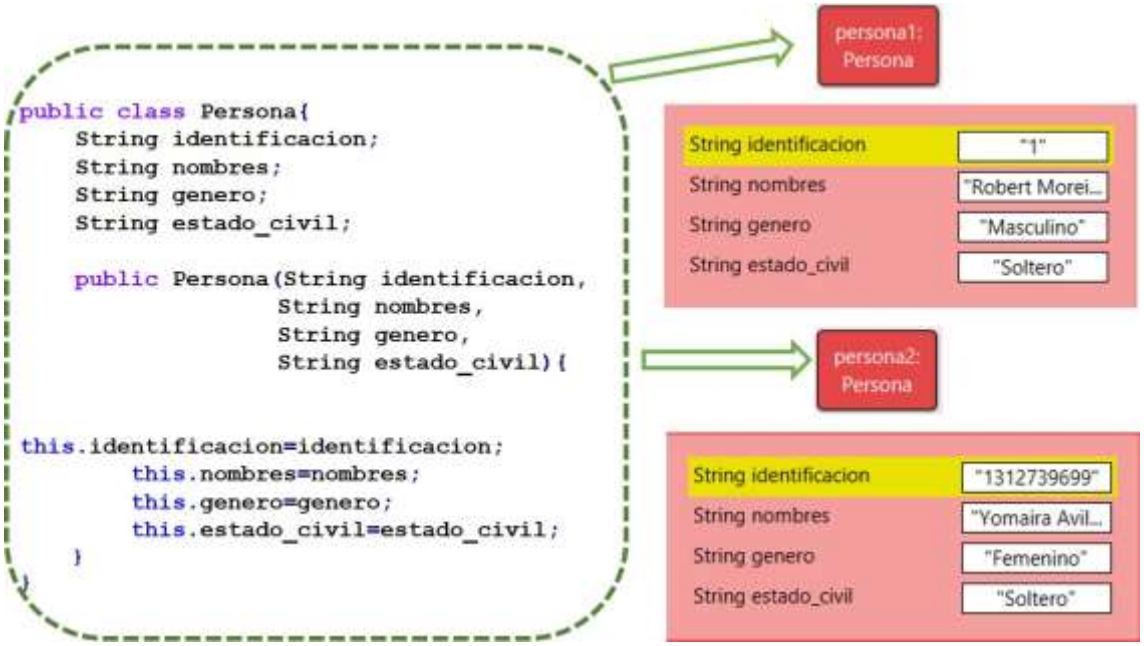

Figura 3 Ejemplo de encapsulamiento a través de la definición de la clase Persona.

El gráfico anterior tiene la definición de lo que se denomina comúnmente «clase base», la misma recibe el nombre `Persona`, de esta clase base se ha creado `persona1` y `persona2`, las cuales reciben el nombre de «referencia» o «instancia».

Ahora fijarse en lo que integra a la «clase base», en este caso:

- 4 propiedades: identificación, nombres, género, estado_civil.
- Un método constructor: que recibe el mismo nombre que está después de la palabra reservada `class`, en este caso `public Persona` (*String* `identificacion`, *String* `nombres`, *String* `genero`, *String* `estado_civil`).
- El conjunto de propiedades y métodos están encapsulados en un `class Persona`, lo anterior representa de forma clara el principio de encapsulamiento.

1.2.3 LA HERENCIA

- Es la forma de compartir propiedades y métodos entre clases, es una característica de la Programación Orientada a Objetos que consiste en heredar «comportamiento» de una «clase base» a una o varias «clases hijas», la herencia

en Java es posible a través de las palabras reservadas `extends`, `interface`, `private`, `public` y `protected`.

La herencia en UML se representa:

Figura 4 Digrama UML herencia entre clases.

La codificación de un diagrama UML a código Java de la clase Padre (Persona) es:

Ejemplo 2 Clase Persona.

```
1   public class Persona{
2       public String identificacion;
3       public String nombres;
4       protected String genero;
5       private String estado_civil;
6
7       public Persona(String identificacion, String nombres,
8                      String genero,String estado_civil){
9           this.identificacion=identificacion;
10          this.nombres=nombres;
11          this.genero=genero;
12          this.estado_civil=estado_civil;
13      }
14  }
```

La codificación de un diagrama UML a código Java de la clase Hija (Niño) es:

Ejemplo 3 Herencia entre la clase Persona y Niño.

```
1   public class Niño extends Persona{
2       String pasatiempoEscuela;
3
4       public Niño(String identificacion, String nombres,
5                   String genero,String estado_civil,
6                   String pasatiempoEscuela){
7           super(identificacion,nombres,genero);
8           this.pasatiempoEscuela=pasatiempoEscuela;
9       }
10  }
```

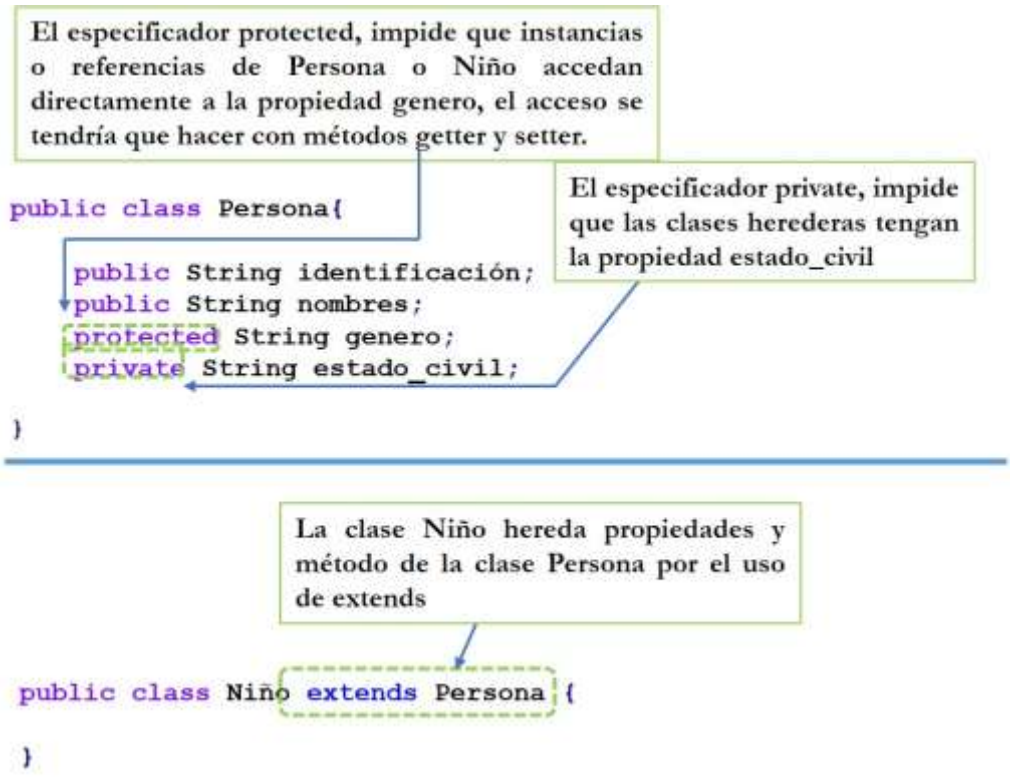

Figura 5 Ejemplo de la Herencia a través de la definición de la clase Persona y Niño.

En el ejemplo anterior la definición de la clase `Niño` usa la palabra reservada `extends` para heredar un conjunto de 3 propiedades, no se cuenta a `estado_civil` en tanto está especificada como `private`, se observa además, en el constructor de la clase hija una invocación usando la palabra reservada `super` en la que efectivamente solo se están usando 3 propiedades de la clase `Padre`, el uso de `protected` e `interface` corresponde a conceptos más profundos de la programación orientada a objetos que no competen al tema tratado en este texto.

1.2.4 EL POLIMORFISMO

El polimorfismo en su semántica para la programación orientada a objetos implica múltiples formas para un mismo método, existen dos formas de implementación del polimorfismo, que son la sobrecarga «overloading» y la sobreescritura «Overriding».

Analicemos el siguiente código:

Ejemplo 4 Sobre carga de constructores.

```
1   public class Persona{
```

```
2       public String identificacion;
3       public String nombres;
4       protected String genero;
5       private String estado_civil;
6
7       public Persona(String identificacion, String nombres,
8                      String genero,String estado_civil){
9           this.identificacion=identificacion;
10          this.nombres=nombres;
11          this.genero=genero;
12          this.estado_civil=estado_civil;
13      }
14
15       public Persona(String identificacion, String nombres,
16                      String genero){
17          this.identificacion=identificacion;
18          this.nombres=nombres;
19          this.genero=genero;
20      }
21
22      public void formaAprender(){
23        System.out.println("Una persona aprende leyendo");
24      }
25  }
```

Ejemplo 5 Sobre escritura de métodos.

```
1   public class Niño extends Persona{
2       String pasatiempoEscuela;
3
4       public Niño(String identificacion, String nombres,
5                   String genero,String estado_civil,
6                   String pasatiempoEscuela){
7           super(identificacion,nombres,genero);
8           this.pasatiempoEscuela=pasatiempoEscuela;
9       }
10  }
11
12  public void formaAprender(){
13        System.out.println("Una niño aprende jugando");
14  }
```

Se llama firma de un método al número y tipos de parámetros que estos reciben. En el código anterior existe «Sobrecarga» para el método constructor `Persona` en tanto que se especifica que reciba 4 parámetros en su primera definición y 3 parámetros en su segunda definición.

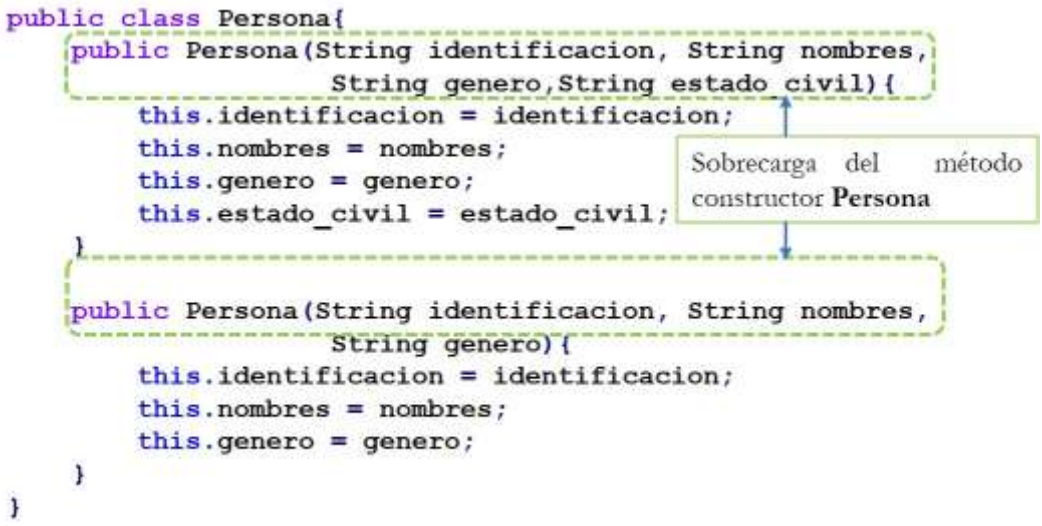

Figura 6 Sobrecarga de un método.

Con respecto al método `formaAprender` el concepto que aplica es «Sobreescritura» en tanto que un mismo método cuenta con distintas implementaciones en la clase base (Persona) y en la clase hija (Niño) tal como se observa en los mensajes.

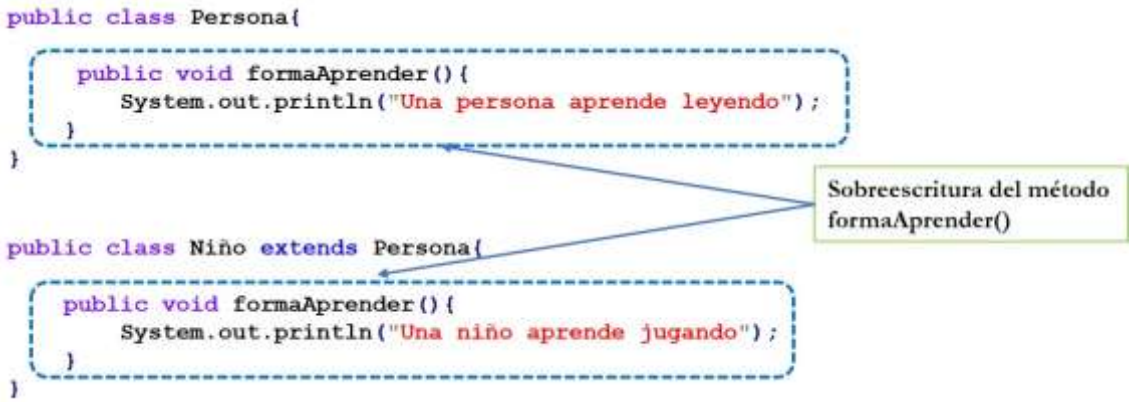

Figura 7 Sobreescritura de métodos.

1.3 TIPOS DE DATOS ABSTRACTOS

Hasta este punto fue necesario entrar en los detalles de variables, y la diferencia de estas con respecto a las referencias o instancias, también conocer que la abstracción es el análisis que un profesional de la información debe realizar para representar un ente de un sistema usando los principios de encapsulamiento, herencia y polimorfismo, a la definición de cada uno de estos entes que se encuentren en un negocio o una necesidad de automatización es lo que se conoce como tipos de datos abstractos.

1.3.1 TIPO DE DATO ABSTRACTO «NODO» Y ORIGEN DE LAS ESTRUCTURAS LINEALES DE DATOS

Una estructura de datos lineal es aquella en la que los datos se caracterizan porque sus elementos están en secuencia, relacionados en forma lineal, uno luego del otro. Cada elemento de la estructura puede estar conformado por uno o varios sub-elementos o campos que pueden pertenecer a cualquier tipo de dato, pero que normalmente son tipos básicos.

Para que sea posible que los elementos estén en secuencia es necesario que una referencia pueda guardar en sí misma, a una referencia de su mismo tipo, nace entonces el concepto de una clase Nodo, cuyos elementos básicos son: dato a almacenar, auto referencia, métodos que lo integran.

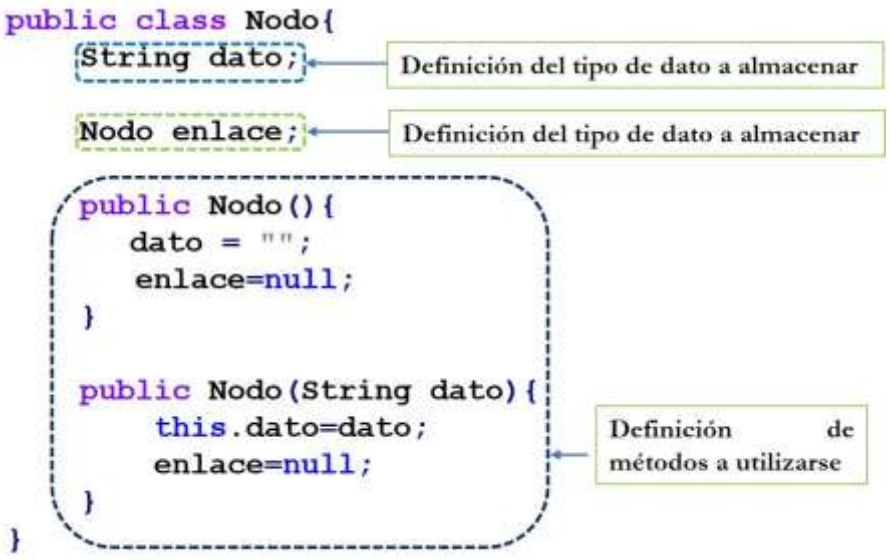

Figura 8 Elementos del tipo de dato abstracto Nodo.

1.4 LA AUTOREFERENCIACIÓN

El concepto de referencia o instancia se obtiene mediante el operador `new`**,** y es el resultado de aplicar la encapsulación, herencia y el polimorfismo, obtenida a través de la definición de clases particulares dentro de un proyecto mediante la palabra reservada `class`.

```
public class Nodo{
	String dato;
	Nodo enlace;

	public Nodo(){
		datos = "";
		enlace = null;
	}

	public Nodo(String dato){
		datos = dato;
		enlace = null;
	}

}
```

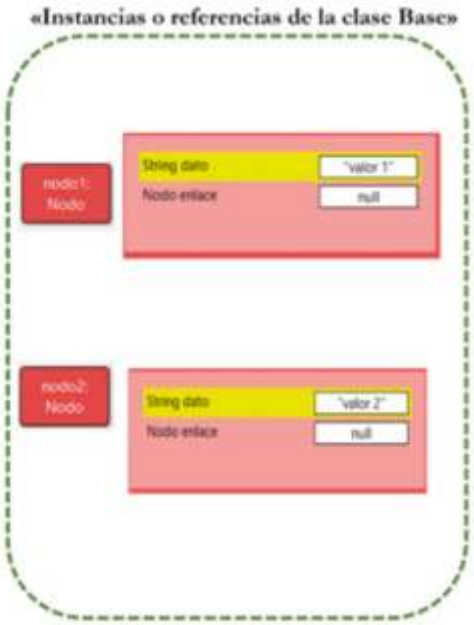

Figura 9 Clases e instancias.

En el gráfico anterior existen dos referencias o instancias, las cuales son `nodo1` y `nodo2`, y fueron definidas a partir de la clase base `Nodo`**.** También es de notar que cuando una referencia se crea y no se instancia el valor de esta es `null`, tal como consta en la creación de las instancias en la derecha en la auto referencia enlace.

Cuando un elemento puede contener dentro de sus atributos un elemento del mismo tipo de la clase base, recibe la denominación de «*Nodo*», pues su propósito es participar en una colección de datos lógicamente ordenados de denominación «Estructura dinámica de datos» y cuyos tipos son pilas, colas, listas, grafos y árboles.

El «*Nodo*» tiene dos partes claves: el valor o conjunto de valores a almacenar y una autoreferencia de su mismo tipo, que puede estar en `null` o contener otro dato de su mismo tipo causando de esta forma un encadenamiento o colección de valores, una interpretación gráfica de este elemento es como sigue.

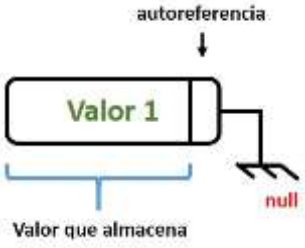

Figura 10 Representación clásica del elemento «Nodo».

1.5 CUESTIONARIO DE LA UNIDAD

1. **¿Cuáles de los siguientes tipos de datos primitivos en Java?**
   - byte, short, int
   - Integer, Double
   - Scanner, System.io.
   - String, Character, Double.
2. **¿Cuáles son los envoltorios de Java?**
   - byte, short, int
   - Integer.
   - Scanner, System.io.
   - String, Character, Double.
3. **¿Cuál es el operador que genera el concepto de instancia en Java?**
   - New.
   - Old.
   - Create.
   - Igualdad.
4. **A una referencia que no ha sido instanciada ¿Cómo se la conoce?**
   - Referencia nula.
   - Instancia.
   - Referencia vacía.
   - Clase de acceso estático.
5. **¿Cómo se llama a la clase que definida por un programador hace referencia dentro de su bloque de elementos a un dato de su mismo tipo?**
   - Estructura de Datos Dinámica.
   - Clase.
   - Arreglo dinámico.
   - Arreglo estático.
6. **¿Cuáles son los elementos básicos de un nodo?**
   - Dato por almacenar, auto referencia, métodos que lo integran.
   - Arreglos estáticos y dinámicos.
   - Operaciones fifo y lifo.
   - Bucles y condiciones.

7. **¿Cuál es la condición necesaria dentro de los principios de la programación orientada a objetos para que exista la sobreescritura?**
   - Herencia
   - Polimorfismo.
   - Encapsulamiento.
   - Abstracción.
8. **¿Cómo se define un dato abstracto en Java?**
   - Usando los principios de encapsulamiento, herencia y polimorfismo.
   - Con funciones.
   - Con variables y wrapper.
   - Con sobrecarga y polimorfismo.
9. **¿Cómo se llama al método con idéntico nombre que el de la clase?**
   - Constructor.
   - Método de arranque.
   - Destructor.
   - Función main.
10. **¿Cuál es la condición necesaria dentro de los principios de la programación orientada a objetos para que exista la sobrecarga?**
    - Encapsulamiento.
    - Polimorfismo.
    - Herencia.
    - Abstración.

2 FUNDAMENTOS DE ESTRUCTURA DE DATOS

En este capítulo se explica el dominio de los tipos de datos en Java, las características básicas de los tipos de datos abstractos y se ejemplifica como se debe estudiar una entidad con la finalidad de extraer su estado como variables y su comportamiento como funciones.

Objetivos:

- Reconocer las palabras reservadas del lenguaje Java.
- Identificar comportamiento y estado de un tipo de dato abstracto en particular.

2.1 ESPECIFICACIÓN DE TIPOS ABSTRACTOS DE DATOS

En términos simples se puede definir como un proceso mental, mediante el cual se extraen los rasgos esenciales de algo para representarlos por medio de un lenguaje gráfico o escrito. Debido a que es un proceso mental es una acción subjetiva y creativa, es decir, depende del contexto psicológico de la persona que la realiza.

En términos de la Programación Orientada a Objetos, constituye un principio donde enfatiza en detalles que son significativos y que suprime otros detalles que son por el momento irrelevante, y denota las características esenciales de un objeto que lo distinguen de todas las otras clases de objetos.

Un tipo de dato abstracto es un conjunto de datos u objetos al cual se le asocian operaciones. Este provee de una interfaz con la cual es posible realizar las operaciones permitidas, abstrayéndose de la manera en como estén implementadas dichas operaciones.

Esto quiere decir que un mismo Tipo de Dato Abstracto (TDA) puede ser implementado utilizando distintas estructuras de datos y proveer la misma funcionalidad.

El paradigma de programación orientada a objetos permite el encapsulamiento de los datos y las operaciones mediante la definición de clases e interfaces, lo cual

permite ocultar la manera en como ha sido implementado el TDA y solo permite el acceso a los datos a través de las operaciones provistas por la interfaz empleada.

Su importancia radica en que la abstracción es una capacidad para modelar una realidad por medio de las herramientas computacionales. UML (*Unified Modeling Language*) es un lenguaje de modelado y permite diagramar un TDA a través de una serie de esterotipos como son las clases. Una clase en UML especifica una especie de TDA. En el paradigma orientado a objetos, una clase es una plantilla que define los atributos y métodos que tendrán los objetos. Los atributos son datos miembros de una clase, donde se especifica el tipo de datos y su dominio. Las funciones son acciones u operaciones soportadas por los objetos que son instancia de la clase.

Por ello, la abstracción es una técnica o metodología que permite diseñar o modelar estructuras de datos que consiste en representar bajos ciertos lineamientos de formato las características esenciales de una estructura de datos. Además, es el proceso por el cual se resaltan o destacan los detalles relevantes para un cierto propósito mientras se ignoran u ocultan los irrelevantes para tal propósito.

Un tipo de dato abstracto (TAD) en un tipo de datos que especifica los tipos de datos almacenados, las operaciones definidas sobre esos datos y los tipos de los parámetros de esas operaciones. Un TAD define qué individuos representa dicho tipo (o pertenecen a), las operaciones que se pueden aplicar sobre tales individuos, la semántica y propiedades de tales operaciones.

Una interfaz es un conjunto de métodos abstractos cuya funcionalidad es la de determinar el funcionamiento de una clase, es decir, funciona como una plantilla. Los métodos al ser abstractos estos no tienen funcionalidad alguna, solo se definen su tipo, argumento y tipo de retorno. En un TAD sólo le interesará la interfaz que este posea. En lenguajes de programación como Java y Python un TDA puede ser expresado por interfaces, que es una simple lista de declaraciones de métodos.

El TAD al estar estar conformado por un conjunto válido de elementos y operaciones se puede estructurar de la siguiente forma:

1. Elementos del TAD.

2. Describe los atributos o características que contiene la estructura, además se identifican que tipo de datos le corresponde a cada atributo. Ejemplo, enteros, cadena, decimales, fecha, hora, etc.
3. Tipo de estructura u organización del TAD: pueden ser lineales, jerárquicas y red.
4. Dominio de la estructura: se detalla la capacidad de la estructura de datos en cuanto al rango posible de datos de cada elemento del TAD.
5. Documentación de método: se debe explicar el alcance del método.
6. Describir las operaciones del TAD.
7. Se componen del nombre de la operación, las entradas o listas de parámetros de la operación, el dato de salida. Se establece la sintaxis de un método.
8. nombre_operación (lista_parámetros): valor_de_salida

Se toma como ejemplo el lugar y fecha que se coloca en los oficios "Loja, 3 de diciembre de 2018". Se puede observar que existen algunas características en este texto, por ejemplo, el tipo de datos, el dominio o rango de valores y la estructura que representan estos datos. La ciudad es una estructura de dato lineal, de tipo cadena y su dominio será las ciudades del Ecuador. El día, mes y año son estructuras de datos lineales, de tipo lineal de y sus dominios en el primero es del 1 al 31, el segundo del 1 al 12 y el tercero del 1 al 2018. En el siguiente ejemplo ilustra la representación del lugar y fecha a través de un TAD.

Datos:

Lugar y fecha**:** Loja, 3 de diciembre de 2018

Abstracción de la Información

Tabla 1 Ejemplo de Tipo de dato Abstracto.

TAD: LUGAR_FECHA
1.- Elementos que conforman la estructura de datos
ciudad : cadena
día : entero
mes : entero
año : entero
2.- Tipo de organización
El tipo de organización es lineal
3.- Dominio
Entre {las ciudades del Ecuador}
Entre 1 al 31 el día,

Entre 1 al 12 el mes
Entre 1 al 2018 el año

2.2 NIVELES DE ABSTRACCIÓN DE DATOS

En una estructura de datos se define dos tipos de niveles, lógico y físico.

2.2.1 NIVEL LÓGICO

También se lo denomina nivel abstracto que define a una estructura de datos a través de la especificación lógica del TDA. Se identifican y describen las operaciones, elementos y tipo de organización que se relacionan que ejecutará la estructura. Este nivel es independiente de cualquier lenguaje de programación. En el paradigma orientado a objeto un TDA también se lo puede representar a través de UML.

2.2.2 NIVEL FÍSICO

Definido los elementos, operaciones y tipo de organización de la estructura, se decide el lenguaje de Programación en el que se va a implementar. Cada lenguaje de programación al estar compuesto con un conjunto de palabras reservadas y de seguir una sintaxis se debe traducir el diagrama abstracto a código de un deteminado leguaje de programación. Este nivel también se lo conoce como nivel de implementación.

En Java, un TAD es implícito al declarar una interfaz en una aplicación y se definen los métodos abstractos de la aplicación y las clases implementan las interfaces para darle funcionalidad a los métodos abstractos. Por ejemplo, si se ha especificado el TAD Teléfono para representar la abstracción de las funcionalidades que realiza este de un teléfono convencional o móvil, en la siguiente interfaz se definen los principales métodos:

Ejemplo 6 Interfaz Teléfono.

```
1  public interface Telefono {
2     // operaciones
3     public String llamar(int numero);
4     public void colgar();
5     public boolean estáOcupado();
6  }
```

```
1  public class TeléfonoConvencional implements Telefono
2     String dirección, numero;
3     boolean disponible = false;
4     public String llamar(int numero){
5        return "llamando al "+ numero;
6     }
7     public void colgar(){
8         System.out.println("colgando");
9     }
10    public boolean estáOcupado(){
11       if (disponible)
12          return true;
13       else
14          return false;
15    }
16 }
```

2.3 ESTRUCTURAS DE DATOS

Es una colección o agrupación de datos organizados que están asociados para verse como una unidad en su conjunto. Estas colecciones se construyen a partir de los tipos de datos simples o primitivos. Además, se pueden dividir en simples y compuestas.

Las estructuras de datos no solo representan la información, también tienen un comportamiento interno, se rige por ciertas reglas restricciones dadas por la forma en que esta construida internamente.

Las estructuras de datos permiten solucionar un problema de manera más sencilla gracias a que las reglas que las rigen nunca cambian, así que se puede asumir que ciertas cosas son siempre ciertas; si se usa lenguaje de programación como Java, se sabrá que se necesita definir el tamaño de los arreglos antes de ser usados.

Usando una estructura de datos, se puede hacer un *Array* de tamaño indeterminado.

La flexibilidad de las estructuras de datos permite su aprovechamiento de formas muy variadas. Un buen reflejo de ello lo aporta el grafo, que se trata de una versión en la cual los datos están conectados debido a la presencia de nodos.

Todos estos nodos no solo disponen de un valor determinado que les ha sido asignado, sino que cuentan con vínculos con otros de los nodos. La facilidad en la comunicación de estos nodos abre muchas posibilidades y es por lo que se usan en la representación de redes. Adicionalmente estas estructuras son dinámicas.

Las estructuras simples son homogéneas y hacen referencia a un solo valor a la vez por lo que ocupan una sola casilla de memoria. En Java se los denomina tipos de datos primitivos y son valores de tipo enteros, reales, lógicos y caracter.

Los datos numéricos enteros en Java son datos de tipo `byte`, `short`, `int`, `long`. Se declaran así:

```
1  int a = 1;
2  byte b = 2;
3  short c = 3;
4  int d = 4;
5  long e = 5;
```

Los datos de tipo decimal almacenan datos con punto flotante. Están compuesto tanto por la parte entera como la decimal. Hay dos tipos: float y double. En Java se declaran así:

```
1  float a = 5.5f;
2  double b = 3.0;
```

Los datos booleanos permiten almacenar valores lógicos, indican si es `true` o `false`. Cuando un atributo o variable no se fija un valor, el valor por defecto es `false`. Se declaran así:

```
1  boolean verdadero = true;
2  boolean falso = false;
```

Los datos tipo caracter sirven para almacenar letras, números, caracteres especiales, símbolos, etc. En Java se puede usar cualquier caracter del sistema UNICODE y se declaran así:

```
1  char caracter = 'a';
2  char caracter = 45;
```

En la siguiente tabla se describe los bytes que ocupan los diferentes tipos de datos:

Tabla 2 Tipos de datos primitivos.

Tipo	Bytes que ocupa
byte	8 bits
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes (tipos reales)
double	8 bytes
char	Sin signo 2 bytes
boolean	1 bit

Las estructuras compuestas es una colección de datos primitivos o de instancia que hace referencia a un grupo de casillas de memoria. En este tipo se puede describir a los arreglos, listas, grafos o árboles.

2.4 LA ESTRUCTURA DE DATOS Y SU RELACIÓN CON LOS LENGUAJES DE PROGRAMACIÓN

La estructura de Datos comprende la utilidad si está relacionado con saber también qué tipo de relación tienen con los lenguajes de programación. En este sentido hay que dejar totalmente de lado los lenguajes de bajo nivel, dado que no tienen soporte para estas estructuras.

Una buena demostración de este tipo de relación entre estructuras y lenguaje se puede ver en dos lenguajes muy específicos que se utilizan en la actualidad y que siguen siendo de gran rendimiento: Pascal y C.

Este tipo de soporte siempre ha ayudado a los programadores y con el paso del tiempo las propias plataformas que acompañan a los lenguajes modernos han ido dando soporte a estas estructuras de manera interna, permitiendo así que sigan presentes para aprovecharlas en sus respectivos contextos.

2.5 TIPOS DE ESTRUCTURAS DE DATOS

Las principales estructuras son:

- ***Arrays***: es la estructura de datos más común es el array lineal o array de una dimensión. Un array es una lista de números finitos de datos similares,

referenciados por medio de un conjunto de n números consecutivos, normalmente 1,2,3, 4, 5 … *n*.

- **Pila**: también denominada sistema último en ingresar – primero en salir (LIFO), es una lista lineal en la cual las inserciones y extracciones tienen lugar solo por un extremo llamado cúspide.
- **Cola**: también denominada sistema primero en ingresar - primero en salir (FIFO), es una lista lineal en la cual las extracciones se realizan siempre por un extremo llamado inicio y las inserciones por el extremo contrario llamado final.
- **Grafos**: los datos contienen, en algunos casos, relaciones entre ellos que no son necesariamente jerárquicos. La estructura de datos que refleja esta relación recibe el nombre de grafo.

2.6 OPERACIONES QUE SE PUEDEN REALIZAR CON ESTRUCTURAS DE DATOS

- **Recorrido:** implica el acceder a cada registro una única vez, aunque uno o más ítems del registro sean procesados.
- **Búsqueda:** implica la localización de un registro y esto se puede determinar por medio de una determinada clave o el acceso a todos los registros que cumplan una o más condiciones.
- **Inserción:** permite añadir nuevos registros a la estructura.
- **Eliminación:** permite realizar una acción de borrado de un registro de la estructura.
- **Ordenación:** sirve para clasificar los registros conforme a un orden lógico determinado (por ejemplo, ordenado de orden alfabético, o de manera ascendente, de acuerdo con una clave de caracteres, o numérica).

2.7 CUESTONARIO DE LA UNIDAD

1. ¿A través de qué elementos se define el estado de un tipo de dato abstracto?

- Constantes
- Variable
- Tipos primitos
- Wrapper

2. **¿A través de qué elementos se define el comportamiento de un tipo de dato abstracto?**

- Las funciones
- Los estados
- Las cadenas
- Los atributos

3. **¿De qué formas pueden ser la estructura y representación de un TAD?**

- Árbol
- Lineales
- Jerárquicas
- Red
- Grafos

4. **¿Cuáles son los tipos de datos enteros en Java?**

- boolean
- byte
- short
- int
- long.
- character

5. **¿Cómo se llama en el paradigma de la programación orientada a objetos al lenguaje que permite identificar y describir las operaciones, elementos y tipo de organización de un tipo de dato abstracto?**

- TAD
- UML
- DHL
- JAVA

6. **Enumere los tipos de estructura de datos**

- Arrays
- Pila
- Cola
- Grafos
- Matriz
- bucle

7. **La Pila es:**

- La operación que permite combinar dos archivos previamente ordenados en uno
- Una pila es una lista lineal en la cual las inserciones y extracciones tienen lugar solo por un extremo llamado cúspide
- Una acción de borrado de un registro de la estructura
- Una estructura lineal que permite enlazar al nodo raíz con el nodo cima

8. **Subraye lo correcto:**

Con las estructuras de datos se pueden realizar estas operaciones:

- Barrido
- Recorrido
- Supresión
- Búsqueda
- Inserción
- Copiado
- Eliminación
- Personalización
- Ordenación
- Combinación

9. La Ordenación:

- Sirve para clasificar los registros conforme a un orden lógico determinado (por ejemplo, ordenado alfabéticamente, o de manera ascendente, de acuerdo con una clave de caracteres, o numérica).
- Implica la localización de un registro y esto se puede determinar por medio de una determinada clave o el acceso a todos los registros que cumplan una o más condiciones.
- Implica el acceder a cada registro una única vez, aunque uno o más ítems del registro sean procesados.
- Permite aplicar buscar los datos en cada nodo y ubicarlos ascendentemente.

10. Seleccione lo correcto

Los datos booleanos permiten almacenar valores lógicos, indican si es true o false. Cuando un atributo o variable no se fija un valor, el valor por defecto es false. Se declaran así:

```
1  char caracter = 'a';
2  char caracter = 45;
```

```
1  boolean verdadero = true;
2  boolean falso = false
```

3 ESTRUCTURA DE DATOS BÁSICAS «Cadenas»

En este capítulo se explica los tipos de datos envoltorios básicos y estáticos, que son las cadenas, la importancia de conocer los métodos principales y explicar cómo funcionan, su comportamiento a través de ejemplos con código.

Objetivos:

- Diferenciar las cadenas explícitas e implícitas.
- Explicar el funcionamiento de los principales métodos de la clase *String*.

Las cadenas y arreglos son considerados como estructuras de datos básicos estáticas, debido a que no pueden incrementar más valores después de que son creadas.

3.1 MANIPULANDO CADENAS

Las cadenas son objetos especiales que están conformadas por un conjunto finita de caracteres. En Java las cadenas se las define a través de la clase *String* y es parte de las clases fundamentales. Para establecer que un valor es una cadena se debe utilizar la doble comilla, por ejemplo:

- "Manta"
- "14 de febrero"
- "# 89"

Se establecen dos formas para declarar cadenas:

- La primera forma es definir un atributo o variable de la clase *String*. Se la conoce como definición implícita. Por ejemplo:

```
1  String correo = "usuario@gmail.com";
2  String pais = "Ecuador";
3  String fecha = "28 de mayo de 1990";
```

- La segunda forma es crear un objeto directamente. Se la conoce como forma explícita

```
1  String correo = new String("usuario@gmail.com");
2  String pais = new String("Ecuador");
3  String fecha = new String("28 de mayo de 1990");
```

La clase *String* es un tipo de dato tanto primitivo y de clase que representa un conjunto de caracteres y en casi todas las clases se definen atributos, parámetros o variables con este tipo de dato. Ejemplo: las cédulas, nombres y apellidos de una persona, nombres de países, descripción de productos, etc.

3.2 CONCATENACIÓN

Para realizar la suma de dos números positivos se utiliza el operador aritmético '+', esta operación da como resultado un tercer número entero (2 + 4 = 6). Al aplicar el mismo principio en cadenas, el operador '+' permite combinar dos objetos cadena y el resultado es un nuevo objeto cadena ("2" + "4" = "24"). La concatenación también se puede aplicar si alguno de sus operandos es de tipo cadena (2 + 5 + "4" = "74").

Hay que conocer cuatro aspectos básicos del operador '+'

1. Si ambos operandos son datos numéricos, se realiza una operación de suma.
2. Si cualquiera de los operandos es una cadena, se realiza una operación de concatenación.
3. Si ambos operandos son cadenas, se realiza una operación de concatenación
4. Todas las expresiones se leen de izquierda a derecha.

Evaluemos los siguientes ejercicios

Ejemplo 7 Concatenación de cadenas.

```
1  public class Concatenacion {
2      public static void main(String[] args) {
3          System.out.println(5 + 7);
4          System.out.println("5" + 7 + 2);
5          System.out.println(7 + 2 + "5");
6          System.out.println("5" + "7");
7      }
8  }
```

La salida es:

```
1  12
2  572
3  95
4  57
```

En la línea 3 de la clase `Concatenación` los operandos son datos enteros positivos, por lo tanto, se realiza una operación de suma. En el ejemplo dos, (línea 4) la expresión

tiene dos operadores y aplicando las reglas de procedencia, se evalúa de izquierda a derecha, el primer operando es una cadena y el segundo es un entero, al ejecutar se unen ambos valores "57". Este valor se concatena con el tercer operando "57" + 2 = "572".

En el ejemplo 3 (línea 5), el primer operando se suma con el segundo 7 + 2 = 9 y el resultado entre ambos se concatena con el tercer operando 9 + "5" = "95". El cuarto ejemplo es una simple operación de concatenación dado que los dos operandos son cadenas.

La concatenación se puede realizar directamente a través del operador aritmético + y el operador de asignación = en un solo componente +=. En el siguiente ejemplo, en la línea 4, la instrucción m += " en" se puede definir m = m + " en" y el resultado es el mismo.

Ejemplo 8 Concatenación de cadenas a través de operadores de asignación.

```
1  public class Concatenacion {
2      public static void main(String[] args) {
3          String m = "programación";
4          m += " en";
5          m += " Java";
6          System.out.println(m);
7      }
8  }
```

La salida es:

```
1  programación en Java
```

3.3 MÉTODOS IMPORTANTES

Las cadenas al ser un tipo de dato importante en los programas es necesario conocer los principales métodos para el manejo de cadenas.

3.3.1 MÉTODO length()

Permite devolver el número de caracteres que compone una cadena o su tamaño. La definición de método es:

```
1  public int length()
```

En el siguiente ejemplo, se obtendrá el número de caracteres de la variable cadena al invocar al método `length`:

```
public class Cadenas {
    public static void main(String[] args) {
            String cadena = "Universidad Eloy Alfaro";
            System.out.println("Nro de caracteres: "
                                + cadena.length());
    }
}
```

Su salida es:

```
Nro de caracteres: 23
```

3.3.2 MÉTODO charAt()

Permite obtener un caracter específico de una cadena dependiendo de una posición. La definición del método es:

```
public char charAt(int index)
```

Al utilizar este método es necesario comprender que la cadena está conformada por una colección de caracteres organizados en una estructura lineal estática como un arreglo. Para acceder a cualquier caracter se utiliza un índice o posición, la primera posición de la cadena es 0 y la última posición es el número de caracteres menos uno.

- Índice del primer caracter de la cadena = 0
- Índice del último caracter = cadena.length() - 1

El siguiente ejemplo muestra cómo se utiliza el método charAt():

Ejemplo 9 Método charAt.

```
public class Cadenas {
    public static void main(String[] args) {
        String cadena = "uleam";
        System.out.println("Primer caracter: " +
                            cadena.charAt(0));
        System.out.println("Segundo caracter: " +
                            cadena.charAt(1));
        System.out.println("Tercer caracter: " +
                            cadena.charAt(2));
        System.out.println("Cuarto caracter: " +
                           cadena.charAt(3));
        System.out.println("Último caracter: " +
                           cadena.charAt(cadena.length() - 1));
    }
}
```

Su salida es:

```
Primer caracter: u
Segundo caracter: l
Tercer caracter: e
Cuarto caracter: a
Último caracter: m
```

Este ejemplo se ha definido una cadena asignándole el valor "uleam", se compone de 5 caracteres y para acceder a cada uno de ellos es a través de su posición o índice. En la siguiente figura se ilustra las partes que componen una cadena:

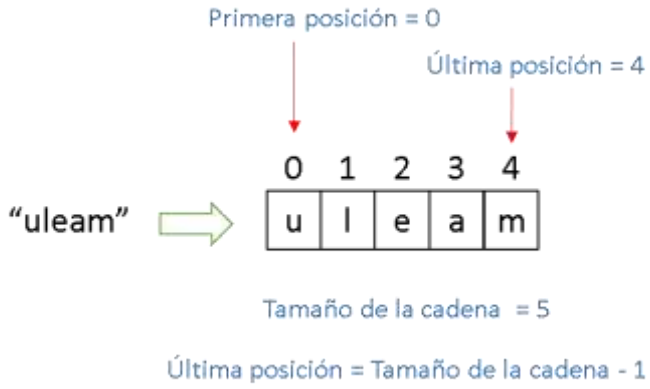

Figura 11 Estructura de una cadena.

Un problema común que tienen los programadores son las excepciones que se generan al acceder a un caracter mediante un índice que está fuera del rango de caracteres que componen la cadena. En el siguiente ejemplo, se trata de acceder a un caracter en la posición 6 en la cadena, el error que se genera es de ejecución "Java.lang.StringIndexOutOfBoundsException: String index out of range: 6":

Ejemplo 10 Método chartAt.

```
public class Cadenas {

    public static void main(String[] args) {
        String cadena = "uleam";
        System.out.println("caracter " + cadena.charAt(6));
    }
}
```

Su salida es:

```
Exception in thread "main"
Java.lang.StringIndexOutOfBoundsException: String index out of
range: 6
      at Java.lang.String.charAt(String.Java:658)
      at proyectolibro.ProyectoLibroJava.main(Cadena.Java:5)

```

3.3.3 MÉTODO toUpperCase() y toLowerCase()

El método `toUpperCase`() convierte la cadena de texto en mayúsculas mientras que el método `toLowerCase`() convierte la cadena de texto en minúsculas.
Convierten una cadena en mayúsculas o minúsculas.

La definición de los métodos es:

```
String toUpperCase(String str)
String toLowerCase(String str)
```

Ejemplo 11 Métodos toUpperCase y toLowerCase.

```
public class Cadenas {
    public static void main(String[] args) {
      String mensaje = "UleaM";
      // cadena en mayúsculas
      System.out.println(mensaje.toUpperCase());
      // cadena en minúsculas
      System.out.println(mensaje.toLowerCase());
    }
}
```

Su salida es

```
ULEAM
Uleam
```

3.3.4 MÉTODO equals() y equalsIgnoreCase()

El método `equals`() verifica si dos cadenas son idénticas o contienen los mismos caracteres devolviendo un valor verdadero o falso si no lo son. Mientras el método `equalsIgnoreCase`() tiene la misma función y comportamiento que el método *equals*, la única diferencia es que al momento de comparar no tiene en cuenta si las cadenas están en mayúsculas o minúsculas. La definición de los métodos es:

```
boolean equals(String str)
boolean equalsIgnoreCase(String str)
```

El siguiente ejemplo muestra la forma de utilizar los métodos:

Ejemplo 12 Método equals y equalsIgnoreCase.

```
public class Cadenas {
    public static void main(String[] args) {
        System.out.println("uleam".equals("uleam"));
        System.out.println("UlEaM".equals("uleam"));
        System.out.println("ULEAM".equalsIgnoreCase("uleam"));
    }
```

```
7 }
```

Su salida es:

```
1 true
2 false
3 true
```

En el ejemplo 12, línea 4 las dos cadenas son iguales y devolverá un valor `true`. En la línea 5, las cadenas tienen el valor semántico pero diferentes por las mayúsculas y minúsculas y devolverá un valor false. En la línea 6, el método ignora si las cadenas están en mayúsculas o minúsculas, pero deben tener el mismo valor semántico.

3.3.5 MÉTODO compareTo()

Compare dos cadenas, pero toma en cuenta el orden alfabético del código ASCII. Si las cadenas son iguales devuelva un valor 0, si la primera cadena es mayor devuelva un valor positivo (1). Si la segunda cadena es mayor devuelva un valor negativo (-1). La definición del método es:

```
1 int compareTo(String str)
```

Ejemplo:

Ejemplo 13 Método compareTo.

```
1 public class Cadenas {
2     public static void main(String[] args) {
3         System.out.println("a".compareTo("A"));
4         System.out.println("A".compareTo("a"));
5         System.out.println("a".compareTo("a"));
6     }
7 }
```

Su salida es:

```
1 32
2 -32
3 0
```

3.3.6 MÉTODO concat()

Aplicar la misma función que el operador +, permitiendo unir cadenas. A continuación se realiza esta operación de concatenación entre dos cadenas. La definición del método es:

```
1 String concat(String str)
```

Ejemplo:

Ejemplo 14 Método para concatenar cadenas.

```
1  public class Cadenas {
2      public static void main(String[] args) {
3          String cadena1 = "Universidad Laica";
4          String cadena2 = "Eloy Alfaro de Manabí";
5          String cadena3 = cadena1.concat(cadena2);
6          System.out.println(cadena3);
7      }
8  }
```

Su salida es:

```
1  Universidad LaicaEloy Alfaro de Manabí
```

3.3.7 MÉTODO replace()

Permite buscar y reemplazar un caracter por otro. Para que la cadena almacene los nuevos cambios, se debe asignar la cadena reemplazada. La definición de los métodos es:

```
1  String replace(char oldChar, char newChar)
2  String replace(CharSequence oldChar, CharSequence newChar)
```

Ejemplo:

Ejemplo 15 Método para reemplazar cadenas.

```
1  public class Cadenas {
2      public static void main(String[] args) {
3          String mensaje = "Universidad Laica Eloy Alfaro de
4                            Manabí";
5          System.out.println("cadena: " + mensaje);
6          mensaje = mensaje.replace('a', '@');
7          System.out.println("cadena nueva: " + mensaje);
8      }
9  }
```

Su salida es:

```
1  cadena: Universidad Laica Eloy Alfaro de Manabí
2  cadena nueva: Universid@d L@ic@ Eloy Alf@ro de M@n@bí
```

3.3.8 MÉTODO trim()

Elimina los espacios al inicio y final de la cadena. Este método devuelve una cadena. La definición del método es:

```
1  public String trim()
```

Ejemplo:

Ejemplo 16 Método para eliminar espacios.

```
1  public class Cadenas {
2      public static void main(String[] args) {
3          System.out.println("   Universidad    ");
4          System.out.println("   Universidad    ".trim());
5      }
6  }
```

Su salida es:

```
1     Universidad
2  Universidad
```

3.3.9 MÉTODO split()

Permite convertir una cadena en un arreglo a partir de un caracter. Es decir, se necesita que exista un caracter que sirva como separador entre los elementos del arreglo. Es importante recordar que un arreglo es una estructura lineal estática que permite almacenar datos del mismo tipo. Para definirlo se utiliza [ ].

La definición del método es:

```
1  public String [] split(char character)
```

Ejemplo:

Ejemplo 17 Método para descomponer una cadena en arreglos.

```
1  public class Cadenas {
2
3      public static void main(String[] args) {
4          String mensaje = "Universidad Laica Eloy Alfaro de
5                            Manabí";
6          String [] arregloCaracteres = mensaje.split(" ");
7          for (int i = 0; i < arregloCaracteres.length; i++) {
8              System.out.println("arreglo [" + i + "] = "
9                                 + arregloCaracteres [i]);
10         }
11     }
12 }
```

Su salida es:

```
1  arreglo [0] = Universidad
2  arreglo [1] = Laica
3  arreglo [2] = Eloy
4  arreglo [3] = Alfaro
5  arreglo [4] = de
6  arreglo [5] = Manabí
```

3.3.10 MÉTODO indexOf()

Devuelve la posición o índice al encontrar una cadena o un caracter, siempre devuelve la posición de la primera coincidencia. Si la cadena de búsqueda no exista se devuelve un valor de -1. La definición del método es:

```
1  int indexOf(char ch)
2  int indexOf(char ch, index fromIndex)
3  int indexOf(String str)
4  int indexOf(String str, index fromIndex)
```

Ejemplo:

Ejemplo 18 Método para encontrar cadenas o caracteres.

```
1  public class Cadenas {
2      public static void main(String[] args) {
3          String mensaje = "universidad";
4          System.out.println(mensaje.indexOf('e'));
5          System.out.println(mensaje.indexOf("ar"));
6          System.out.println(mensaje.indexOf('n', 9));
7          System.out.println(mensaje.indexOf("iv", 1));
8      }
9  }
```

En el primer ejemplo (línea 5) el método busca el caracter 'e' en la cadena e inicia desde la posición 0 y la coincidencia se encuentra en la posición 4. En el segundo ejemplo (línea 6) busca la cadena "ar" y al no encontrarse devuelve el valor – 1. En el tercer ejemplo (línea 7) la búsqueda inicia desde la posición 9 y cuarto ejercicio (línea 8) la búsqueda inicia desde la posición 1.

Su salida es:

```
1  4
2  -1
3  -1
4  2
```

3.3.11 MÉTODO contains()

Permite buscar uno o más caracteres en una cadena. Si la búsqueda es exitosa devuelve un valor booleano verdadero caso contrario un valor falso. La definición del método es:

```
1  boolean contains(String str)
```

Ejemplo:

Ejemplo 19 Método para buscar cadenas

```
1  public class Cadenas {
2      public static void main(String[] args) {
3          System.out.println("uleam".contains("a")); // true
4          System.out.println("uleam".contains("E")); // false
5      }
6  }
```

Su salida es:

```
1  True
2  false
```

3.3.12 MÉTODO subString()

Permite buscar y devolver parte de una cadena dependiendo de la posición inicial y final. A partir de este método se puede obtener una porción de una cadena respecto de otra.

La definición del método es:

```
1  int substring(int beginIndex)
2  int substring(int beginIndex, int endIndex)
```

Ejemplo:

Ejemplo 20 Método para manejo de subcadenas.

```
1  public class Cadenas {
2      public static void main(String[] args) {
3          String mensaje = "Universidad Laica Eloy Alfaro";
4          System.out.println(mensaje.substring(23));
5          System.out.println(mensaje.substring(
6                                  mensaje.indexOf('L')));
7          System.out.println(mensaje.substring(12, 17));
8      }
9  }
```

El método `subString` tiene dos variantes, la primera (línea 5 y 6) solo con la posición de inicio, esto supone que la posición final será hasta el último valor de la cadena. La segunda se puede obtener una subcadena desde una posición de inicio y una final (línea 7).

Al ejecutar se imprime:

```
1  Alfaro
2  Laica Eloy Alfaro
```

```
3   Laica
```

1.3.9. MÉTODO valueOf()

Permite convertir valores que no son cadena en cadena. Analizar el siguiente ejemplo:

Ejemplo 21 Método para convertir un tipo de datos a cadenas.

```
1   public class Cadenas {
2       public static void main(String[] args) {
3           Date fecha = new Date();
4           System.out.println(String.valueOf(fecha));
5           System.out.println(String.valueOf(9087));
6           System.out.println(String.valueOf(9.987));
7           System.out.println(String.valueOf(false));
8       }
9   }
```

Con el método "`valueOf`" se puede transformar un tipo de datos a cadena, por ejemplo, las fechas, números enteros, decimal, datos booleanos, etc. Es útil para imprimir el contenido de un objeto.

Su salida es:

```
1   Thu Mar 08 06:06:13 ECT 2018
2   9087
3   9.987
4
5   False
```

Además, se pueden utilizar los siguientes métodos para transformar diferentes tipos de datos en cadenas:

```
1   String valueOf(boolean b);
2   String valueOf(int i);
3   String valueOf(long l);
4   String valueOf(float f);
5   String valueOf(double d);
6   String valueOf(Object obj);
```

1.4. CLASE StringBuilder

Los objetos de la clase *String* son elementos inmutables esto significa que cualquier operación en un *String* no altera el contenido del objeto, sino que se crea nuevo valor transformado, en otras palabras, se crea una nueva instancia. Las concatenaciones con cadenas se las realiza con el operador "+".

Analizar el siguiente código:

```
1  public class Cadena {
2      public static void main(String[] args) {
3          String cadena = "";
4          cadena+="La ";
5          cadena+="casa ";
6          cadena+="es ";
7          cadena+="mía.";
8          System.out.println(cadena.toString());
9      }
10 }
```

En este ejemplo se crea un objeto tipo *String* llamada cadena, pero al concatenar se crea una nueva instancia implícitamente. En conclusión se han creado 5 instancias porque el objeto *String* es un elemento inmutable y el objeto cadena inicia con un valor "" y en la línea 4 se le asigna un nuevo objeto "`La `". En la línea 5 se le asigna un nuevo objeto "`La casa `" y la secuencia sigue en dos siguientes asignaciones. Al crear nuevos objetos se asignan más recursos en memoria RAM en nuestros computadores que dependiendo de las operaciones con las cadenas puede volverse en unas operaciones ineficientes.

Para ello, se creó la clase `StringBuilder` y similar al *String* pero se puede fijar su tamaño, y su contenido puede modificarse sin la necesidad de crearse nuevas instancias en cada operación de agregación o concatenación. Además, un `StringBuilder` no es inmutable y esta indexado, esto ayuda que el rendimiento del manejo de cadenas de caracteres sea más eficiente.

Sus métodos principales son:

Tabla 1 Métodos principales de la clase StringBuilder.

Métodos	**Descripción**
append(String cadena)	Permite añadir cadenas de caracteres y retorna la misma instancia.
capacity( )	Devuelve la capacidad del *StringBuilder*.
length( )	Devuelve el número de caracteres del *StringBuilder*.
delete(int start, int end) deleteCharAt(int index)	Método opuesto al insert, elimina un conjunto de caracteres desde una posición inicial y final.
insert(int offset, String str)	Agrega caracteres en una determinada posición, devuelve el mismo objeto *StringBuilder*.
reverse( )	Permite invertir el orden de los caracteres del *StringBuilder*, devuelve la misma instancia.
toString( )	Permite convertir la instancia a un *String*.

Analizar el siguiente código:

```
1  public class DemostracionStringBuilder {
2      public static void main(String[] args) {
3          StringBuilder cadena = new StringBuilder();
4          cadena.append("la ");
5          cadena.append("casa ");
6          cadena.append("es ");
7          cadena.append("mía.");
8          System.out.println(cadena.toString());
9      }
10 }
```

En la línea 3 se crea un objeto de la clase `StringBuilder` y en la línea 4 llama al método `append()` que permite añadir varias cadenas utilizando el mismo objeto, evitando crear instancias implícitas. Si lo ejecutamos nos dará la siguiente salida:

Su salida es:

```
1  \programas_libro> Java DemostracionStringBuilder
2  la casa es mía
```

Existe tres formas de crear objetos StringBuilder:

```
1  StringBuilder cadena1 = new StringBuilder();
2  StringBuilder cadena2 = new StringBuilder("uleam");
3  StringBuilder cadena3 = new StringBuilder(20);
```

En el primer caso se crea un objeto con caracteres vacíos. En el segundo caso se crea el objeto con un conjunto de caracteres iniciales o un valor predefinido. En el último caso se crea un objeto con un tamaño específico de 20 caracteres. Es necesario tener en cuenta que los tres objetos se puede modificar los valores con los método `append`().

Recuerda
Los métodos charAt(), indexOf(), length(), and substring() en un StringBuilder trabajan exactamente que un String.

3.4 EJEMPLOS

Ejercicio 1: Crear un programa que permita obtener cada caracter que componen una cadena de texto.

Solución: Para obtener cada caracter de la cadena se utiliza el método `charAt`.

```java
public class Ejecutor {
    public static void main(String[] args) {
        String provincia = "Manabí";
        for (int i = 0; i < provincia.length(); i++) {
            System.out.println("posición " + i + " " +
                               provincia.charAt(i));
        }
    }
}
```

Su salida es:

```
posición 0 M
posición 1 a
posición 2 n
posición 3 a
posición 4 b
posición 5 í
```

Ejercicio 2: Crear un programa que permita ingresar una frase y el programa debe presentar cuántas palabras de una letra, de dos letras, de tres letras, de cuatro letras y de cinco o más letras existen en el texto ingresado. La frase sólo se debe ingresar letras.

Solución:

```java
import Java.util.Scanner;

public class Palabras {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        System.out.println("Ingrese una frase");
        String frase = lector.nextLine();
        String [] arreglo = frase.split(" ");
        int una = 0, dos = 0, tres = 0, cuatro = 0, cinco = 0;
        for (int i = 0; i < arreglo.length; i++) {
            switch (arreglo[i].length()){
                case 1:
                    una++;
                    break;
                case 2:
                    dos++;
                    break;
                case 3:
                    tres++;
                    break;
                case 4:
                    cuatro ++;
                    break;
                default:
                    cinco++;
            }
        }
        System.out.println("frase: " + frase);
        System.out.println("1 frase: " + una);
        System.out.println("2 frases: " + dos);
        System.out.println("3 frases: " + tres);
        System.out.println("4 frases: " + cuatro);
        System.out.println("+ 5 frases: " + cinco);
```

```
34      }
35  }
```

Su salida es:

```
Ingrese una frase
Java es un lenguaje de programación de propósito general,
concurrente, orientado a objetos, que fue diseñado
específicamente para tener tan pocas dependencias de
implementación como fuera posible
frase: Java es un lenguaje de programación de propósito general,
concurrente, orientado a objetos, que fue diseñado
específicamente para tener tan pocas dependencias de
implementación como fuera posible
1 frase: 1
2 frases: 5
3 frases: 3
4 frases: 3
5 frases: 15
```

3.5 CUESTIONARIO DE LA UNIDAD

11. ¿Cuál es la salida del siguiente ejercicio?

```
1  StringBuilder objeto = new StringBuilder();
2  objeto.append("aa").insert(0, "bb").insert(4, "cc");
3  System.out.println(objeto);
```

- ccbbaa
- aabbcc
- bbccaa
- ccaabb
- aaccbb
- bbaacc

12. ¿Cuál es la salida del siguiente ejercicio?

```
1  String cadena1 = "uleam";
2  StringBuilder cadena2 = new StringBuilder("uleam");
3  if (cadena1 == cadena2)
4       System.out.print("1");
5     if (cadena1.equals(cadena2))
6               System.out.print("2");
```

- 1 y 2
- Excepción
- 1
- Error de compilación
- 2
- false

13. ¿Cuál es la salida del siguiente ejercicio?

```
1  public class Gato {
2       public void maullar(String m1, StringBuilder m2){
3           m1.concat("au");
4           m2.append("au");
5       }
6  }
7
8  public class LibroED {
```

```
9       public static void main(String[] args) {
10            String miau1 = "mi";
11            StringBuilder miau2 = new StringBuilder("mi");
12            Gato g =new Gato();
13            g.maullar(miau1, miau2);
14            System.out.println(miau1 + " " + miau2);
15      }
16  }
```

- mi miau
- mi mi
- mirau miau
- Excepción
- miau mi
- Error de compilación

14. ¿Cuál es la salida del siguiente ejercicio?

```
1  String letters = "portoviejo";
2  System.out.println(letters.length());
3  System.out.println(letters.charAt(2));
4  System.out.println(letters.charAt(7));
5  System.out.println(letters.charAt(10));
```

- Error de Compilación
- r i o
- o i o
- Error de Ejecución
- o v o
- 10 r o

15. ¿Cuál es la salida del siguiente ejercicio?

```
1  String letras = "abcdefghi";
2  System.out.print(letras.substring(1, 3));
3  System.out.print(letras.substring(5, 5));
4  System.out.print(letras.substring(7));
```

- aei
- bchi
- Error de compilación
- Espacio en blanco
- bcdfgh
- Error de Ejecución
- abci

16. ¿Cuál es la salida del siguiente ejercicio?

```
1  String frase = "elemento";
2  frase.toUpperCase();
3  frase.trim();
4  frase.substring(0, 2);
5  frase += " vacio";
6  System.out.println(frase.length());
```

- Error de compilación
- 10
- 15
- Error de Ejecución
- 0
- 14

17. ¿Cuál es la salida del siguiente ejercicio?

```
1  String a = "";
2  a += 6;
3  a += "Si";
4  a += true;
5  if (a == "6Sitrue")
6        System.out.println("iguales");
7  if (a.equals("6Sitrue"))
8        System.out.println("idénticos");
```

- Error de compilación en la línea 4
- 6Sitrue
- iguales
- idénticos
- Error de Ejecución
- Error de compilación en la línea 5

18. ¿Cuál es la salida del siguiente ejercicio?

```
1  int cantidad = 0;
2  StringBuilder objeto = new StringBuilder("abcdefg");
3  cantidad += objeto.substring(1, 2).length();
4  cantidad += objeto.substring(6, 6).length();
5  cantidad += objeto.substring(5, 6).length();
6  System.out.println(cantidad);
```

- 0
- 1
- Error de compilación
- Error de Ejecución
- 2
- 3

19. ¿Cuál es la salida del siguiente ejercicio?

```
1  StringBuilder cadena = "universidad";
2  cadena.append(4).deleteCharAt(3).delete(3, cadena.length() - 1);
3  System.out.println(cadena);
```

- uni
- Error de compilación
- univer
- universi
- universidad
- Error de ejecución

20. ¿Cuál es la salida del siguiente ejercicio?

```
1  StringBuilder caracteres = new StringBuilder("abcdefghi");
2  caracteres.delete(2,  8);
3  caracteres.append("-").insert(2, "+");
   System.out.println(caracteres);
```

- Espacio en blanco
- +-abi
- ab+-i
- Error de compilación
- Error de ejecución
- ab+i-

4 ESTRUCTURA DE LINEAL ESTÁTICA «Arreglos»

En este capítulo se explica los tipos de datos lineales estáticos denominados arreglos unidimensionales y bidimensionales. Además, se explica las operaciones básicas de definir, crear, escribir y leer. Se aborda un ejercicio para explicar su comportamiento y funcionalidad.

Objetivos:

- Explicar las operaciones fundamentales en un arreglo.
- Definir las partes que componen a los arreglos.

Las cadenas de texto *String* o `StringBuilder` son estructuras de datos lineales que almacenan un conjunto de caracteres y se podrían definir como arreglos implícitos. Sin embargo, los arreglos no solo se limitan a almacenar caracteres sino diferentes tipos de datos primitivos (enteros, decimales, booleanos, etc.) u objetos (*Date*, Persona, etc.). Además, permite agrupar un conjunto de valores siempre que sean del mismo tipo de datos.

Los arreglos pueden clasificarse como unidimensionales o bidimensionales.

4.1 ARREGLOS UNIDIMENSIONALES

Forman una estructura en una sola dimensión. En otras palabras, definen una secuencia de elementos del mismo tipo. Para utilizar esta estructura, partiremos con su declaración a través de la sintaxis:

tipo_dato [ ] *identificador_arreglo*;
o
tipo_dato *identificador_arreglo* [ ];

Ejemplos:

```
1  double [ ] calificaciones;
2  int [ ] edades;
```

En ambos casos se utiliza los corchetes [ ] e indica al programa que se está declarando un arreglo y almacenará en el primer caso datos decimales y en el segundo caso enteros. La forma simple para crear un arreglo es:

identificador_arreglo = **new** tipo_datos [*tamaño*]
o
tipo_dato [ ] *identificador_arreglo* = **new** tipo_datos [*tamaño*]

Ejemplos:

```
1  double [ ] calificaciones = new double[8];
2  int [ ] numeros = new int double[5];
```

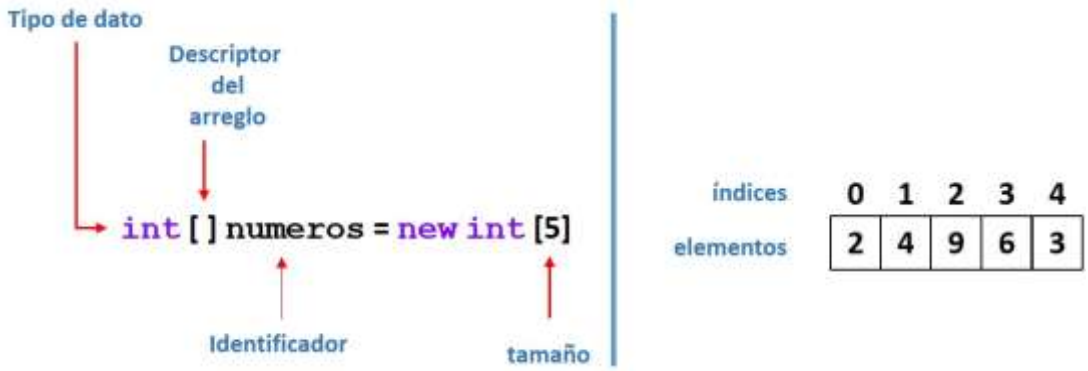

Figura 12 Estructura de un arreglo unidimensional.

En el gráfico anterior se observa que los arreglos almacenan datos enteros y su tamaño es 5. Pero hay que analizar dos partes fundamentales, los índices y elementos. Cada índice presenta la posición de un elemento en el arreglo y los elementos son datos que se almacenan en una determinada posición. Al inicializar el arreglo, se fijan valores por defecto dependiendo del tipo de dato declarado, en este caso los valores son cero. Sin embargo, sí un arreglo es inicializado con objetos se inicializarán con valores nulos.

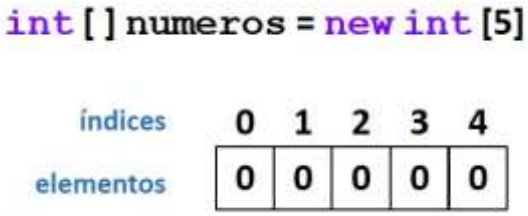

Figura 13 Definición e Inicialización de un arreglo unidimensional.

Otra forma para implementar un arreglo es especificar sus valores. Su sintaxis es:

tipo_dato [] *identificador_arreglo* = { *valor1*, *valor2*, *valor3*, *valor4*, ... }

Ejemplos:

```
1  int [] numeros = {8, 7, 5, 6};
```

Definido y creado el arreglo, se debe escribir o llenar de elementos al arreglo y se lo realiza indicando su posición y el valor que se ingresará. En la figura se indica la estructura para acceder a un elemento del arreglo:

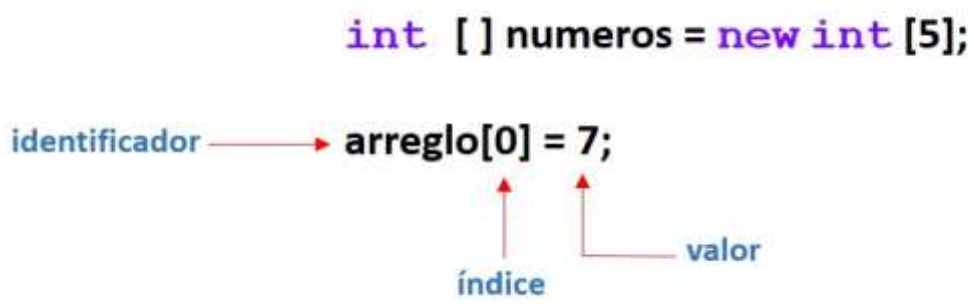

Figura 14 Escritura de un arreglo.

Ejemplo:

```
1  int numeros [] = new int [5];
2  numeros[0] = 56;
3  numeros[1] = 7;
4  numeros[2] = 68;
5  numeros[3] = 989;
6  numeros[4] = 95;
```

Las estructuras de control repetitivas `for` permiten realizar los recorridos de los arreglos.

```
1  public class Arreglo {
2      public static void main(String[] args) {
3          int [] numeros = {8, 7, 5, 6};
4          for (int i = 0; i < numeros.length; i++) {
5              System.out.println(arreglo[i]);
6          }
7      }
8  }
```

Su salida es:

```
1  9
2  7
3  7
4  0
5  55
6  98
```

4.2 ARREGLOS BIDIMENCIONALES

Son arreglos de dos dimensiones, que establecen filas y columnas para formar una matriz. Para utilizar esta estructura es necesario establecer su definición a través de la siguiente sintaxis:

tipo_dato [ ] [ ] *identificador*;
o
tipo_dato *identificador* [ ] [ ];

Ejemplos:

```
1  int [][] numeros;
2  double [][] precios;
```

En ambos casos se utiliza los corchetes [ ][ ] e indica al programa que se está declarando un arreglo bidimensional y almacenará en el primer caso datos enteros y en el segundo caso decimales. La forma simple para crear una matriz es:

Tipo_dato [ ] [ ] *identificador* = **new** tipo_datos [*filas*] [*columnas*]

Figura 15 Estructura de un arreglo bidireccional.

Ejemplos:

```
1  int [][] numeros = new int [4][5];
2  double [][] notas = new double [5][4];
```

Al crear una matriz, los tipos de datos definidas en ella se inicializan con valores por defecto. En los dos ejemplos anteriores los valores por defecto son 0.

En Java, si una matriz es creada con datos primitivos siempre se inicializarán con su valor por defecto, ejemplo si una matriz es de enteros sus valores por defecto serán cero. Si el tipo de dato declarado no es primitivo, el valor por defecto será null.

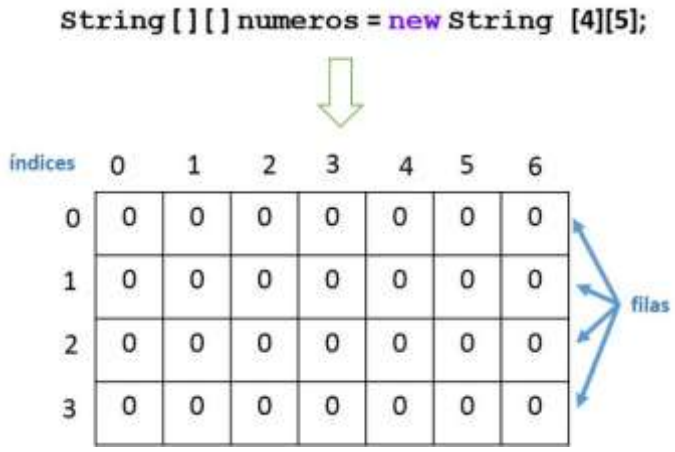

Figura 16 Representación de un arreglo bidimensionale.

Las operaciones básicas de las matrices son:

- Escritura
- Lectura

En la escritura de la matriz se indica la posición de la fila y la columna para asignarle un nuevo valor, como se indica en el siguiente ejemplo:

```
1  int numeros [][] = new int [4][5];
2  numeros [3][1] = 7;
3  numeros [4][2] = 52;
```

En la operación de lectura, es necesario indicar cuál es la posición de elemento a leer a través de los índices que indiquen la fila y columna como se ilustra en el siguiente ejemplo:

```
1  System.out.println(numero[3][1]); // devuelvo 7
2  System.out.println(numero[4][2]); // devuelvo 52
```

En términos normales la lectura de la matriz se realiza mediante dos estructuras repetitivas anidadas for.

```
1  public class Matríz {
2      public static void main(String[] args) {
3          int matriz [][] = {{35,4},{28,42},{54,20}};
4          for (int i = 0; i < matriz.length; i++)
5              for (int j = 0; j < matriz[0].length; j++)
6                  System.out.println("posición [" + i + "][" + j
7                                      + "] = " + matriz[i[j]);
8      }
9  }
```

Su salida es:

```
1  Java Matriz
2  posición [0][0] = 35
```

```
posición [0][1] = 4
posición [1][0] = 28
posición [1][1] = 42
posición [2][0] = 54
posición [2][1] = 20
```

4.3 CUESTIONARIO

1. ¿Cuáles definiciones sobre arreglos son incorrectas?

- Java.util.Random[] aleatorios[] = new Java.util.Random[4][];
- int[][] valores = new int[];
- double[][] cadenas = new double[][];
- int[][] numero = new int[5][];
- String[][][] letras = new String[3][0][5];
- boolean estados[] = new estados[2];

2. ¿Cuáles ítems generan error?

```
double[] arreglo = new double[2];
```

- int longitud = arreglo.length;
- int longitud = arreglo.length();
- int longitud = arreglo.size;
- int longitud = arreglo.size();
- int longitud = arreglo.capacity;
- int longitud = arreglo.capacity();

3. Analice el siguiente código y seleccione la respuesta correcta

```
int[] random = { 6, 5, 2, 40, 1 };
ramdom.add(4);
```

- Error de ejecución
- Se inserta el valor de 4 en la última posición
- El arreglo debe utilizar el método append()
- Los arreglos no tienen el método add.
- El arreglo se debe definir su tamaño ante de agregar valores
- Se agrega un valor en la posición 4

4. Analice el siguiente código y seleccione la respuesta correcta

```
int[] arreglo = { 206, 58, -2, 77, 8};
arreglo[6] = 2;
```

- Se genera una exception porque no existe la posición 6
- Se inserta el valor 2 en la posición 6
- Se genera un error de compilación
- Se reemplaza el valor del último elemento por 2

- Todos los valores se inicializan con 2
- Se agrega un nuevo valor en la última posición

5. **Analice el siguiente código y seleccione la respuesta correcta**

```
1  int[] arreglo = new int [5];
```

- Los valores del arreglo es 5
- Los índices inician de 1 al 5
- El arreglo se inicializa con valores por defecto 0
- Los valores por defecto del arreglo es null
- El arreglo estará compuesto por 6 elementos
- El último índice del arreglo es 6

6. **Analice el siguiente código y seleccione la respuesta correcta**

```
1  int[] arreglo = {2,5,8,4};
2  for(int i = 0; i <= arreglo.length; i++){
3        System.out.println(arreglo[i]);
4  }
```

- Imprime los valores del arreglo 2 5 8 4
- No imprime ningún valor porque no se ejecuta el for
- Imprime los valores de 2 5 8
- Error de compilación
- Error de ejecución
- Imprime los valores del arreglo de forma inversa 4 8 5 2

7. **¿Cuál es el resultado de las siguientes instrucciones?**

```
1  int[] arreglo = {2,5,8,4,10};
2  System.out.println(arreglo[4]);
```

- Muestra el contenido del cuarto elemento
- No muestra ningún valor
- Muestra el valor por defecto 0
- Muestra el valor del quinto elemento
- Error de ejecución
- Error de compilación

8. **Si un arreglo es declarado con 7 elementos. Los valores de sus índices son:**
   - 1..7
   - 1..6
   - 0..7
   - 0..6
   - 1..8
   - 0..8

9. **Qué estructura se recomienda utilizar para recorrer un arreglo unidimensional o bidimensional?**
   - while
   - do while
   - for
   - foreach
   - if
   - switch

10. **Cuál es la salida del siguiente código?**

```
1  int[] arreglo = {2,5,7,8,9,4,1};
2  for(int i = 3; i == arreglo.length; i++)
3        System.out.print(i + " ");
```

- nada
- 2 5 7 8 9 4 1
- 5 7 8 9 4 1
- 7 8 9 4 1
- 8 9 4 1
- Error de compilación
- Error de ejecución

5 ESTRUCTURA DE DATOS DINÁMICA LINEAL «PILA»

La pila o *stack* en su denomiación inglesa, es una de las estructuras de datos dinámicas lineales más sencilla, se asemeja al principio lógico de apilar algo, como si se tuviese un conjunto de elementos que se apilan distingue 3 elementos básicos: la cima o tope será el último elemento que llegó, la cabecera fue el primer elemento en llegar y la operación de retirar que sigue el principio LIFO (*Last In Firs Out*) último en entrar primero en salir.

Objetivos:

- Identificar las propiedades de la estructura de datos dinámica pila.
- Programar las operaciones básicas de la estructura de datos dinámica pila.
- Desarrollar un programa usando la estructura de datos dinámica.

5.1 CONCEPTO

La Pila es una estructura lineal que se forma con el tipo de dato abstracto Nodo. El último elemento que se ingresa en una pila es el primero en salir, de allí su asociación con el método de inventario LIFO «*Last In, First Out*» «Último en entrar, Primero en salir».

Las Operaciones básicas son:
- Insert → Push
- Delete → Pop
- Search → Tope

La representación gráfica de la pila es:

Figura 17 Estructura de una pila

5.2 FUNCIONAMIENTO LÓGICO DE LA PILA

La primera referencia existente será de tipo **Nodo** con un nombre descriptivo respecto a la estructura de datos dinámica lineal que representa y estará inicializada en **null,** la denominación comúnmente usada es **Pila.**

Figura 18 Abstracción para representación de una referencia iniciada en null.

Cuando se inserte un nodo nuevo en la Pila, en el caso de que este sea el primero la referencia null será reemplazada por el elemento nodoNuevo.

Figura 19 Creación del primer elemento de la Pila.

Cuando se inserte un nodo nuevo en la Pila, después de existir ya el primero, ese nodo nuevo almacenará la referencia Pila en su propiedad enlace, a la vez que se convierte en la primera referencia existente.

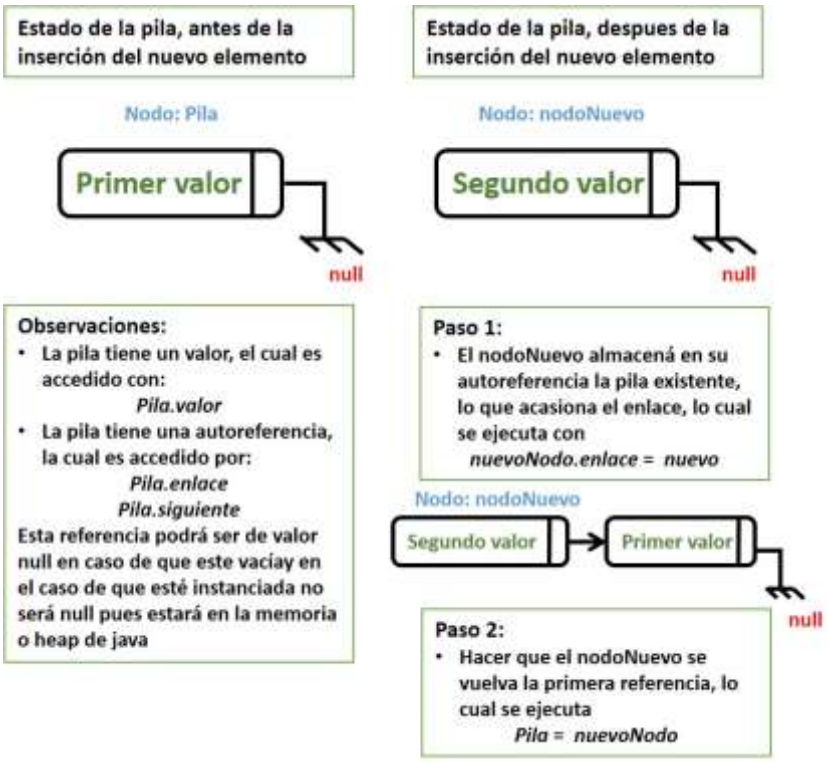

Figura 20 Proceso de inserción en la Pila.

De la imagen resultante es importante destacar que existe:

- Una referencia de denominación Pila.
- Una autoreferencia la cual es anónima pues se accede a ella a través `Pila.enlace`.

La anterior forma de acceso implica que cuando se quiera imprimir:

- El primer valor, se lo hará con la siguiente instrucción: `Pila.valor`
- El segundo valor, se lo hará con la instrucción: `Pila.enlace.valor`

La operación de extracción de una pila clásicamente denominada «***pop***», es destructiva, lo cual implica que, al extraer un elemento de la estructura dinámica, el mismo debe ser eliminado.

La operación de búsqueda no es destructiva, en tanto que informa si un elemento en particular se encuentra dentro de la estructura dinámica.

5.3 PROGRAMACIÓN TIPO CONSOLA PARA UNA PILA

La consola como elemento universal dentro del mundo informático, está representada por una ventana que puede ocupar parte del monitor o su extensión completa, y permite entradas y salidas básicas, esto quiere decir, entradas por teclado y salidas por pantalla.

En Java una forma sencilla de representar la entrada básica por teclado es el uso de la clase *Scanner* disponible dentro del paquete Java.util, y la representación de la salida básica se realiza mediante la sentencia `System.out.println`.

Se desarrollará un menú para trabajar con el programa de pila tipo consola.

```
1   public static void main(String[] args){
2         System.out.println("\n\n\t\t\t========Menú Pila" +
3                            "========");
4         System.out.println("\t\t\t=                              =");
5         System.out.println("\t\t\t= 1- Insertar elmento        =");
6         System.out.println("\t\t\t= 2- Eliminar elmento        =");
7         System.out.println("\t\t\t= 3- Buscar elmento          =");
8         System.out.println("\t\t\t= 4- Imprimir elmento        =");
9         System.out.println("=====================================");
10        System.out.println("\t\t\tIngrese la Opción");
11        Scanner lector = new Scanner(System.in);
12        int opcion = lector.nextInt();
13  }
```

5.3.1 ALGORITMOS DE SOBRECARGA PARA EL CONSTRUCTOR DEL TIPO ABSTRACTO NODO

El código de este método pertenece a la clase NodoPila.

```
1   public NodoPila(){
2         //Sobrecarga del constructor, para no recibir parámetros
3         dato="";
4         enlace = null;
5   }
6
7   public NodoPila(String dato){
8         //Sobrecarga del constructor, para recibir parámetros
9         this.dato = dato;
10        enlace = null;
11  }
```

Con respecto a estas instrucciones, se resume su funcionamiento:

Se considera el caso de una instanciación sin parámetros en la cual el dato será seteado como una cadena de caracteres vacía y la autoreferencia en **null**, la línea de invocación es **new** NodoPila();

Se considera el caso de una instanciación con parámetros en la cual se pasa como argumento un valor, el dato será seteado a la cadena de caracteres que corresponda y la autoreferencia en **null**, la línea de invocación es **new** NodoPila("Cadena");

5.3.2 ALGORITMO PARA COMPROBAR SI LA PILA ESTÁ VACÍA

El código de este método pertenece a la clase NodoPila.

```
1  public static boolean pilaVacia(NodoPila Pila){
2        if (Pila==null)
3              return true;
4        return false;
5  }
```

Con respecto a estas instrucciones, se resume su funcionamiento:

El método obligatoriamente debe ser estático «**static**», esto debido a que la referencia Pila inicialmente está seteada en **null**, lo cual quiere decir que no existen métodos ni propiedades disponibles en el momento de arranque del programa, devuelve **true** si la estructura de datos dinámica este vacía, y **false** en caso contrario.

5.3.3 ALGORITMO DE INSERCIÓN DE LA PILA (PUSH)

El código de este método pertenece a la clase NodoPila.

```
1  public static NodoPila insertarPila(NodoPila Pila, String dato){
2        NodoPila nodoNuevo=new NodoPila(dato);
3        if(NodoPila.pilaVacia(Pila))
4              Pila=nodoNuevo;
5        else{
6              nodoNuevo.enlace=Pila;
7              Pila=nodoNuevo;
8        }
9        return Pila;
10 }
```

Con respecto a estas instrucciones, se resume su funcionamiento:

- El método obligatoriamente debe ser estático «**static**», esto debido a que la referencia Pila inicialmente está seteada en **null**, lo cual quiere decir que no

existen métodos ni propiedades disponibles en el momento de arranque del programa, por ello se utiliza un método estático que puede ser invocado desde la clase `NodoPila` sin necesidad de instanciar previamente.

- Un método estático cuando va a ser usado a nivel de instancias obligatoriamente debe ser fijado en su tipo de retorno en el mismo tipo de la instancia, por ello se coloca `NodoPila` después de **`static`**, y además se ubica como parámetro del método a la Pila sobre la que se hará la inserción.
- El tipo de dato que se usa para almacenar en cada elemento de la estructura de datos dinámica es *String*, por ello se ubica como parámetro del método.
- El método estático `pilaVacia` retornará un tipo booleano, **`true`** si la estructura de datos dinámica está vacía, **`false`** en caso contrario.
- En el inicia de la pila la misma está en **`null`**, lo cual implica que no está referencia y no existe dentro de la memoria ram, el código que se ejecuta en el verdadero del if se asegura de que la pila inicie por primera vez.
- El lado **`false`** del **`if`** se ejecuta desde el segundo nodo para la estructura de datos dinámica, la línea `nodoNuevo.enlace=Pila;` causa que el nodoNuevo tenga almacenado en su enlace la Pila que existe previamente, y la línea `Pila=nodoNuevo` hace que el nodoNuevo se convierta en el elemento inicial, cumpliendo de esta forma con el principio LIFO (*Last in, First out*), pues el último nodo ingresado se vuelve el primero.

5.3.4 ALGORITMO DE ELIMINACIÓN DE LA PILA (POP)

- El código de este método pertenece a la clase `NodoPila`.

Ejemplo 22 Método para eliminar una pila.

```
1  public static NodoPila eliminarPila(NodoPila Pila){
2        if (NodoPila.pilaVacia(Pila)){
3              System.out.println("La pila no contiente
4                                  elementos..");
5              return Pila;
6        }
7        System.out.println("Se eliminó del tope de la pila a: " +
8                           Pila.dato);
9        Pila=Pila.enlace;
10       return Pila;
11 }
```

Con respecto a estas instrucciones, se resume su funcionamiento:

- Se comprueba si la pila está vacía con el método `pilaVacia`.
- Se deben eliminar elementos de la Pila suprimiéndolos desde el último ingresado, dado que el último elemento ingresado se vuelve el primero en el método `insertarPila`, entonces la línea `Pila=Pila.enlace;` actualiza la estructura dinámica de datos reduciendo en uno su longitud.

5.3.5 ALGORITMO DE IMPRESIÓN DE LA PILA

El código de este método pertenece a la clase NodoPila.

Ejemplo 23 Método para imprimir los datos cada nodo de la pila.

```
1  public static void imprimirPila(NodoPila Pila){
2        NodoPila temporal=Pila;
3        if(NodoPila.pilaVacia(temporal))
4              System.out.println("La pila no contiene
5                                  elementos..");
6        else{
7              int numeroElemento=0;
8        while(temporal!=null){
9              System.out.printf("\nImprimiento elemento Nro. %d: %s
10                                \n",numeroElemento,temporal.dato);
11             numeroElemento++;
12             temporal=temporal.enlace;
13             }
14       }
15 }
16
```

Con respecto a estas instrucciones, se resume su funcionamiento:

- Es común utilizar un elemento temporal para recorrer una estructura de datos dinámicas, aunque esta costumbre deriva de lenguajes de programación como c++, si se quitase el elemento temporal, el algoritmo es perfectamente funcional.
- La ejecución por el lado verdadero del **`if`** indica que la Pila esta vacía.
- La ejecución por el lado falso utiliza como alternativa una variable numeroElemento para contabilizar el número de elementos disponible en la estructura de datos dinámica, el ciclo **`while`** hará un recorrido desde la referencia inicial hasta las subsiguientes autoreferencias e imprimirá los datos almacenados en cada una de ellas, la línea `temporal=temporal.enlace;` permite el desplazamiento de temporal a lo largo de toda la Pila.

5.3.6 ALGORITMO DE BÚSQUEDA DE LA PILA

El código de este método pertenece a la clase `NodoPila`.

```
1  public static void buscarPila(NodoPila Pila, String
2                                datoBusqueda){
3        NodoPila temporal=Pila;
4        if(NodoPila.pilaVacia(temporal))
5              System.out.println("La pila no contiente
6                                  elementos..");
7        else{
8              int numeroElemento=0;
9        while(temporal!=null){
10             if(datoBusqueda.equals(temporal.dato)){
11                   System.out.printf("\nEl elemento %s está en la
12                                      posición: %d de la pila \n",
13                                     datoBusqueda,numeroElemento);
14                   break;
15             }
16             numeroElemento++;
17             temporal=temporal.enlace;
18             }
19       }
20  }
```

Con respecto a estas instrucciones, se resume su funcionamiento:

- Se utiliza el elemento nodo temporal para recorrer la lista.
- Se utiliza la variable `numeroElemento` para almacenar la posición dentro de la estructura dinámica en la que está el valor buscado.
- Dentro del ciclo **`while`** se avanza a temporal a lo largo de la estructura dinámica con la línea `temporal=temporal.enlace`; se hace dentro de cada ciclo la comprobación de si el elemento es el buscado con la sentencia `datoBusqueda.equals(temporal.dato)`

5.4 PROGRAMACIÓN GRÁFICA PARA UNA PILA

La programación tipo consola provee de un esquema nutritivo para la lógica de los profesionales de la información, el saber como se ejecuta el código desde su entrada básica hasta su salida básica dota de un fuerte entendimiento de las soluciones desarrolladas, sin embargo, se plantea el siguiente ejercicio a modo gráfico utilizando los algoritmos planteados, esto a propósito de tener aún más perspectiva del funcionamiento completo de la Pila, se propone para desarrollo la siguiente interface.

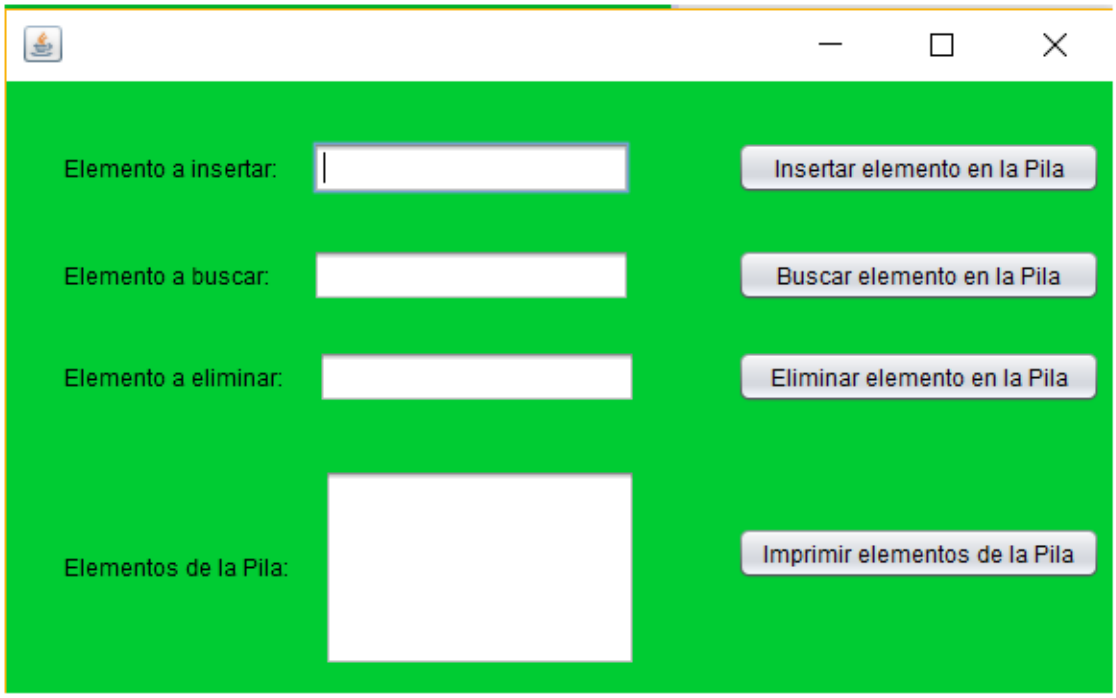

Figura 21 Programación gráfica para la estructura de datos dinámica pila.

5.5 OTRA FORMA DE IMPLEMENTAR LA PILA

El cógido se divide en tres clases:

- Nodo: será el puntero de la pila que contendrá el dato y la referencia al siguiente nodo.
- Pila: tendrá todas las operaciones de apilar (*PUSH*), desapilar (*POP*), verificar si está vacía la pila, busca un elemento en la pila e imprimir todos los elementos de la pila
- EjecutorPila: Es la clase Ejecutora dónde se inicia todo el programa.

Ejemplo 24 Clase Nodo Abstracto.

```
1  public class Nodo{
2      public String dato;
3      public Nodo siguiente;
4
5      public Nodo() {
6          siguiente = null;
7          dato = null;
8      }
9  }
```

Ejemplo 25 Clase Pila.

```
public class Pila {
    public Nodo top;

    public Pila() {
        top = null;
    }

    public boolean esVacia( ){
        if(top == null)
            return true;
        else
             return false;
    }

    public void hacePush(String d) {
        Nodo temp = new Nodo();
        temp.dato = d;
        temp.siguiente = top;
        top=temp;
        System.out.println( "push(" + top.dato + ")" );
    }

    public void hacePop() {
        if (esVacia())
            System.out.print("No hay elementos en la pila");
        else{
            System.out.println( "pop(" + top.dato + ")" );
            top=top.siguiente;
        }
    }

    public String buscar(){
        if(esVacia())
            return null;
        else
          return top.dato;
    }

    public void imprimir(){
        Nodo aux;
        aux = top;
        System.out.print("Pila: [");
        while(aux != null){
            if(aux.siguiente != null)
                System.out.print(aux.dato + ",");
            else
                System.out.print(aux.dato);
            aux=aux.siguiente;
        }
        System.out.print("]\n");
    }
}
```

Ejemplo 26 Clase Ejecutora.

```
public class EjecutorPila {
    public static void main(String[] args) {
        Pila p = new Pila();
        p.imprimir();
```

```
5           p.hacePush("Rene");
6           p.imprimir();
7           p.hacePush("Oscar");
8           p.imprimir();
9           p.hacePush("Mike");
10          p.imprimir();
11          p.hacePush("Robert");
12          p.imprimir();
13          System.out.println("Elemento de la cima de la pila es:" +
14                             p.buscar());
15          p.hacePush("Xavier");
16          p.hacePush("Denisse");
17          p.imprimir();
18          System.out.println("Elemento de la cima de la pila es:" +
19                             p.buscar());
20          p.hacePop();
21          p.imprimir();
22          p.hacePop();
23          p.imprimir();
24          p.hacePop();
25          p.imprimir();
26          p.hacePop();
27          p.imprimir();
28          p.hacePush("Vilka");
29          p.imprimir();
30      }
31  }
```

Su salida es:

```
1   Pila: []
2   push(Rene)
3   Pila: [Rene]
4   push(Oscar)
5   Pila: [Oscar,Rene]
6   push(Mike)
7   Pila: [Mike,Oscar,Rene]
8   push(Robert)
9   Pila: [Robert,Mike,Oscar,Rene]
10  Elemento de la cima de la pila es:Robert
11  push(Xavier)
12  push(Denisse)
13  Pila: [Denisse,Xavier,Robert,Mike,Oscar,Rene]
14  Elemento de la cima de la pila es:Denisse
15  pop(Denisse)
16  Pila: [Xavier,Robert,Mike,Oscar,Rene]
17  pop(Xavier)
18  Pila: [Robert,Mike,Oscar,Rene]
19  pop(Robert)
20  Pila: [Mike,Oscar,Rene]
21  pop(Mike)
22  Pila: [Oscar,Rene]
23  push(Vilka)
24  Pila: [Vilka,Oscar,Rene]
```

5.6 CUESTIONARIO DE LA UNIDAD

1. ¿Cuáles son los elementos básicos de la estructura de datos lineal pila?

- Cima o tope, cabecera y operación de retirar.
- Método fifo.
- La definición de su nombre.
- Método lifo.

2. ¿Cuál es la dirección de memoria con la que esta una pila inicialmente?

- Null.
- La de una referencia vacia.
- La de una referencia nula.
- Una dirección en formato hexadecimal

3. ¿Qué tipo de referencia se utiliza para recorrer una estructuar de datos dinámica pila?

- Referencia temporal.
- Referencia original.
- Referencia nula
- Referencia vacia.

4. Con respecto al algoritmo de impresión de la pila, ¿Qué indica la condición temporal!=null?

- Que se avanza en el recorrido hasta que la pila está vacía.
- Solo es una comprobación ejecutada una vez
- Es una condición que se ejecuta del lado verdadero del if.
- Es una condición que incrementa el número de elementos de la pila.

5. ¿Cómo se realiza la eliminación de elementos en una pila?

- Suprimiéndolos desde el último ingresado.
- Suprimiéndolos desde el primer ingresado.
- Suprimiéndolos en orden aleatorio.
- Suprimiéndolos en orden fifo.

6. ¿Por qué se define como estático el método insertarPila?

- Debido a que la referencia Pila inicialmente está seteada en null.
- Para coherencia de retorno entre instancia e invocación del método.
- Por coherencia en el almacenamiento de la instancia y el retorno del método.
- Por principios de la programación orientada a objetos.

7. Con respecto al algoritmo de búsqueda de la pila, ¿la firma del método es?

- NodoPila Pila, String datoBusqueda.
- NodoPila, String.
- Pila, datoBusqueda.
- buscarPila.

8. **¿En una estructura de datos dinámica pila con dos nodos, cuantas direcciones de memoria existen?**

   - 3.
   - 2.
   - 1.
   - 0.

9. **¿En una estructura de datos dinámica pila con dos nodos cuantas referencias existenten en memoria?**

   - 1.
   - 0.
   - Ninguna.
   - El nombre con el que haya sido definida.

10. **¿En una estructura de datos dinámica pila con tres nodos cuantas autoreferencias existenten en memoria?**

    - 2.
    - 1.
    - 0.
    - 3.

6 ESTRUCTURA DE DATOS DINÁMICA LINEAL «COLA»

La cola o *queue* en su denomiación inglesa, es una de las estructuras de datos dinámicas lineales se asemeja al principio lógico de encolar algo que distingue 3 elementos básicos: la cima o tope será el último elemento que llegó, la cabecera fue el primer elemento en llegar y la operación de retirar que sigue el principio FIFO (*First In Firs Out*) Primero en entrar primero en salir.

Objetivos:

- Identificar las propiedades de la estructura de datos dinámica cola.
- Programar las operaciones básicas de la estructura de datos dinámica cola.
- Desarrollar un programa usando la estructura de datos dinámica.

6.1 CONCEPTO

La Cola es una estructura lineal que se forma con el tipo de dato abstracto Nodo. El primer elemento que se ingresa en una pila es el primero en salir, de allí su asociación con el método de inventario FIFO «*First In, First Out*» «Primero en entrar, Primero en salir».

Las operaciones básicas son:

- Insert → Encolar
- Delete → Desencolar
- Search → Tope

La representación gráfica de la cola es:

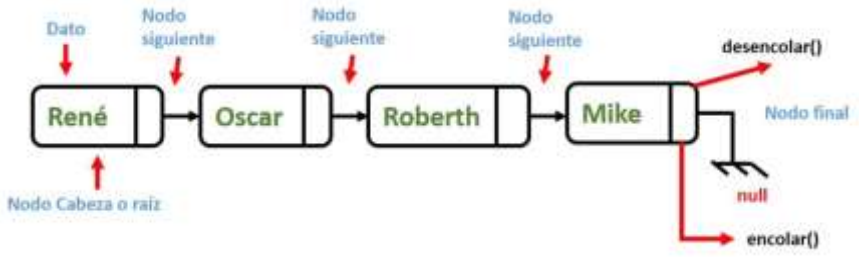

Figura 22 Representación de una cola

6.2 FUNCIONAMIENTO LÓGICO DE LA COLA

- La primera referencia existente será de tipo Nodo con un nombre descriptivo respecto a la estructura de datos dinámica lineal que representa y estará inicializada en null, la denominación comúnmente usada es Cola.

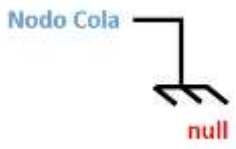

Figura 23 Abstracción para representación de una referencia iniciada en null.

- Cuando se inserte un nodo nuevo en la Cola, en el caso de que este sea el primero la referencia null será reemplazada por el elemento nodoNuevo.

Figura 24 Creación del primer elemento de la cola.

- Cuando se inserte un nodo nuevo en la Cola, después de existir ya el primero, ese nodo nuevo será almacenado en el último enlace de la estructura dinámica de datos, sin modificar nunca la primera referencia existente.

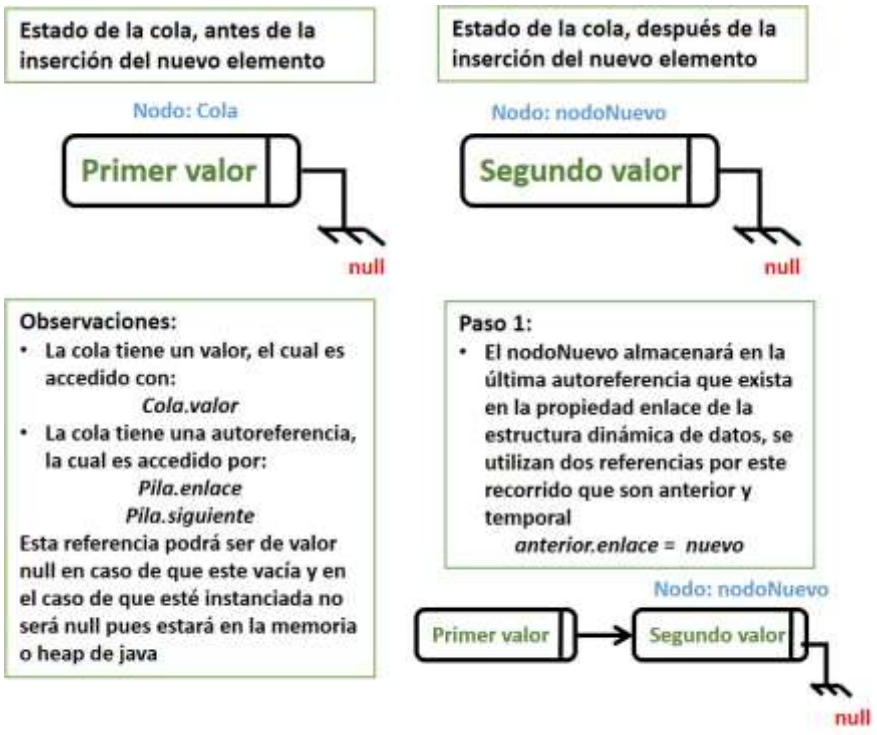

Figura 25 Proceso de inserción en la cola

De la imagen resultante es importante destacar que existe:

- Una referencia de denominación Cola.
- Una autoreferencia la cual es anónima pues se accede a ella de la forma `Cola.enlace`.

La anterior forma de acceso implica que cuando se quiera imprimir:

- El primer valor, se lo hará con la siguiente instrucción: `Cola.valor`
- El segundo valor, se lo hará con la instrucción: `Cola.enlace.valor`

La operación de extracción de una pila clásicamente denominada «*pop*», es destructiva, lo cual implica que, al extraer un elemento de la estructura dinámica, el mismo debe ser eliminado.

La operación de búsqueda no es destructiva, en tanto que informa si un elemento en particular se encuentra dentro de la estructura dinámica.

Los siguientes métodos detallan el código avanzado para realizar las operaciones de una cola utilizando la clase preconstruida *QUEUE* y el tipo de dato abstracto nodo.

6.2.1 ALGORITMO PARA ENCOLAR BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 27 Método encolar.

```
1  public static NodoCola encolar(NodoCola cola){
2        String valoringresado;
3        Scanner teclado = new Scanner(System.in);
4        System.out.println("Ingrese valor: ");
5        Valoringresado = teclado.nextLine();
6        NodoCola nodoNuevo = new NodoCola(valoringresado);
7
8        if(NodoCola.BuscarCola(valoringresado, cola)){
9              System.out.println("Elemento repetido... operación
10                                imposible");
11             return cola;
12       }
13
14       if(cola == null)
15             cola = nodoNuevo;
16       else{
17             NodoCola anterior = cola;
18             NodoCola temporal = cola;
19             while(temporal != null){
20                anterior = temporal;
21                temporal = temporal.siguiente;
22             }
23             anterior.siguiente = nodoNuevo;
24       }
25       System.out.println("Elemento insertado en la cima...");
26       return cola;
27       }
28 }
```

6.2.2 ALGORITMO PARA ENCOLAR BASADO EN LA CLASE PRECONSTRUIDA QUEUE

Ejemplo 28 Método encolar a través de la clase Queue.

```
1  public static Queue encolar(Queue cola){
2        Scanner teclado = new Scanner(System.in);
3        System.out.println("Ingrese valor: ");
4        String valoringresado=teclado.nextLine();
5
6        if(ColaPreconstruida.BuscarCola(valoringresado, cola)){
7              System.out.println("Elemento repetido... operación
8                                 imposible");
9              return cola;
10       }
11
12       cola.add(valoringresado);//1°Método de queue
13       System.out.println("Elemento ingresado en la cima..");
14       return cola;
14  }
```

6.2.3 ALGORITMO PARA DESENCOLAR BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 29 Desencolar un nodo.

```
public static Queue desencolar(NodoCola cola){
      if(cola == null){
            System.out.println("Cola vacia... operación
                              imposible");
            return cola;
      }
      System.out.println("Elemento desencolado: "+cola.valor);
      cola = cola.siguiente;
      return cola;
}
```

6.2.4 ALGORITMO PARA DESENCOLAR BASADO EN LA CLASE PRECONSTRUIDA QUEUE

Ejemplo 30 Desencolar a través de la clase Queue.

```
public static Queue desencolar(Queue cola){
      System.out.println("Elemento desencolado: "+cola.remove());
      return cola;
}
```

6.2.5 ALGORITMO PARA IMPRIMIR BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 31 Método para imprimir elementos de la cola.

```
public static void ImprimirCola(NodoCola cola){
      NodoCola temporal = cola;
      if(cola == null)
            System.out.println("cola vacia... no se puede
                              imprimir");
      else{
            while(temporal != null){
                  System.out.println("Elemento:
                  " + temporal.valor);
                  temporal = temporal.siguiente;
            }
      }
}
```

6.2.6 ALGORITMO PARA IMPRIMIR BASADO EN LA CLASE PRECONSTRUIDA QUEUE

Ejemplo 32 Método para imprimir a través de la clase Queue.

```
public static void ImprimirCola(Queue cola){
      if (cola.isEmpty())
```

```
            System.out.println("cola vacia... no se puede
                               imprimir");
      else
            System.out.println("Cola: " + cola.toString());
}
```

6.2.7 ALGORITMO PARA BUSCAR BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 33 Método para buscar un nodo.

```
public static boolean BuscarCola(NodoCola cola){
      String valorParaBuscar;
      Scanner teclado = new Scanner(System.in);
      System.out.println("Ingrese valor a Buscar: ");
      valorParaBuscar = teclado.nextLine();

      NodoCola temporal = cola;
      while(temporal != null){
            if (valorParaBuscar.equals(temporal.valor))
                  return true;
            temporal = temporal.siguiente;
      }
      return false;
}
```

6.2.8 ALGORITMO PARA BUSCAR BASADO EN LA CLASE PRECONSTRUIDA QUEUE

Ejemplo 34 Buscar nodo a través de la clase Queue.

```
public static boolean BuscarCola(Queue cola){
      Scanner teclado = new Scanner(System.in);
      System.out.println("Ingrese valor a buscar: ");
      String valorBuscado = teclado.nextLine();
      if(cola.contains(valorBuscado))
            return true;
      return false;
}
```

6.3 OTRA FORMA DE IMPLEMENTAR UNA COLA

El cógido se divide en tres clases:

- **Nodo**: será el puntero de la pila que contendrá el dato y la referencia al siguiente nodo.

- **Cola**: tendrá todas las operaciones de encolar, descolar, verificar si está vacía la cola, buscar un elemento en la cola, vaciar la cola e imprimir todos los elementos de la pila.
- **EjecutorCola**: Es la clase Ejecutora que da inicio al programa.

Ejemplo 35 Clase Nodo.

```
1  public class Nodo{
2      public String dato;
3      public Nodo siguiente;
4
5      public Nodo() {
6          siguiente = null;
7          dato = null;
8      }
9  }
```

Ejemplo 36 Clase Cola.

```
1  public class Cola {
2      public Nodo top;
3      public Nodo last;
4
5      public Cola( ){
6          top = last = null;
7      }
8      public Cola(String dato){
9          this.top.dato = dato;
10         this.top.siguiente = null;
11     }
12
13     public boolean esVacia( ){
14         if(top == null)
15             return true;
16         else
17             return false;
18     }
19
20     public void vaciar( ){
21         top = last = null;
22     }
23
24     public void desencolar( ){
25         if (esVacia( )){
26             System.out.print("\n Error al desencolar ...");
27         }else{
28             System.out.print("\n Descencolar: " + top.dato);
29             top = top.siguiente;
30         }
31     }
32
33     public void encolar(String d){
34         Nodo temp = new Nodo();
35         temp.dato = d;
36         if (esVacia( ))
37             top = last = temp;
38         else{
```

```
            last.siguiente = temp;
            last = temp;
        }
        System.out.print("\n Encolar: " + last.dato);
    }

    public void imprimir(){
        Nodo aux;
        aux=top;
        System.out.print("\n Cola : [");
        while(aux!=null){
            if(aux.siguiente != null)
                System.out.print(aux.dato + ",");
            else
                System.out.print(aux.dato);
            aux=aux.siguiente;
        }
        System.out.print("]");
    }
}
```

```
public class EjecutorCola {
    public static void main(String[] args)throws Exception {
        Cola c= new Cola();
        c.imprimir();
        c.encolar("Roberth");
        c.imprimir();
        c.encolar("René");
        c.imprimir();
        c.encolar("Oscar");
        c.imprimir();
        c.encolar("Mike");
        c.imprimir();
        c.desencolar();
        c.imprimir();
        c.desencolar();
        c.imprimir();
        c.desencolar();
        c.imprimir();
        c.desencolar();
        c.imprimir();
        c.desencolar();
        c.imprimir();
        c.desencolar();
        c.imprimir();
        c.encolar("Denisse");
        c.imprimir();
        c.imprimir();
        c.encolar("Edison");
        c.imprimir();
        c.desencolar();
        c.imprimir();
    }
}
```

Su salida es:

```
Cola : []
Encolar: Roberth
Cola : [Roberth]
Encolar: René
```

```
5   Cola : [Roberth,René]
6   Encolar: Oscar
7   Cola : [Roberth,René,Oscar]
8   Encolar: Mike
9   Cola : [Roberth,René,Oscar,Mike]
10  Descencolar: Roberth
11  Cola : [René,Oscar,Mike]
12  Descencolar: René
13  Cola : [Oscar,Mike]
14  Descencolar: Oscar
15  Cola : [Mike]
16  Descencolar: Mike
17  Cola : []
18  Error al desencolar ...
19  Cola : []
20  Encolar: Denisse
21  Cola : [Denisse]
22  Cola : [Denisse]
23  Encolar: Edison
24  Cola : [Denisse,Edison]
25  Descencolar: Denisse
26  Cola : [Edison]
```

6.4 CUESTIONARIO DE LA UNIDAD

1. **¿Cuál es el primer elemento que se retirará en un tipo de dato abstracto cola?**
   - El último elemento que llegó.
   - El primer elemento que llegó.
   - El orden es irrelevante.
   - El que determine el método fifo.

2. **¿Cuál es la dirección de memoria con la que está una cola inicialmente?**
   - Null.
   - La de una referencia vacía.
   - La de una referencia nula.
   - Una dirección en formato hexadecimal.

3. **¿Qué tipo de referencia se utiliza para recorrer una estructua de datos dinámica cola?**
   - Referencia temporal.
   - Referencia original.
   - Referencia nula
   - Referencia vacía.
   -

4. **Con respecto al algoritmo de impresión de la cola, ¿Qué indica la condición temporal!=null?**
   - Que se avanza en el recorrido hasta que la cola está vacía.
   - Solo es una comprobación ejecutada una vez
   - Es una condición que se ejecuta del lado verdadero del if.
   - Es una condición que incrementa el número de elementos de la cola.

5. **¿Cómo se imprime en una cola basada en la clase preconstruida QUEUE?**

   - Utilizando el método toString.
   - Utilizando un ciclo while.
   - Utilizando el método isEmpty.
   - Utilizando una estructura condicional cualquiera.

6. **¿Por qué se define como estático el método insertarCola?**

   - Debido a que la referencia Cola inicialmente está seteada en null.
   - Para coherencia de retorno entre instancia e invocación del método.
   - Por coherencia en el almacenamiento de la instancia y el retorno del método.
   - Por principios de la programación orientada a objetos.

7. **¿Cuál es el método que se utiliza para insertar en una cola basada en la clase preconstruida QUEUE?**

   - Add.
   - Añadir.
   - Enqueue.
   - Update.

8. **¿Por qué se asocia el tipo de dato abstracto cola al método de inventario fifo?**

   - El primer elemento que se ingresa en una cola es el primero en salir.
   - El primer elemento que se ingresa en una cola es el último en salir.
   - Por su definición de métodos en la clase preconstruida QUEUE.
   - Por su algoritmo de ingreso.

9. **¿Qué pasa con el elemento nodoNuevo en el algoritmo de inserción de la cola?**

   - En el caso de que este sea el primero la referencia null será reemplazada por el elemento nodoNuevo.
   - Será puesto en la cima de la cola.
   - Será puesto en la cabecera de la cola.
   - Reemplazará a todos los elementos guardados en la cola.

10. **¿En qúe método de la cola basada en QUEUE se utiliza el método remove?**

    - Desencolar.
    - Encolar.
    - Imprimir
    - Modificar.

7 ESTRUCTURA DE DATOS DINÁMICA LINEAL «LISTA»

La lista o *Linkedlist* en su denomiación inglesa, es una de las estructuras de datos dinámicas lineales se asemeja al principio lógico de encolar algo que distingue 3 elementos básicos: la cima o tope será el ultimo elemento que llegó, la cabecera fue el primer elemento en llegar y la operación de retirar que sigue los principios FIFO o LIFO según convenga.

Objetivos:

- Identificar las propiedades de la estructura de datos dinámica lista.
- Programar las operaciones básicas de la estructura de datos dinámica lista.
- Desarrollar un programa usando la estructura de datos dinámica lista.

7.1 CONCEPTO

La Lista es una estructura lineal que se forma con el tipo de dato abstracto nodo o con la clase preconstruida *LinkedList*, es entendida fácilmente por como si fuese una lista de supermercado, una persona podrá empezar comprando los productos iniciales pero si surgen nuevas prioridades podrá desplazarse por su voluntad a comprar productos de en medio o final de su lista, es así que es una estructura de datos dinámica muy flexible en tanto que el orden de inserción y extracción de elementos esta sujeto a mucha flexibilidad.

La representación gráfica de una lista simple es:

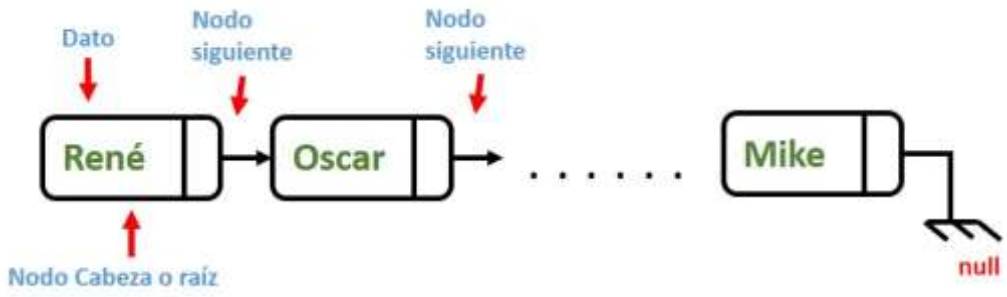

Figura 26 Estructura de una lista simplemente enlazada.

Representación:
Una lista se puede representar de dos formas:

- **Estática:** su tamaño se limita en tiempo de compilación Ej.: arreglos.
- **Dinámica:** su tamaño puede crecer indefinidamente en tiempo de ejecución Ej.: listas enlazadas simples, listas enlazadas dobles, listas circulares, pilas y colas.

Características:

- En cada elemento o nodo de la lista a excepción del primero, tiene un único predecesor se debe indicar donde se encuentra el siguiente elemento.
- Cada elemento de la lista, a excepción del último tiene un único sucesor.
- Las listas son flexibles y permiten cambios en la implementación.

Operaciones:

En una lista se pueden efectuar las siguientes operaciones:

- **Insertar**: Agregar un nuevo nodo a la lista.
- **Eliminar**: Quitar un nodo de la lista.
- **Buscar**: Permite encontrar un nodo dentro de la lista.
- **Modificar**: Actualiza la información de un determinado nodo dentro de la lista.

7.2 FUNCIONAMIENTO LÓGICO DE LA LISTA

La lista cuenta entre sus operaciones básicas el insertar en la cabecera, insertar en la cima, imprimir, eliminar, modificar, buscar elemento e insertar después de algún elemento en particular, se proporcionará las formas de programas basadas en el tipo de dato abstracto nodo y en la clase preconstruida *LinkedLists*.

7.2.1 ALGORITMO PARA INSERTAR EN LA CABECERA BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 37 Método para insertar un nodo cabecera.

```
1  public static NodoLista insertarCabecera(nodoLista lista){
2        Scanner teclado = new Scanner(System.in);
3        String valoringresado;
4        System.out.println("Ingresa el elemento a guardar: ");
5        Valoringresado = teclado.nextLine();
6        nodoLista nodoNuevo = new nodoLista(valoringresado);
7
8        if(nodoLista.buscarElemento(valoringresado,lista) == true){
9              System.out.println("Elemento repetido...");
```

```
10              return lista;
11        }
12
13        if (lista == null)//CASO 1: Cuando la lista esta vacia{
14              lista=nodoNuevo;
15        else{
16              nodoNuevo.siguiente=lista;
17              lista = nodoNuevo;
18        }
19        return lista;
20  }
```

7.2.2 ALGORITMO PARA INSERTAR EN LA CABECERA BASADO EN LA CLASE PRECONSTRUIDA LINKEDLIST

Ejemplo 38 Método para insertar la cabecera a través de LinkdList.

```
1   public static LinkedList<String>
2   insertarCabecera(LinkedList<String> lista){
3         Scanner teclado = new Scanner(System.in);
4         System.out.println("Ingrese valor a guardar: ");
5         String valor = teclado.nextLine();
6         lista.addFirst(valor);
7         return lista;
8   }
```

7.2.3 ALGORITMO PARA INSERTAR EN LA CIMA BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 39 Método para insertar en la cima de la lista.

```
1   public static nodoLista insertarCima(nodoLista lista){
2         Scanner teclado = new Scanner(System.in);
3         String valoringresado;
4         System.out.println("Ingresa el elemento a guardar: ");
5         valoringresado=teclado.nextLine();
6         nodoLista nodoNuevo = new nodoLista(valoringresado);
7
8         if (lista == null){
9               lista = nodoNuevo;
10        }else {
11              nodoLista temporal=lista;
12              nodoLista anterior=lista;
13              while(temporal != null){
14                    anterior = temporal;
15                    temporal = temporal.siguiente;
16              }
17              anterior.siguiente=nodoNuevo;
18        }
19        return lista;
20  }
```

7.2.4 ALGORITMO PARA INSERTAR EN LA CIMA BASADO EN LA CLASE PRECONSTRUIDA LINKEDLIST

Ejemplo 40 Método para insertar en la cima a través de LinkedList.

```
public static LinkedList<String> insertarCima(LinkedList<String>
lista){
      Scanner teclado = new Scanner(System.in);
      System.out.println("Ingrese valor a guardar: ");
      String valor = teclado.nextLine();
      lista.addLast(valor);
      return lista;
}
```

7.2.5 ALGORITMO PARA IMPRIMIR BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 41 Método para imprimir la lista.

```
public static void imprimir(nodoLista lista){
      nodoLista temporal = lista;
      if(lista == null)
            System.out.println("Lista vacia... imposible
                              imprimir");
      else{
            while(temporal != null){
                  System.out.println("Elemento: " +
                                    temporal.elemento);
                  temporal=temporal.siguiente;
            }
      }
}
```

7.2.6 ALGORITMO PARA IMPRIMIR BASADO EN LA CLASE PRECONSTRUIDA LINKEDLIST

Ejemplo 42 Imprimir la lista por medio de LinkedList.

```
public static void imprimir(LinkedList<String> lista){
      if(!ListaPreconstruida.estaVacia(lista))
            System.out.println("Elementos: "+lista.toString());
      else
            System.out.println("La lista esta vacia");
}
```

7.2.7 ALGORITMO PARA MODIFICAR BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 43 Modificar nodo.

```
public static nodoLista modificarLista(nodoLista lista){
      if(lista == null){
            System.out.println("Imposible modificar.... lista
                              vacia");
            return lista;
      }
```

```
7         Scanner teclado = new Scanner(System.in);
8         String valorParaEliminar;
9         String valorNuevo;
10        System.out.println("Ingresa el elemento a Modificar: ");
11        valorParaEliminar = teclado.nextLine();
12        System.out.println("Ingrese el elemento Nuevo");
13        valorNuevo=teclado.nextLine();
14
15        if(valorParaEliminar.equals(lista.elemento)){
16              lista.elemento = valorNuevo;
17              System.out.println("elemento, era el primero...");
18              return lista;
19        }
20
21        nodoLista temporal=lista;
22        nodoLista anterior=lista;
23        while(temporal != null){
24              if(valorParaEliminar.equals(temporal.elemento))
25                    break;
26              anterior = temporal;
27              temporal = temporal.siguiente;
28        }
29        if(anterior.siguiente == null){
30              System.out.println("El elemento no se encontró..");
31              return lista;
32        }
33
34        anterior.siguiente.elemento = valorNuevo;
35        return lista;
36  }
```

7.2.8 ALGORITMO PARA MODIFICAR BASADO EN LA CLASE PRECONSTRUIDA LINKEDLIST

Ejemplo 44 Modificar nodo a través de la clase LinkedList.

```
1   public static LinkedList<String>
2   modificarLista(LinkedList<String> lista){
3         if(lista.isEmpty()){
4               System.out.println("Lista vacia... imposible
5                                   modificar");
6               return lista;
7         }
8
9         Scanner teclado = new Scanner(System.in);
10        System.out.println("Ingrese valor a modificar: ");
11        String valorModificar = teclado.nextLine();
12
13        if(!ListaPreconstruida.buscarElemento(valorModificar,
14           lista)){
15              System.out.println("El elemento no esta en la
16                                  lista... ");
17              return lista;
18        }
19        System.out.println("Ingrese valor Nuevo: ");
20        String valorNuevo = teclado.nextLine();
21        lista.set(lista.indexOf(valorModificar),valorNuevo);
22        System.out.println("Modificación exitosa...");
```

```
23        return lista;
24  }
```

7.2.9 ALGORITMO PARA BUSCAR BASADO EN EL TIPO DE DATO ABSTRACTO NODO

Ejemplo 45 Método buscar nodo.

```
1   public static boolean buscarElemento(nodoLista lista){
2         if(lista == null){
3               System.out.println("Imposible buscar.... lista
4                                   vacia");
5               return false;
6         }
7         Scanner teclado = new Scanner(System.in);
8         String valorParaBuscar;
9         System.out.println("Ingresa el elemento a buscar: ");
10        valorParaBuscar=teclado.nextLine();
11        nodoLista temporal = lista;
12        while(temporal != null){
13              if(valorParaBuscar.equals(temporal.elemento))
14                    return true;
15              temporal = temporal.siguiente;
16        }
17        return false;
18  }
```

7.2.10 ALGORITMO PARA BUSCAR BASADO EN LA CLASE PRECONSTRUIDA LINKEDLIST

Ejemplo 46 Método buscar nodo a través de la clase LInkedList

```
1   public static boolean buscarElemento(String valorBuscar,
2   LinkedList<String> lista){
3         if(lista.contains(valorBuscar))
4               return true;
5         return false;
6   }
```

7.3 CUESTIONARIO DE LA UNIDAD

1. ¿Cuál es el primer elemento que se retirará en un tipo de dato abstracto lista?

- Según convenga.
- El último elemento que llegó.
- El primer elemento que llegó.
- El orden es irrelevante.

2. ¿Cuál es la dirección de memoria con la que está una lista inicialmente?

- Null.

- La de una referencia vacía.
- La de una referencia nula.
- Una dirección en formato hexadecimal.

3. **¿Qué tipo de referencia se utiliza para recorrer una estructura de datos dinámica lista?**

   - Referencia temporal.
   - Referencia original.
   - Referencia nula
   - Referencia vacía.

4. **Con respecto al algoritmo de impresión de la lista ¿Qué indica la condición temporal!=null?**

   - Que se avanza en el recorrido hasta que la lista está vacía.
   - Sólo es una comprobación ejecutada una vez
   - Es una condición que se ejecuta del lado verdadero del if.
   - Es una condición que incrementa el número de elementos de la lista.

5. **¿Cómo se imprime en una cola basada en la clase preconstruida LINKEDLIST?**

   - Utilizando el método toString.
   - Utilizando un ciclo while.
   - Utilizando el método isEmpty.
   - Utilizando una estructura condicional cualquiera.

6. **¿Por qué se define como estático el método insertarCima e insertarCola?**

   - Debido a que la referencia lista inicialmente está seteada en null.
   - Para coherencia de retorno entre instancia e invocación del método.
   - Por coherencia en el almacenamiento de la instancia y el retorno del método.
   - Por principios de la programación orientada a objetos.

7. **¿Cuál es el método que se utiliza para insertar en una lista basada en la clase preconstruida LINKEDLIST?**

   - AddFirst.
   - AddLast.
   - AddFirst y AddLast.
   - Add.

8. **¿Por qué se asocia el tipo de dato abstracto lista al método de inventario fifo y lifo?**

   - Se ingresa y retira elementos en función de los métodos LIFO y FIFO.
   - El primer elemento que se ingresa en una cola es el primero en salir.
   - El primer elemento que se ingresa en una cola es el último en salir.

- Por su algoritmo de ingreso.

9. **¿Cuál de los siguientes enunciados es más descriptivo para el tipo de dato abstracto lista?**

   - Hay desplazamiento a voluntad por los elementos de la lista.
   - Hay desplazamiento secuencial a los elementos de la lista.
   - El desplazamiento a un elemento de la lista es al azar.
   - Usa métodos de encolar y desencolar comunes con QUEUE.

10. **¿Qué hace la sentencia "lista.set(lista.indexOf(valorModificar),valorNuevo);" en el bloque de modificación de listas basado en la clase preconstruida LINKEDLIST?**

    - A través del método indexOf retorna la posición del elemento que se busca y con el método set actualiza al nuevo valor.
    - A través del método indexOf actualiza al nuevo valor y con el método set retorna la posición del elemento que se busca.
    - Elimina el elemento que se busca.
    - Actualiza usando el método indexOf.

BIBLIOGRAFÍA

Arnow, D., & Weiss, G. (2001). *Introducción a la programación con Java*. Pearson Publications Company.

Barnes, D. J., & Kölling, M. (2007). *Programación orientada a objetos con Java*. Manejo eficiente del tiempo.

Boyarsky, J., y Selikoff, S. (2014). *OCA: Oracle Certified Associate Java SE 8 Programador I Study Guide,* Examen 1Z0-808.

John Wiley & Sons. Dean, J. S., & Dean, R. H. (2000). *Introducción a la programación con Java*. McGraw-Hill Interamericana.

Deitel, H. M., & Deitel, P. J. (2004). *Cómo programar en Java. Pearson Educación*.

Durán, F., Gutiérrez, F., & Pimentel, E. (2007). *Programación orientada a objetos con Java. Editorial Paraninfo.*

Froufe Quintas, A., & Cárdenes, J. (2004). *J2ME, Java 2 Micro Edition: manual de usuario y tutorial*.

Guaman, R., Guaman, J., Erreyes, D. & Torres, H. (2017). *Programación en Java. Orientada a objetos a través de ejemplos*.

Heileman, Gregory L (1998). *Estructura de Datos, Algoritmo y Programación Orientada a Objetos.*

Joyanes Aguilar, L., & Zahonero Martínez, (2002). *Programación en Java 2*. Madrid: McGraw-Hill.

Joyanes Aguilar, L. (2008). *Fundamentos de Programación: Algoritmos, Estructuras de Datos y Objetos.*

Osorio, F. L. (2007). *Introducción a la Programación en Java*. ITM.

Sánchez, J. (2004). *Java2-Incluye Swing, Threads, programación en red, JavaBeans, JDBC y JSP/Servlets*. Recuperado de www.jorgesanchez.net

Sznajdleder, P. (S.a) *Programación Orientada a Objetos y Estructura de Datos a Fondo Implementación de Algoritmos en Java.*

Torres, S. A. C., Valbuena, S. J., & Ramirez, M. L. V. (2008). *Introducción a la Programación en Java*. ELIZCOM SAS.

Wu, C. T., & Wu, C. T. (2001). *Introducción a la programación orientada a objetos con Java*.

Wanumen, S., García Laura., & Mosquera Darín (2017). *Estructuras de datos en Java.*

SOBRE LOS AUTORES

ROBERT WILFRIDO MOREIRA CENTENO

Nació en Ecuador en 1983, recibió el título de Ingeniero en Sistemas Informáticos en la Universidad Técnica de Manabí en el 2009. A nivel de postgrado tiene el título de Magíster en Sistemas de Información Gerencial en la Escuela Superior Politécnica del Litoral en el año 2013. Se ha desempañado como docente en la Universidad Técnica de Manabí, Universidad Estatal del Sur de Manabí y Universidad Laica Eloy Alfaro de Manabí en donde actualmente trabaja, impartiendo las cátedras de Programación Orientada a Objetos, Estructuras de Datos, Análisis, Diseño y Administración de Bases, de Datos.

EDWIN RENÉ GUAMÁN QUINCHE

Nació en Ecuador en 1982, es Ingeniero en Sistemas, a nivel de postgrado tiene un Máster en Sistemas Informáticos Avanzados en la Universidad del País Vasco - España. Desde el año 2017 se desempeña como docente de la Facultad de Ciencias Informáticas en la Universidad Laica Eloy Alfaro de Manabí – Ecuador, tutorando asignaturas relacionadas a la Programación Orientada a Objetos, Web y Móvil. Ha investigado sobre tecnologías web, sistemas distribuidos y lenguajes de programación, habiendo publicado varios artículos y realizado ponencias al respecto.

ÓSCAR ARMANDO GONZÁLEZ LÓPEZ

Nació en Ecuador en 1974. Docente de la Facultad de Ciencias Informáticas desde el 2017, Analista en Sistemas por la Universidad Laica Eloy Alfaro de Manabí (Ecuador) con estudios de cuarto nivel realizados en la Universidad Federico Santa María de Chile con el título Magíster en Informática de Gestión y Nuevas Tecnologías. Con experiencia en el sector privado en Asesoría en sistemas informáticos, y manejo técnico de testeo de Software.

MIKE PAOLO MACHUCA ÁVALOS

Nació en Ecuador en 1981. Docente de la Facultad de Ciencias Informáticas desde el 2017, Ingeniero Eléctrico por la Universidad Laica Eloy Alfaro de Manabí (Ecuador) con estudios de postgrado realizados en la Universidad del Mar de Chile con el título Magíster en Finanzas y Comercio Internacional. Con amplia experiencia en el sector privado en la Construcción de redes de telecomunicaciones, diseño y construcción de líneas y redes eléctricas e Implementación de sistemas automatizados. Su interés en investigación se relaciona con el Internet de las Cosas (IoT) y Domótica.

La programación es una técnica muy interesante y dentro de las existentes la que cuenta con una vertiginosa evolución, aunque su desarrollo se enmarcó inicialmente en un begin y end, ahora el alcance de los trabajos que se realizan en ella se expande debido a las características modulares en los lenguajes y herramientas actuales.

Los autores de esta obra han vivido la experiencia educativa y por ende conocen algunas de las dificultades que surgen cuando se entra en el campo de la tecnología, este libro constituye un buen inicio para tan interesante disciplina como lo es la programación, puesto que se abarca desde el uso básico de las variables hasta las operaciones comunes numéricas y ahonda en las estructuras de datos.

Uleam
UNIVERSIDAD LAICA
ELOY ALFARO DE MANABÍ

ISBN: 978-9942-775-27-6
9789942775276