Tipos de datos en Java (primitivos y objetos)

1. Tipos de datos en Java (primitivos y objetos)

En Java, los tipos de datos se dividen en primitivos (como *int, float, boolean*) y de referencia (objetos). Los tipos primitivos representan valores simples, mientras que los de referencia corresponden a objetos complejos que almacenan referencias a los datos. Los tipos primitivos son *boolean, byte, short, int, long, float, double* y *char*.

Tipos de datos primitivos

Son los tipos de datos más básicos de Java y almacenan valores simples directamente.

- *Boolean*: almacena valores lógicos true (verdadero) o false (falso).
- *Byte*: entero de 8 bits; permite almacenar números en el rango de -128 a 127.
- *Short*: entero de 16 bits.

- *Int*: entero de 32 bits, adecuado para la mayoría de los números enteros.
- *Long*: entero de 64 bits, para valores enteros de gran tamaño.
- *Float*: número de punto flotante de 32 bits.
- *Double*: número de punto flotante de 64 bits, con mayor precisión.
- *Char*: representa un carácter Unicode individual, entre comillas simples (por ejemplo: 'A').

Como se almacenan directamente en la pila, su tamaño respectivo puede tener una importancia decisiva. Los tipos de datos primitivos también son relevantes porque Java es un lenguaje tipado estáticamente. Esto significa que, al crear un programa, el tipo de datos de una variable debe estar definido previamente. Solo entonces puede ejecutarse el código sin generar mensajes de error.

Tabla 1. Resumen de los tipos de datos primitivos en Java

Tipo de dato	Tamaño	Rango de valores	Valor predeterminado	Clase envoltorio
boolean	1 bit	*true* o *false*	false	java.lang.Boolean
byte	8 bits	-128 a 127	0	java.lang.Byte
short	16 bits	-32768 a 32767	0	java.lang.Short
int	32 bits	-2147483648 a 2147483647	0	java.lang.Integer
long	64 bits	-9223372036854775808 a 9223372036854775807	0	java.lang.Long
float	32 bits	hasta siete decimales	0,0	java.lang.Float
double	64 bits	hasta 16 decimales	0,0	java.lang.Double
char	16 bits	**'\u0000' (también 0) a '\uffff' (equivale a 65535)	'\u0000'	java.lang.Character

Fuente: elaboración propia.

Tipos de datos de referencia (objetos)

Estos tipos almacenan referencias a los datos reales y son más complejos que los primitivos.

- **Clases y *strings*:** la clase *String*, utilizada para representar texto, es un ejemplo común. También se incluyen todas las demás clases definidas por el programador y las que forman parte de la biblioteca de Java.

- **Matrices:** arreglos de datos de cualquier tipo (primitivo u objeto).

- **Interfaces:** definen un contrato que deben cumplir las clases que las implementan.

- ***Null*:** representa la ausencia de un objeto.

Diferencias clave

En la siguiente tabla presentamos las principales diferencias entre datos primitivos y de referencia.

Tabla 2. Comparación entre tipos primitivos y de referencia en Java

Característica	Tipos primitivos	Tipos de referencia (Objetos)
Almacenamiento	Los valores se almacenan	Las referencias a los objetos se almacenan en la pila, mientras que los objetos

	directamente en la memoria de pila.	mismos se encuentran en el montón (heap).
Métodos	No tienen métodos asociados.	Tienen métodos, lo que permite realizar operaciones complejas sobre ellos.
Por defecto	Tienen valores predeterminados (por ejemplo, 0 para int).	Su valor predeterminado es null

Fuente: elaboración propia.

Boolean

El boolean de Java también se conoce como valor de verdad. Es el más simple de todos los tipos primitivos, ya que solo admite dos valores posibles: true o false. Se utiliza cuando se requiere un operando lógico y, por tanto, pertenece al grupo de los tipos de datos lógicos. En las expresiones, sus dos posibles valores suelen representar una condición que se cumple (*true*) o no (*false*). Si no se asigna un valor a un boolean, su valor por defecto es *false*.

```
public class Main {

public static void main(String[] args) {

boolean x = true;

System.out.println(x);
```

```
}

}
```

Una salida con el comando de Java «System.out.println» se muestra de la siguiente manera:

true

Byte

El byte es la opción más pequeña dentro de la categoría de tipos de datos enteros. Tiene un rango de valores limitado, que va de -128 a 127. Sin embargo, solo requiere 8 bits de memoria. Su nombre proviene del hecho de que 8 bits equivalen a un byte. Si solo se necesitan valores dentro de ese rango reducido, se puede declarar un byte de la siguiente manera:

```
public class Main {

public static void main(String[] args) {

byte x = 101;

System.out.println(x);

}
```

```
}
```

El resultado sería: 101

Short

El short ocupa el doble de espacio que el byte, pero es uno de los tipos primitivos menos utilizados en Java. No obstante, si el byte resulta insuficiente y el int demasiado grande, puede emplearse este tipo de dato entero. Se declara de la siguiente manera:

```
public class Main {

public static void main(String[] args) {

short x = -27412;

System.out.println(x);

}

}
```

El resultado es el siguiente: -27412

Float

Para representar subconjuntos de números racionales, Java ofrece dos tipos primitivos de coma flotante. El float es el menor de ellos y requiere 32 bits. Puede mostrar hasta siete cifras decimales. Sin embargo, no es muy preciso y, por tanto, se utiliza con poca frecuencia. Si aun así se necesita, se declara del siguiente modo:

```
public class Main {

public static void main(String[] args) {

float x = 7.22f;

System.out.println(x);

}
```

Se debe insertar una «f» (minúscula o mayúscula) después del número para indicar al compilador que se trata de un float y no de un double. La letra no se muestra en la salida: 7.22.

Double

El segundo tipo de datos de coma flotante en Java es double. Aunque ofrece una precisión mucho mayor que float, no garantiza resultados completamente exactos. Para obtener una

precisión más rigurosa, puede utilizarse la clase BigDecimal. Si double resulta suficiente, a continuación se presenta un ejemplo de uso:

```
public class Main {

public static void main(String[] args) {

double x = 7.2252;

System.out.println(x);

}
```

En este caso, no es necesario añadir ningún sufijo. El resultado es el siguiente: 7.2252.

Char

El tipo char se utiliza para representar un carácter en Unicode, en un rango que va desde '\u0000' hasta '\uffff', es decir, desde 0 hasta 65 535. De este modo, el tipo de dato char puede representar casi todos los caracteres europeos y asiáticos. Cada uno de estos caracteres ocupa 16 bits de memoria. Los valores de este tipo primitivo se encierran entre comillas simples. Así se verá el código:

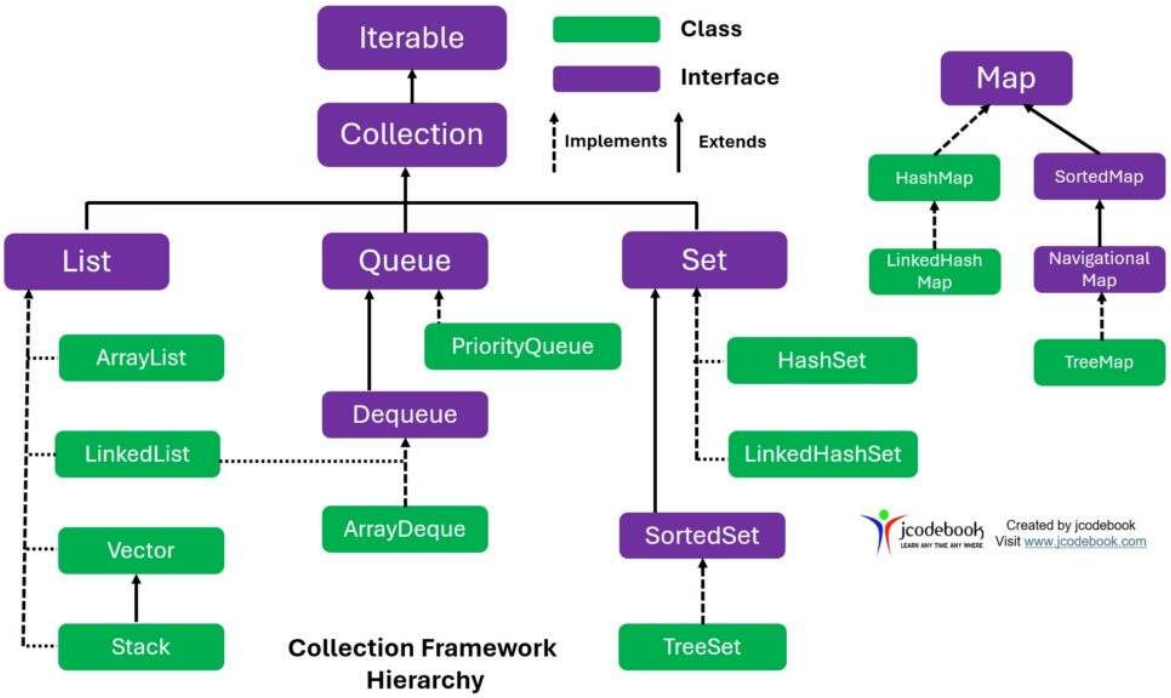
```
public class Main {

public static void main(String[] args) {

double x = 7.2252;

System.out.println(x);

}
```

Se obtendrá este resultado: &.

Colecciones y su jerarquía (*List*, *Set*, *Map*)

La jerarquía de colecciones en Java se basa en dos interfaces principales: *Collection* y *Map*. *Collection* es la interfaz raíz para colecciones de elementos individuales; de ella derivan *List* (secuencia ordenada que permite elementos duplicados) y Set (colección de elementos únicos, sin duplicados). Por otro lado, *Map* es una colección de pares clave-valor y no hereda directamente de *Collection*.

Figura 1. Jerarquía del *framework* de colecciones en Java



Fuente: [imagen sin título sobre jerarquía del framework de colecciones en Java], (s.f.), https://short.do/jHdEdJ

Jerarquía de colecciones

A continuación, se describen las principales interfaces y clases que forman parte de esta jerarquía:

Collection —

Es la interfaz raíz de la que heredan List y Set. Define métodos comunes para las colecciones, como «add()», «remove()», «size()», «isEmpty()» y «contains()». A su vez, hereda de la interfaz Iterable, lo que permite iterar sobre sus elementos.

List —

Representa una secuencia ordenada de elementos. Permite elementos duplicados y el acceso a cada uno se realiza a través de su posición (índice).

Ejemplos de implementación: *ArrayList* (basada en arreglos) y *LinkedList* (lista doblemente enlazada).

Set —

Define un grupo de elementos únicos. No admite duplicados y, por lo general, no garantiza un orden específico.

Ejemplos de implementación: HashSet y TreeSet.

Map —

Representa una colección de pares clave-valor. Las claves deben ser únicas, aunque los valores pueden repetirse. Esta interfaz no extiende de *Collection*. Ejemplos de implementación: *HashMap* y *TreeMap*.

Tabla 3. Comparación entre List, Set y Map

Característica	*List*	*Set*	*Map*
Permite duplicados	Sí	No	No (claves únicas), sí (valores)
Ordenación	Ordenada (por inserción)	No garantizada	No garantizada (por claves en *HashMap*) u ordenada por clave en *TreeMap*
Acceso	Por índice	No por índice, sino por elemento	Por clave
Relación jerárquica	Hereda de *Collection*	Hereda de *Collection*	No extiende de *collection*

Fuente: elaboración propia.

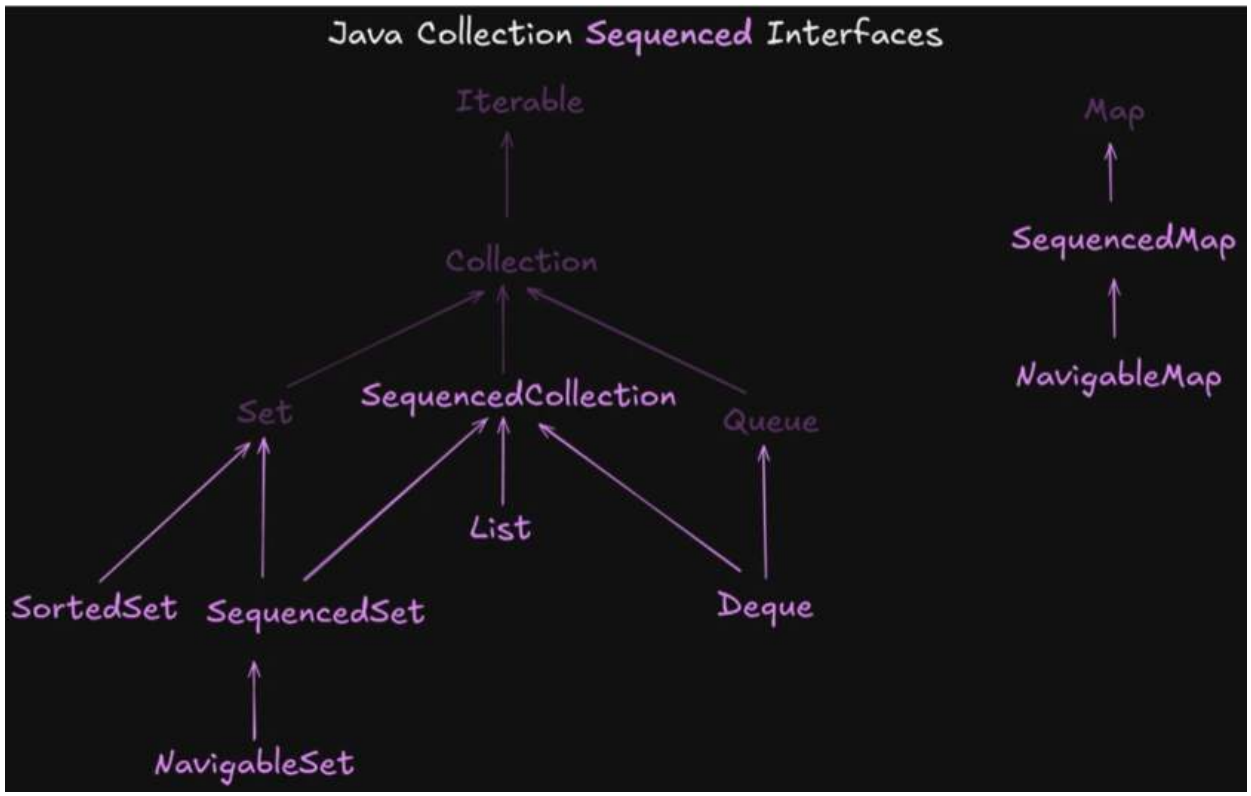
Java Collections List vs. Set (I)

En este punto se introducen las listas (*List*) y los conjuntos (Set). Para ello, primero se analizará en qué lugar se ubican dentro de la jerarquía de clases del *framework* de colecciones.

A partir de la versión 21 de Java, se incorporaron algunos cambios, como la interfaz *SequencedCollection*, que pertenece al paquete *java.util* (ver figura 2). Esta interfaz proporciona una forma de

manejar colecciones que mantienen un orden definido de los elementos.

Figura 2. Modificaciones introducidas en Java 21



Fuente: elaboración propia

A diferencia de una colección estándar, una *sequenced collection* tiene un orden secuencial definido, lo que permite realizar operaciones que dependen de la posición de los elementos dentro de la colección. En la

siguiente imagen se muestra una versión simplificada de la jerarquía visual a partir de Iterable.

Figura 3. Versión de jerarquía simplificada de Iterable

Fuente: elaboración propia.

List<E>

Interfaz para colecciones ordenadas que permiten elementos duplicados.

Figura 4. Interfaz List<E>

```java
public interface List<E> extends Collection<E> {
    // Métodos de acceso posicional
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Operaciones de lista
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int fromIndex, int toIndex);

    // Métodos static (Java 9+)
    static <E> List<E> of() { ... }
    static <E> List<E> of(E e1) { ... }
    static <E> List<E> of(E... elements) { ... }

    // Métodos default (Java 8+)
    default void replaceAll(UnaryOperator<E> operator) {
        Objects.requireNonNull(operator);
        final ListIterator<E> li = this.listIterator();
        while (li.hasNext()) {
            li.set(operator.apply(li.next()));
        }
    }

    default void sort(Comparator<? super E> c) {
        Collections.sort(this, c);
    }
}
```

Fuente: elaboración propia.

Set<E>

Interfaz para colecciones que no permiten elementos duplicados.

Figura 5. Interfaz Set<E>

```
public interface Set<E> extends Collection<E> {
    // Hereda todos los métodos de Collection

    // Métodos static (Java 9+)
    static <E> Set<E> of() { ... }
    static <E> Set<E> of(E e1) { ... }
    static <E> Set<E> of(E... elements) { ... }

    // Métodos default (Java 8+)
    default Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, Spliterator.DISTINCT);
    }
}
```

Fuente: elaboración propia.

Así se puede continuar con la definición de otras interfaces dentro del *framework* de colecciones:

SortedSet —

Es un Set ordenado según el orden natural de sus elementos o

mediante un *Comparator*.

NavigableSet —

Extiende *SortedSet* e incorpora métodos de navegación, como acceso a elementos mayores, menores o límites.

Queue —

Interfaz para colecciones que funcionan como una cola, siguiendo el orden FIFO (primero en entrar, primero en salir).

Deque. —

Representa una cola de doble extremo, que permite insertar y eliminar elementos por ambos lados.

A continuación, se presentan algunas de las interfaces más relevantes del *framework* de colecciones en Java, junto con los métodos que definen o heredan:

Iterable<T>

public interface Iterable<T>

Define un método fundamental para iterar sobre una colección:

default void forEach(Consumer<? super T> action) - Permite aplicar una acción a cada elemento de la colección.

Collection<E>

public interface Collection<E> extends Iterable<E>

Amplía Iterable e incorpora métodos comunes para manipular colecciones:

- int size()
- boolean isEmpty()
- boolean contains(Object o)
- Object[] toArray()
- boolean add(E e) – Adiciona un elemento.
- boolean remove(Object o) – Elimina un elemento.
- boolean containsAll(Collection<?> c) – Verifica si contiene todos los elementos de otra colección (bulk operation).

- boolean addAll(Collection<? extends E> c)
- boolean removeAll(Collection<?> c)

SequencedCollection<E>

public interface SequencedCollection<E> extends Collection<E>

Proporciona una manera de manejar colecciones que mantienen un orden definido de los elementos. El orden secuencial permite operaciones basadas en la posición:

- SequencedCollection<E> reversed()
- default void addFirst(E e)
- default void addLast(E e)
- default E getFirst() – Retorna el primer elemento.
- default E getLast() – Retorna el último elemento.
- default E removeFirst()
- default E removeLast()

List<E>

public interface List<E> extends SequencedCollection<E>

Permite colecciones ordenadas con acceso posicional. Algunos métodos destacados son los siguientes:

- int size()
- boolean isEmpty()
- boolean add(E e)
- boolean remove(Object o)
- boolean containsAll(Collection<?> c)
- E get(int index) – Accede a un elemento por índice.
- E set(int index, E element) – Actualiza un elemento en una posición específica.
- void add(int index, E element) – Inserta un elemento en una posición.
- E remove(int index) – Elimina un elemento en una posición.

- int indexOf(Object o) – Busca la posición de un elemento.
- default void addFirst(E e) { this.add(0, e); }
- default void addLast(E e) { this.add(e); }

Map<K, V>

public interface Map<K, V>

No forma parte de la jerarquía de Collection, pero es clave para almacenar pares clave-valor:

- int size()
- boolean isEmpty()
- V get(Object key) – Obtiene el valor asociado a una clave.
- V put(K key, V value) – Asocia un valor a una clave.
- V remove(Object key) – Elimina el valor asociado a una clave.

A partir de aquí, se podrían enumerar más interfaces y métodos, pero esta descripción servirá como base general para los ejemplos que se presentarán más adelante, donde se utilizarán algunos de los métodos mencionados.

List

A continuación, se muestra la secuencia de las siguientes líneas de código:

```
ArrayList<String> lista = new ArrayList<>();

lista.add("Walter");

lista.add("Fabian");

lista.add(2, null);

lista.add(3, "Aguero");

lista.add(2, "de");

lista.set(2, "San Luis");

lista.remove("San Luis");
```

```
lista.remove(2);
```

Paso 1

Crear la lista

Index	Value
int	*<String>*

Tabla 4. Estado inicial de la lista (vacía)

List<String> lista = new ArrayList<>();

Paso 2

Index	Value
0	Walter

Tabla 5. Lista luego de agregar «Walter»

lista.add("Walter");

Se agrega el primer elemento al final de la lista.

Paso 3

Index	Value
0	Walter
1	Fabián

Tabla 6. Lista luego de agregar «Fabián»

lista.add("Fabian");

Se agrega otro elemento al final de la lista.

Paso 4

Index	Value
0	Walter
1	Fabián
2	*null*

Tabla 7. Lista luego de insertar null en la posición 2

lista.add(2, null);

Se inserta un valor null en la posición 2 (al final). Los elementos existentes no se modifican.

Paso 5

Index	Value
0	Walter
1	Fabián
2	*null*
3	Agüero

Tabla 8. Lista luego de insertar «Agüero» en la posición 3

lista.add(3, "Aguero");

Se agrega el valor «Agüero» en la posición 3. El null se desplaza una posición.

Paso 6

Index	Value
0	Walter
1	Fabián
2	de
3	*null*
4	Agüero

Tabla 9. Lista luego de insertar «de» en la posición 2

lista.add(2, "de");

Se inserta la palabra «de» en la posición 2. Todos los elementos desde esa posición se desplazan una posición hacia la derecha.

Paso 7

Index	Value
0	Walter
1	Fabián
2	San Luis
3	*null*
4	Agüero

Tabla 10. Lista luego de reemplazar «de» por «San Luis»

lista.set(2, "San Luis");

Se reemplaza el valor en la posición 2 («de») por «San Luis».

Paso 8

Index	Value
0	Walter
1	Fabián
2	*null*
3	Agüero

Tabla 11. Lista luego de eliminar «San Luis»

lista.remove("San Luis");

Se elimina el valor «San Luis» de la lista. Los elementos posteriores se reordenan para ocupar su lugar.

Paso 9

Step 9 Title

Index	Value
0	Walter
1	Fabián
2	Agüero

Tabla 12. Lista luego de eliminar el elemento en la posición 2

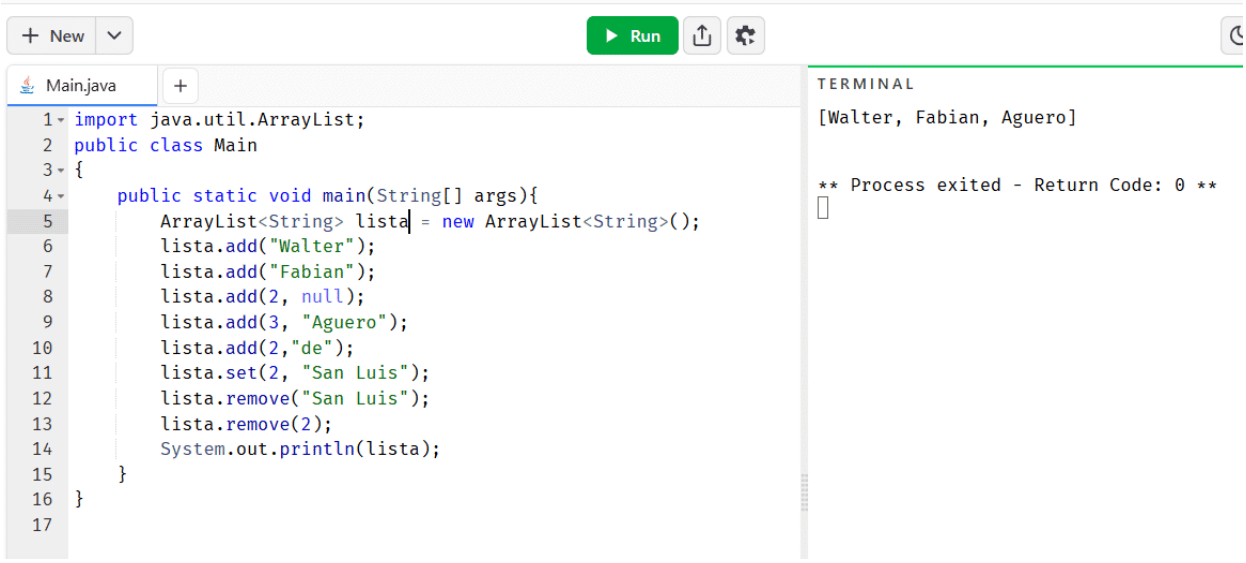
lista.remove(2);

Se elimina el valor en la posición 2 (null).

Se adjunta el código fuente del ejemplo trabajado, disponible en el siguiente enlace, y correspondiente a la figura 6:

https://www.online-java.com/bvkEgYlbKU

Figura 6. Ejemplo de código fuente



Fuente: elaboración propia.

Se puede ampliar el ejemplo incorporando las siguientes sentencias:

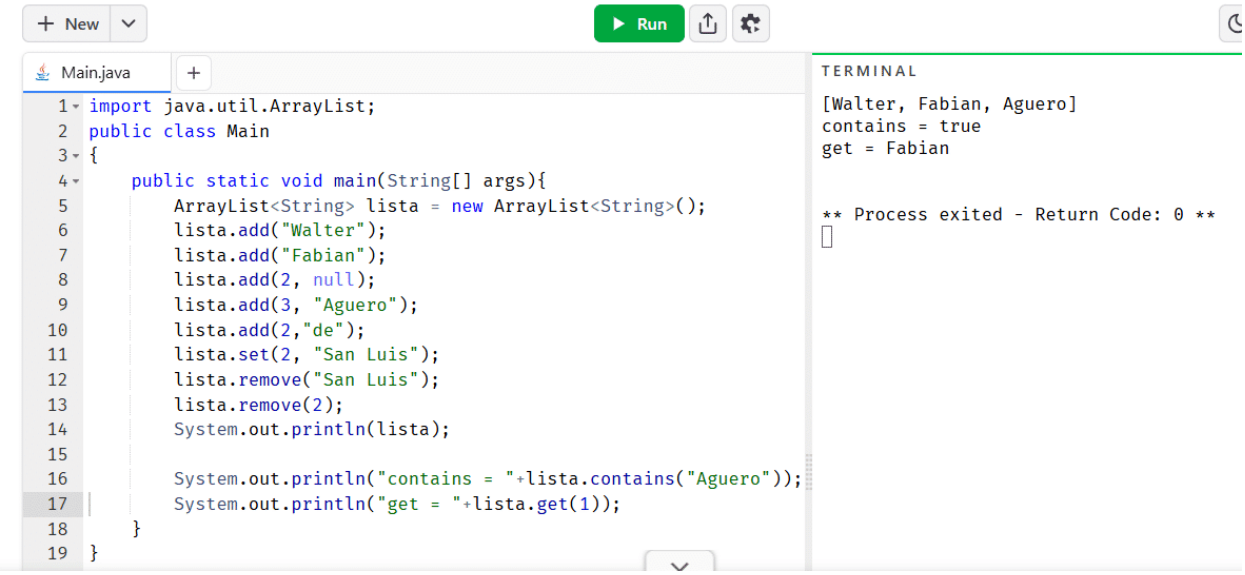
- La primera instrucción devuelve true (verdadero) si el elemento buscado se encuentra en la lista, o false (falso) en caso contrario.
- La segunda muestra el contenido almacenado en la posición con índice 1.

System.out.println("contains = " + lista.contains("Aguero"));

System.out.println("get = " + lista.get(1));

Se invita al alumno a agregar estas dos líneas al final del código y ejecutar el programa para observar el resultado.

Figura 7. Código fuente y salida del programa que utiliza métodos de la clase ArrayList

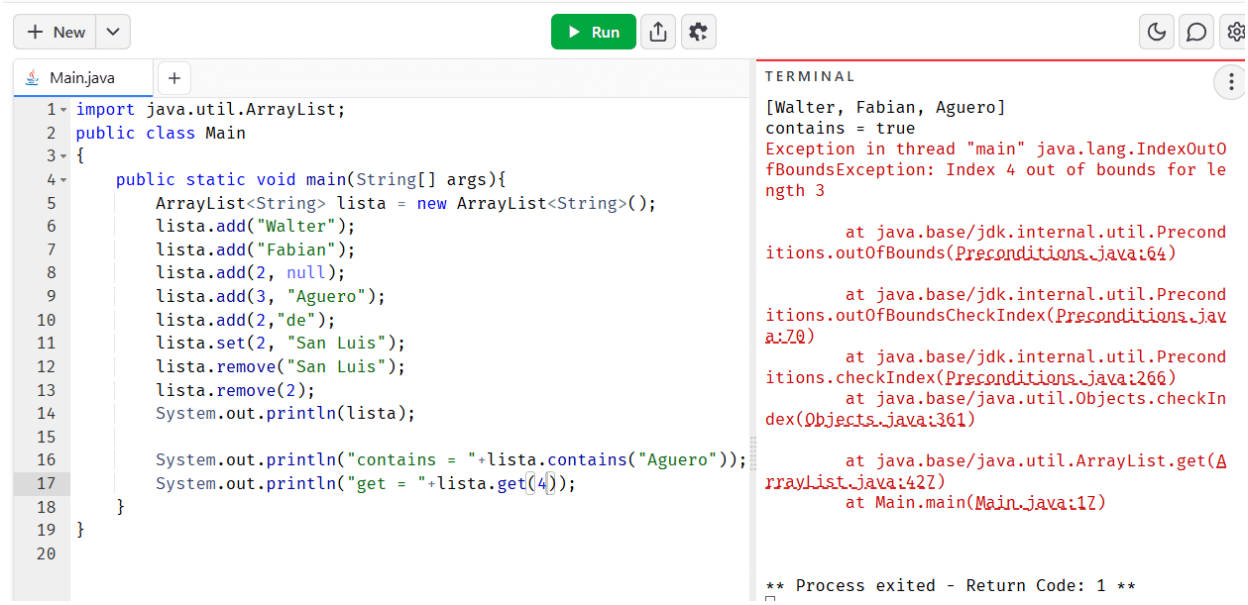


Fuente: elaboración propia.

Como se observa en el paso 9, la lista quedó con tres elementos, enumerados en las posiciones del índice 0 al 2. Si se agrega la siguiente sentencia, que intenta acceder a la posición 4 —la cual no existe—, se producirá un error en tiempo de ejecución:

- System.out.println("get = " + lista.get(4));

Figura 8. Error por intento de acceso a un índice inexistente en una *ArrayList*



Fuente: elaboración propia.

Para evitar este tipo de errores, se recomienda verificar el tamaño de la lista antes de acceder a una posición específica. Esto puede hacerse con el método size(), como se muestra en la siguiente instrucción:

System.out.println("Taminio = "+lista.size());

Esta línea imprimirá la cantidad de elementos que contiene la lista y permitirá confirmar que el índice 4 no es válido en este caso.

Set

Para entender mejor los conceptos asociados a la interfaz Set, se presenta una secuencia de pasos que permite observar el comportamiento de un HashSet ante la inserción y eliminación de elementos. El siguiente código es utilizado como base para el análisis:

```
HashSet<String> pila = new HashSet< String >();

pila.add(“Walter”);

pila.add(“Fabian”);

pila.add(“Fabian”);

pila.add(3, “Aguero”);

pila.remove(“Walter”);

pila.remove(“Walter”);
```

Paso 1

Crear la estructura

Value (*String*)
(vacío)

Tabla 13. Estado inicial del *HashSet*

Set<String> pila = new HashSet<>();

Paso 2

Value (*String*)
Walter

Tabla 14. Lista luego de agregar «Walter»

pila.add("Walter");

Paso 3

Value (*String*)
Walter
Fabián

Tabla 15. Lista luego de agregar «Fabián»

pila.add("Fabian");

Paso 4

Value (*String*)
Walter
Fabián

Tabla 16. El contenido no se modifica

pila.add("Fabian");

No lo agrega porque no admite repetidos e ignora la sentencia.

Paso 5

Value (*String*)
Walter
Fabián
Agüero

Tabla 17. Lista luego de agregar «Agüero»

pila.add("Aguero");

Paso 6

Value (*String*)
Fabián
Agüero

Tabla 18. Lista luego de eliminar «Walter»

pila.remove("Walter");

Paso 7

Value (*String*)
Fabián
Agüero

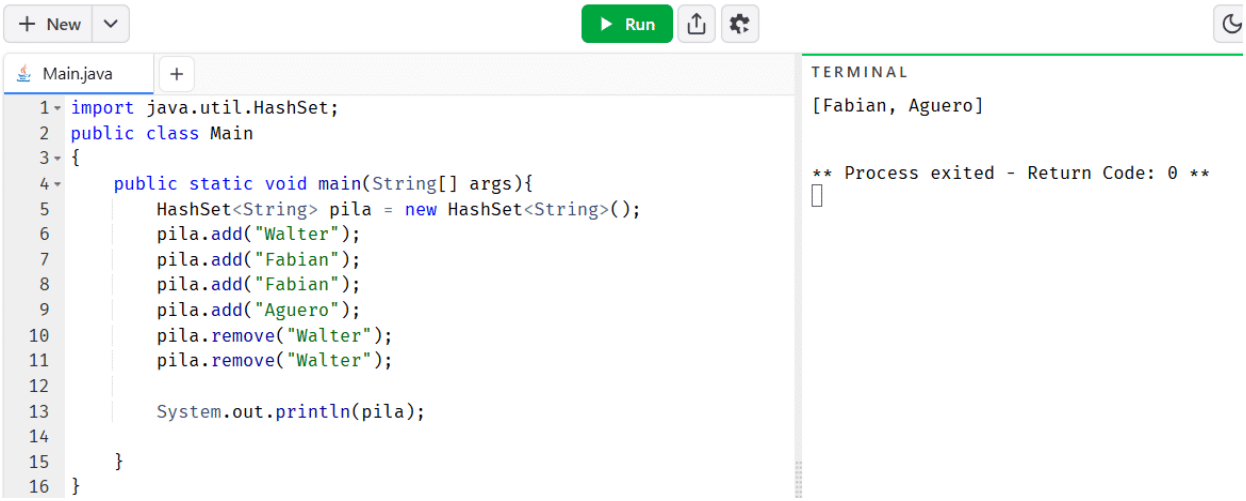
Tabla 19. El contenido no se modifica

pila.remove("Walter");

Ignora la sentencia, ya que el elemento no se encuentra.

El alumno puede ejecutar el código haciendo clic en el siguiente enlace, tal como se muestra en la figura 9: https://www.online-java.com/pmwVU31UW7

Figura 9. Ejemplo de código fuente



Fuente: elaboración propia.

A continuación, se puede hacer uso de la siguiente sentencia:

System.out.println("contains = "+pila.contains("Aguero"));

Esta instrucción arrojará como resultado: true. Se invita al alumno a agregarla y ejecutar el código, tal como se muestra en la figura 10.

Figura 10. Verificación de contenido en un HashSet con el método contains()


```java
import java.util.HashSet;
public class Main
{
    public static void main(String[] args){
        HashSet<String> pila = new HashSet<String>();
        pila.add("Walter");
        pila.add("Fabian");
        pila.add("Fabian");
        pila.add("Aguero");
        pila.remove("Walter");
        pila.remove("Walter");

        System.out.println(pila);
        System.out.println("contains = "+pila.contains("Aguero"));

    }
}
```

Fuente: elaboración propia.

Map

En Java, Map es una interfaz que almacena datos en pares clave-valor, donde cada clave es En Java, Map es una interfaz que almacena datos en pares clave-valor, donde cada clave es única. Esta estructura permite un acceso rápido al valor asociado a una clave. A diferencia de las listas, que permiten duplicados, y de los conjuntos (Set), que no aceptan claves ni valores duplicados, un Map permite claves únicas, pero valores que pueden repetirse.

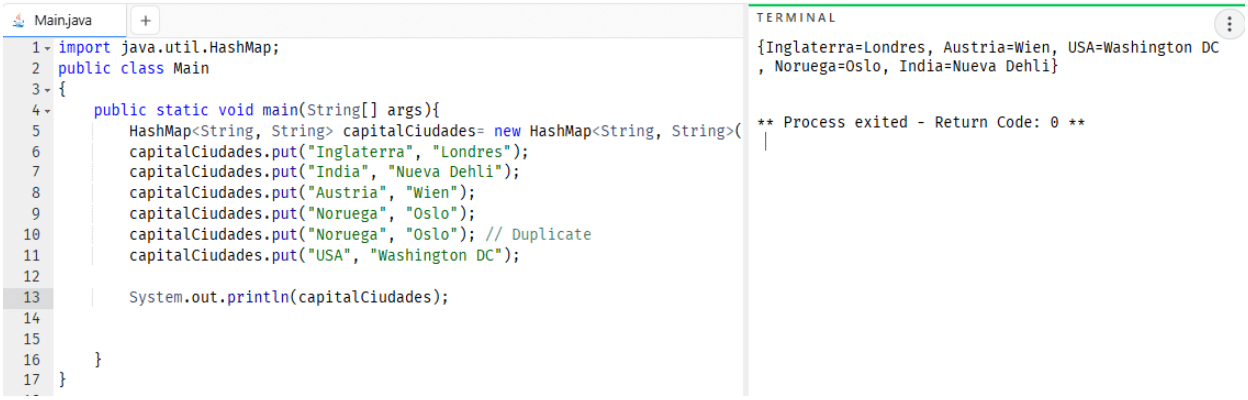
Map puede implementarse de diversas formas, como HashMap, TreeMap o LinkedHashMap, cada una con sus propias características en cuanto a orden y eficiencia.

Como en los casos anteriores, se seguirá el mismo enfoque mediante la explicación del siguiente código:

```
HashMap<String, String> capitalCiudades= new HashMap<String, String>();

  capitalCiudades.put("Inglaterra", "Londres");

  capitalCiudades.put("India", "Nueva Dehli");

  capitalCiudades.put("Austria", "Wien");

  capitalCiudades.put("Noruega", "Oslo");

  capitalCiudades.put("Noruega", "Oslo"); // Duplicate

  capitalCiudades.put("USA", "Washington DC");
```

Este código puede ejecutarse en el siguiente enlace: https://www.online-java.com/751R6BV8kV

Figura 11. Ejemplo de uso de *HashMap* con claves y valores en Java



Fuente: elaboración propia.

Se sugiere al alumno que agregue y ejecute la siguiente línea de código:

System.out.println(capitalCiudades.get("India"));

Ejemplos

Map básico con *HashMap*

Podemos copiar y pegar el siguiente código en un IDE en línea o hacer clic en el enlace para ejecutarlo directamente: https://www.online-java.com/2yEWskNV5b

Tabla 20. Ejemplo de uso de un Map para almacenar y recorrer pares clave-valor

```java
import java.util.*;
public class Main
{
    public static void main(String[] args){
        // Crear un Map (clave -> valor)
        Map<String, Integer> edades = new HashMap<>();

        // Agregar elementos
        edades.put("Ana", 25);
        edades.put ("Carlos", 30);
        edades.put ("Maria", 28);

        // Obtener un valor
        System.out.println("Edad de Ana: " + edades.get("Ana"));

        // Recorrer el Map
        for (Map.Entry<String, Integer> entrada : edades.entrySet()) {

System.out.println(entrada.getKey() + " -> " + entrada.getValue());
        }
    }
}
```

```
TERMINAL

Edad de Ana: 25
Ana -> 25
Maria -> 28
Carlos -> 30


** Process exited - Return Code: 0 **
```

Fuente: elaboración propia.

Contador de palabras

Para acceer al código fuente, hacer clic en el siguiente enlace: https://www.online-java.com/0s1N3KoWI8

Tabla 21. Ejemplo de contador de palabras utilizando un Map en Java

```java
import java.util.*;
public class Main
{
    public static void main(String[] args){
        String texto = "El mundo de Java de programacion";
        String[] palabras = texto.split(" ");
        Map<String, Integer> contador = new HashMap<>();
        for (String palabra : palabras) {
            // Si la palabra ya existe, incrementa el contador
            // Si no existe, la agrega con valor 1
            contador.put(palabra, contador.getOrDefault(palabra, 0) + 1);
        }
        System.out.println("Conteo de palabras:");
        contador.forEach((palabra, cuenta) ->
         System.out.println(palabra + ": " + cuenta));
    }
}
```

```
TERMINAL
Conteo de palabras:
de: 2
Java: 1
El: 1
mundo: 1
programacion: 1

** Process exited - Return Code: 0 **
```

Fuente: elaboración propia.

Diccionario simple

Podemos acceder al código fuente haciendo clic aquí: https://www.online-java.com/ahhGyS5bUb

Tabla 22. Diccionario español–inglés implementado con un Map en Java

```
import java.util.*;
public class Main
{
    public static void main(String[] args){
        Map<String, String> diccionario = new HashMap<>();

        // Agregar traducciones espanol -> ingles
        diccionario.put("casa", "house");
        diccionario.put("perro", "dog");
        diccionario.put("gato", "cat");
        diccionario.put("libro", "book");

        // Buscar traduccion
        Scanner scanner = new Scanner(System.in);
        System.out.print("Ingresa una palabra en espanol: ");
        String palabra = scanner.nextLine();

        if (diccionario.containsKey(palabra)) {

System.out.println("Traduccion: " + diccionario.get(palabra));
        } else {
            System.out.println("Palabra no encontrada en el diccionario");
        }

        scanner.close() }
}
```

```
TERMINAL
Ingresa una palabra en espanol: perro
Traduccion: dog

** Process exited - Return Code: 0 **
```

Fuente: elaboración propia.

Agenda de contactos

Se puede usar el siguiente link para acceder al código fuente del programa que se muestra a continuación: https://www.online-java.com/macPVPtus6

Tabla 23. Agenda de contactos implementada con un TreeMap en Java (ordenado por clave)

```java
import java.util.*;
public class Main
{
    public static void main(String[] args){
        Map<String, String> agenda = new TreeMap<>();
       // TreeMap ordena por clave

        agenda.put("Juan", "123-4567");
        agenda.put("Ana", "987-6543");
        agenda.put("Carlos", "555-1234");
        agenda.put("Maria", "444-5678");

        System.out.println("Agenda de contactos:");
        agenda.forEach((nombre, telefono) ->
            System.out.println(nombre + ": " + telefono));

        // Verificar si existe un contacto
        String buscar = "Ana";
        if (agenda.containsKey(buscar)) {

System.out.println("\nTelefono de " + buscar + ": " + agenda.get(buscar));
        }

        // Eliminar un contacto
        agenda.remove("Carlos");
        System.out.println("\nDespues de eliminar a Carlos:");
        System.out.println(agenda);}
}
```

```
TERMINAL

Agenda de contactos:
Ana: 987-6543
Carlos: 555-1234
Juan: 123-4567
Maria: 444-5678

Telefono de Ana: 987-6543

Despues de eliminar a Carlos:
{Ana=987-6543, Juan=123-4567, Maria=444-5678}


** Process exited - Return Code: 0 **
```

Fuente: elaboración propia.

Map con diferentes tipos

Se puede acceder al código desde el siguiente enlace: https://www.online-java.com/bxEapgJuGZ

Tabla 24. Uso de Map con diferentes tipos de clave y valor en Java

```
import java.util.*;
public class Main
{
    public static void main(String[] args){
        // Map con clave Integer y valor String
        Map<Integer, String> estudiantes = new HashMap<>();
        estudiantes.put(101, "Juan Perez");
        estudiantes.put(102, "Maria Garcia");
        estudiantes.put(103, "Carlos Lopez");

        System.out.println("Estudiante con ID 102: " + estudiantes.get(102));

        // Map con clave String y valor List
        Map<String, List<String>> cursos = new HashMap<>();
        cursos.put("Matematicas", Arrays.asList("Algebra", "Calculo", "Geometria"));
        cursos.put("Programacion", Arrays.asList("Java", "Python", "C++"));

        System.out.println("\nCursos de Programacion:");

cursos.get("Programacion").forEach(System.out::println);

}
}
```

```
TERMINAL
Estudiante con ID 102: Maria Garcia

Cursos de Programacion:
Java
Python
C++

** Process exited - Return Code: 0 **
```

Fuente: elaboración propia.

Operaciones comunes con *Map*

En este ejemplo se puede copiar y pegar el código o hacer clic en el siguiente link https://www.online-java.com/GbEi8m0ez7

Tabla 25. Operaciones comunes con un Map en Java

```java
import java.util.*;
public class Main
{
    public static void main(String[] args){
        Map<String, Double> productos = new HashMap<>();

        // Agregar elementos
        productos.put("Notebook", 899.99);
        productos.put("Mouse", 25.50);
        productos.put("Teclado", 45.75);
        productos.put("Monitor", 299.99);

        // Tamano del Map
        System.out.println("Numero de productos: " + productos.size());

        // Verificar si existe una clave
        System.out.println("Existe Notebook? " + productos.containsKey("Notebook"));

        // Verificar si existe un valor
        System.out.println("Existe precio 25.50? " + productos.containsValue(25.50));

        // Obtener todas las claves
        System.out.println("\nProductos disponibles:");
        for (String producto : productos.keySet()) {
            System.out.println("- " + producto);
        }

        // Obtener todos los valores

System.out.println("\nPrecios:");
        for (Double precio : productos.values()) {
            System.out.println("- $" + precio);
        }

        // Reemplazar un valor
        productos.replace("Mouse", 29.99);
        System.out.println("\nNuevo precio del Mouse: $" + productos.get("Mouse"));
}
```

```
TERMINAL

Numero de productos: 4
Existe Notebook? false
Existe precio 25.50? true

Productos disponibles:
- Monitor
- Notebook
- Mouse
- Teclado

Precios:
- $299.99
- $899.99
- $25.5
- $45.75

Nuevo precio del Mouse: $29.99


** Process exited - Return Code: 0 **
```

Fuente: elaboración propia.

Estos ejemplos muestran algunas de las operaciones más comunes con Map en Java. Se recomienda utilizar *Vibe Coding* junto con cualquier herramienta de IA para solicitar ejemplos prácticos con:

- *HashMap*, cuando se necesita acceso rápido sin mantener un orden;
- *TreeMap*, si se desea mantener las claves ordenadas;
- *LinkedHashMap*, para conservar el orden de inserción de los elementos.

CONTINAR

Referencias

[Imagen sin título sobre jerarquía del framework de colecciones en Java], (s.f.). https://jcodebook.com/courses/learn-java-programming/50193/

Referencias bibliográficas de consulta

Álvarez, M. (2001). Qué es la programación orientada a objetos. Desarrollo Web. https://desarrolloweb.com/articulos/499.php

Edmond, D. (2012). The evolution of programming [Infografía]. SiliconANGLE. http://siliconangle.com/blog/2012/02/29/the-evolution-of-programming-infographic/

Kimble, C. (2016). The software crisis. Some notes on the background and nature of the software crisis. http://www.chris-kimble.com/Courses/World_Med_MBA/Software_Crisis.html

La Revista Informática. (2006). Lenguaje de programación BASIC. http://www.larevistainformatica.com/BASIC.htm

Patterson Hume, J. N., & Stephenson, C. (2000). Introduction to programming in Java. Holt Software Associates Inc.

CONTINUAR

Descarga en PDF