El futuro digital es de todos

MinTIC

COLECCIONES

Universidad Industrial de Santander

MisionTIC 2022

4.1. Introducción

Dentro del desarrollo de aplicaciones resulta necesario agrupar elementos en una unidad simple que permita guardar, recuperar , manipular y comunicar los diferentes datos allí agregados. Cada uno de los elementos están relacionados por características en común. Por ejemplo:

Agrupación de rutas de archivos

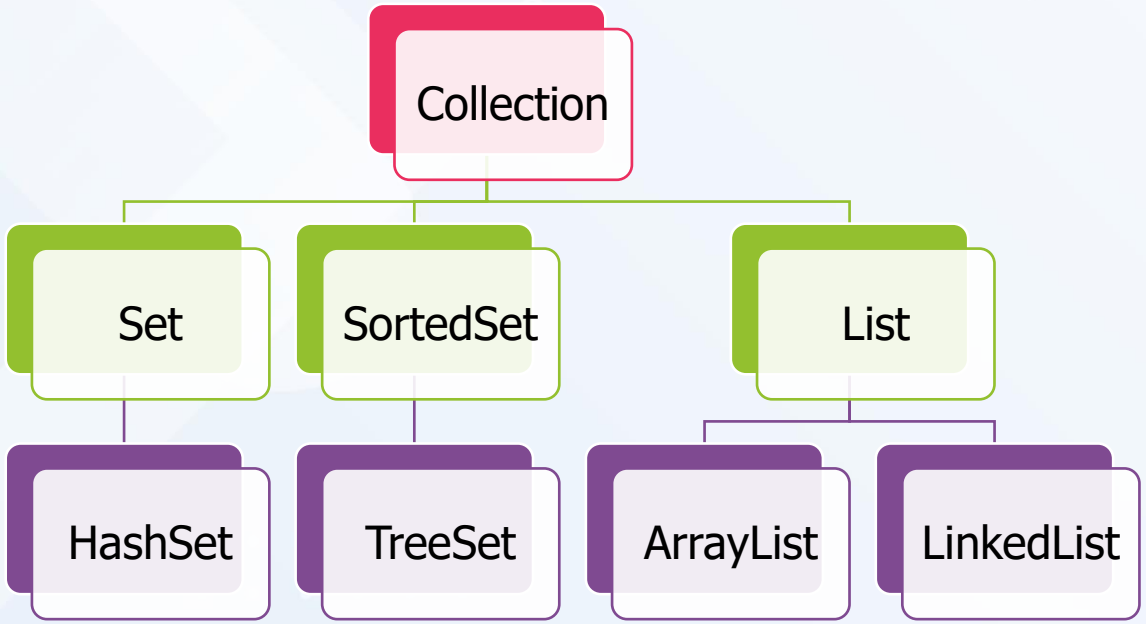
Datos de usuarios

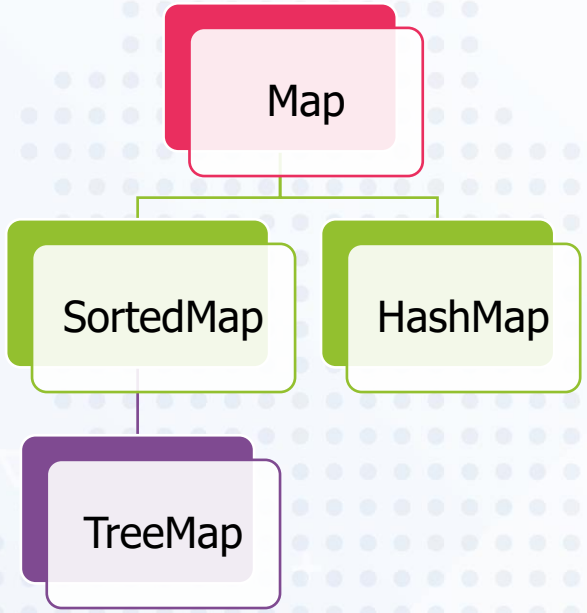
Inventarios de productos

Comentarios de participación en un foro

En general, es una actividad que resulta recurrente; por esta razón, se encuentran en los diferentes lenguajes de programación, estructuras de datos o colecciones, que facilitan la tarea del manejo de los elementos allí contenidos bajo un conjunto de funcionalidades previamente desarrolladas, que hacen más eficiente su tratamiento.

En el lenguaje Java se encuentra una estructura subyacente o Framework de colecciones, que orienta el diseño para la implementación de las diferentes estructuras, conservando funcionalidades generales independientes a los procedimientos desarrollados internamente. Este diseño sigue lineamientos establecidos por tipos de clases abstractas e interfaces que soportadas por el concepto de herencia estandarizan el diseño de las subclases que se cumplan con el diseño establecido.





Jerarquía de herencia de los mapas y las colecciones

4.2. Clases abstractas: modelado e implementación.

Si se piensa en un tipo de clase, es común asumir que dentro de los programas se crearán objetos de este tipo; sin embargo, existen casos donde conviene diseñar clases de las cuales un desarrollador nunca creará objetos. Estas clases son conocidas como clases abstractas, que servirán como clases padre o superclases que permitirán a otras clases heredar estructuras de diseño parcialmente implementadas, las cuales pueden ser sobrescritas o implementadas en las clases hijas o subclases que se relacionan mediante herencia.

Podríamos pensar que así como existen las clases para ser utilizadas como moldes o plantillas para crear objetos a partir de ellas, existen clases que sirven como plantilla para crear otras clases, es en este punto donde las clases abstractas, tienen un valor agregado en términos de desarrollo.

Características de una clase abstracta

Definen parcialmente una clase

En algunos de sus métodos no se conoce la implementación así que es delegada a las subclases

No se pueden crear objetos a partir de ellas

Una clase definida como abstracta puede no contener métodos abstractos

Si una clase hereda de una clase abstracta , debe implementar todos los métodos que hayan sido definidos como abstractos, a menos que ella también sea una clase abstracta

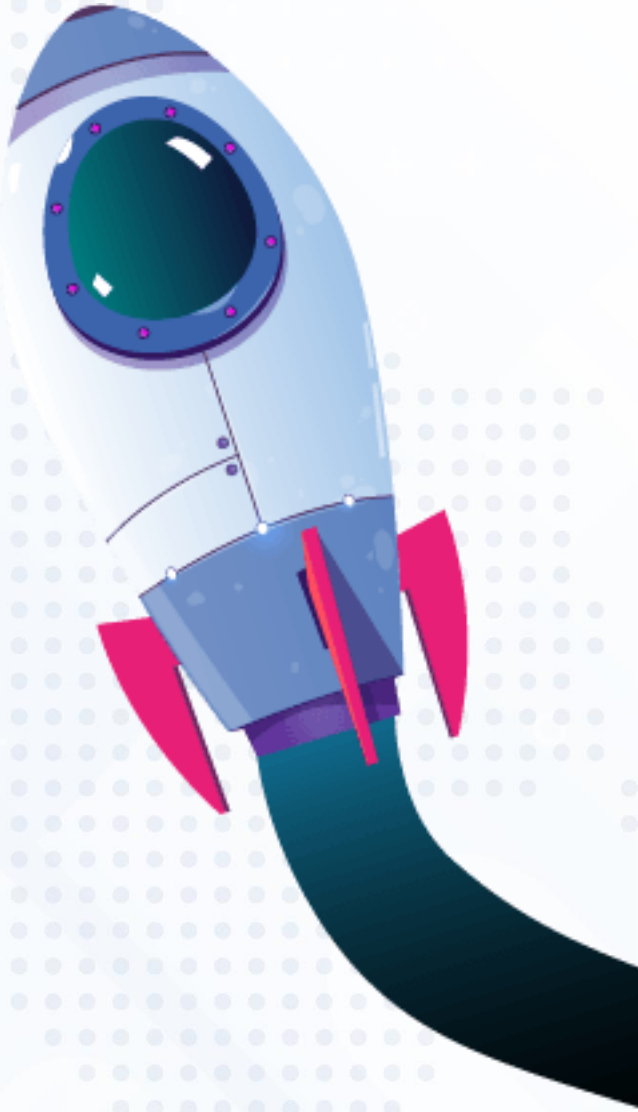
Universidad Industrial de Santander
Mision TIC 2022

Método abstracto

Es un método que se sabe debe ir implementado en las clases que hereden de una clase, pero su implementación es variada en las clases hijas y no se puede generalizar.

4.3. Clases interfaces: modelado e implementación.

Para el caso de las interfaces, suele decirse que son un tipo de clase más abstractas que las abstractas, pero esto ¿qué significa ?, significa que mientras las clases abstractas tienen métodos abstractos y/o métodos parcialmente implementados para heredar a otras clases, las interfaces están compuestas únicamente por métodos abstractos, que orientan el diseño de las clases hijas.



Características de una interface

Su objetivo está centrado en definir un API(Application Programing Interface) para una familia de objetos

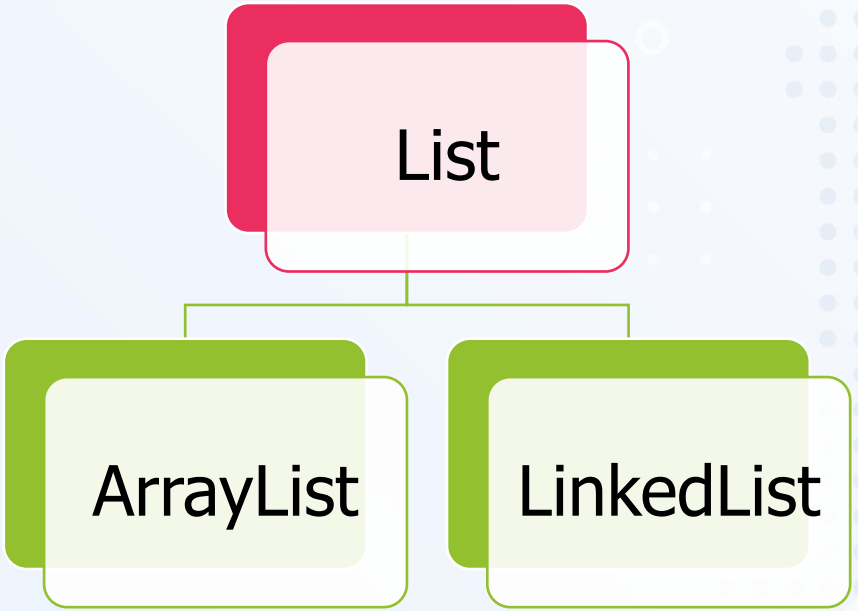
Para su definición se cambia la palabra clave ***class*** por la palabra clave ***interface***

Todos sus métodos tienen visibilidad de tipo ***public,*** Incluso si no se coloca el modificador de manera explícita

No tienen atributos

No tienen método constructor

4.4. Listas



Jerarquía de herencia de la interface List con sus implementaciones ArrayList y LinkedList

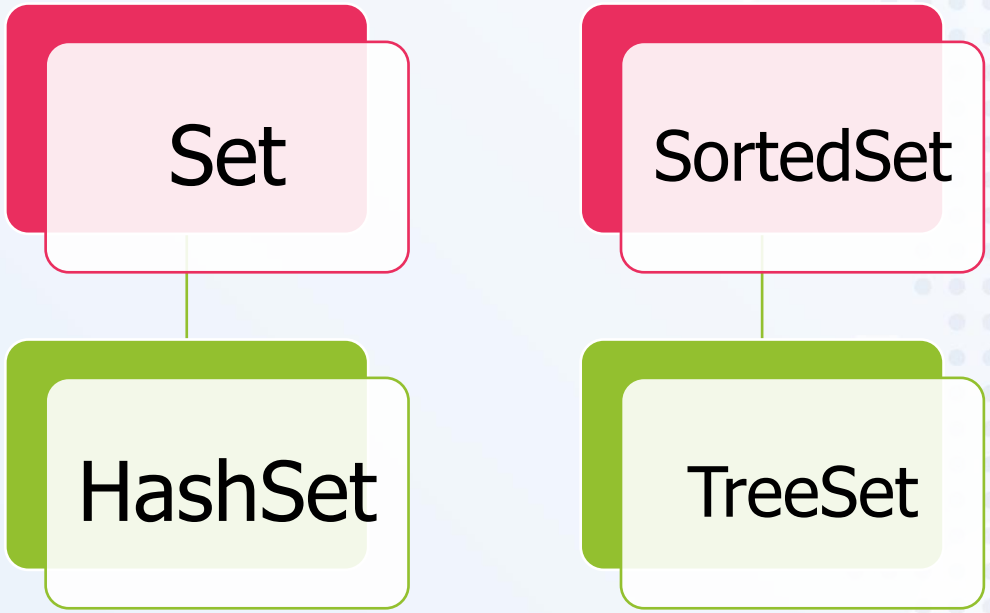
Primera parte del código

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;



public class DemoColecciones {

    public static void main(String[] args) {
       List ejemploObjetoListaLink = new LinkedList();
        ejemploObjetoListaLink.add("elemento1");
        ejemploObjetoListaLink.add("elemento2");
        ejemploObjetoListaLink.add("elemento3");
        ejemploObjetoListaLink.add("elemento3");
        System.out.println("Lista linked list");
        mostrar_elementos(ejemploObjetoListaLink);
```

Segunda parte del código

```
        List ejemploObejetoListaArray = new ArrayList();
        ejemploObejetoListaArray.add("elemento1");
        ejemploObejetoListaArray.add("elemento2");
        ejemploObejetoListaArray.add("elemento3");
        ejemploObejetoListaArray.add("elemento3");
        System.out.println("Lista array list");
        mostrar_elementos(ejemploObejetoListaArray);
    }
    public static void mostrar_elementos(Collection coll) {
        Iterator iter = coll.iterator();
        while (iter.hasNext()) {
            String elem = (String) iter.next();
            System.out.print(elem + " ");
        }
        System.out.println();
    }
}
```

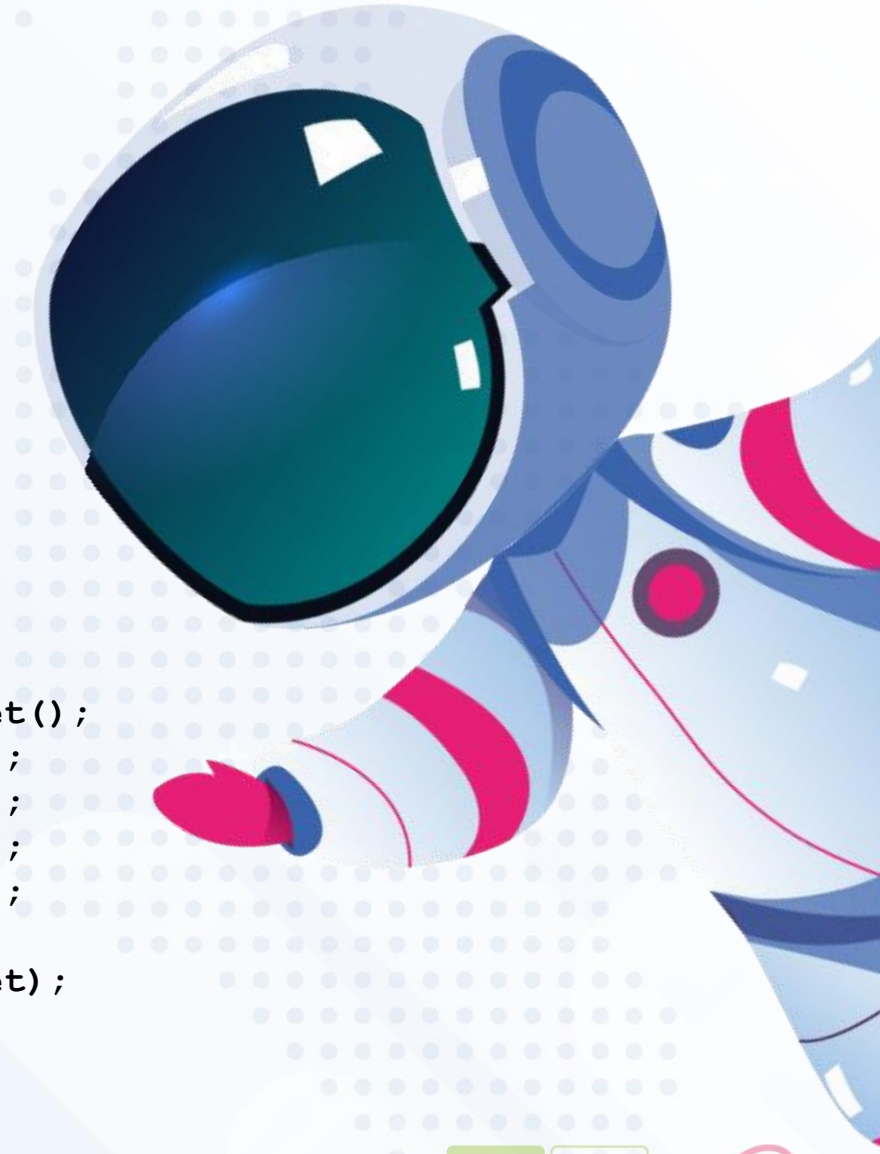
4.5. Conjuntos



Jerarquía de herencia de las interfaces Set y SortedSet con sus implementaciones HashSet y TreeSet

Primera parte del código

```
import java.util.Collection;
import java.util.Iterator;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

public class DemoColecciones {

    public static void main(String[] args) {
        Set ejemploObjetoConjuntoHashSet = new HashSet();
        ejemploObjetoConjuntoHashSet.add("elemento1");
        ejemploObjetoConjuntoHashSet.add("elemento3");
        ejemploObjetoConjuntoHashSet.add("elemento2");
        ejemploObjetoConjuntoHashSet.add("elemento2");
        System.out.println("Conjunto tipo hash");
        mostrar_elementos(ejemploObjetoConjuntoHashSet);
```



Segunda parte del código

```
        SortedSet ejemploObjetoConjuntoTreeSet = new TreeSet();
        ejemploObjetoConjuntoTreeSet.add("elemento1");
        ejemploObjetoConjuntoTreeSet.add("elemento3");
        ejemploObjetoConjuntoTreeSet.add("elemento2");
        ejemploObjetoConjuntoTreeSet.add("elemento2");
        System.out.println("Conjunto tipo tree");
        mostrar_elementos(ejemploObjetoConjuntoTreeSet);
    }

    public static void mostrar_elementos(Collection coll) {
        Iterator iter = coll.iterator();
        while (iter.hasNext()) {
            String elem = (String) iter.next();
            System.out.print(elem + " ");
        }
        System.out.println();
    }
}
```

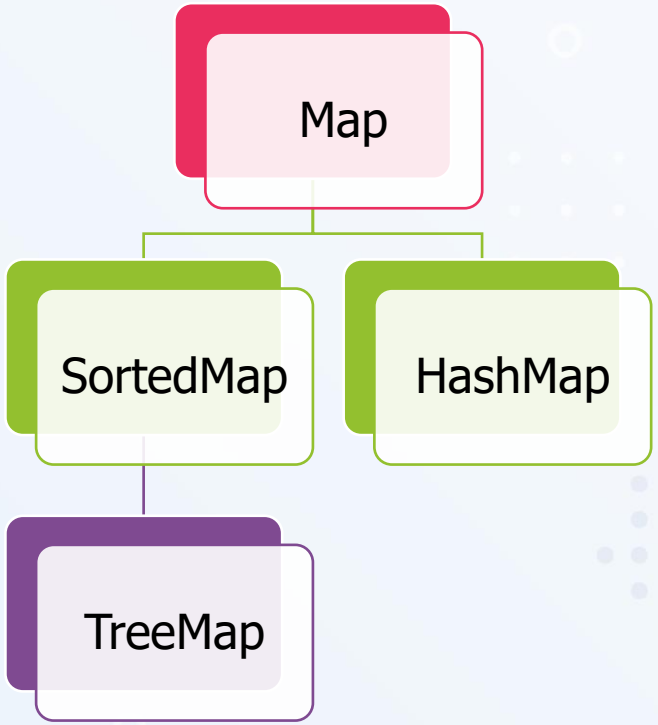
Métodos más comunes para usar colecciones

```
add(E e)
add(int i, E e)
addAll(Collection c)
clear( )
contains(Object obj)
remove(int i)
remove(Object obj)
isEmpty( )
size( )
get(int i)
 indexOf(Object o)
```

Consultar

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

4.6. Mapas



Jerarquía de herencia de la interface Map con sus implementaciones HashMap y TreeMap

Primera parte del código

```
import java.util.Collection;
import java.util.Iterator;
import java.util.Map;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.HashMap;


public class DemoColecciones {

    public static void main(String[] args) {

        Map ejemploObjetoMapaHash = new HashMap();
        ejemploObjetoMapaHash.put("clave1", "valor1");
        ejemploObjetoMapaHash.put("clave2", "valor2");
        ejemploObjetoMapaHash.put("clave3", "valor3");
        ejemploObjetoMapaHash.put("clave3", "valor3");

        System.out.println("Mapa tipo hash");
        mostrar_elementos(ejemploObjetoMapaHash.keySet());
        mostrar_elementos(ejemploObjetoMapaHash.values());
```

Segunda parte del código

```
        SortedMap ejemploObjetoMapaTree = new TreeMap();
        ejemploObjetoMapaTree.put("clave1", "valor1");
        ejemploObjetoMapaTree.put("clave2", "valor2");
        ejemploObjetoMapaTree.put("clave3", "valor3");
        ejemploObjetoMapaTree.put("clave3", "valor3");
        System.out.println("Mapa tipo hash");
        mostrar_elementos(ejemploObjetoMapaTree.keySet());
        mostrar_elementos(ejemploObjetoMapaTree.values());
    }

    public static void mostrar_elementos(Collection coll) {
        Iterator iter = coll.iterator();
        while (iter.hasNext()) {
            String elem = (String) iter.next();
            System.out.print(elem + " ");
        }
        System.out.println();
    }
}
```

Métodos más comunes de los mapas

```
clear()
containsKey(Object key)
boolean containsValue(Object value)
isEmpty()
keySet().
put(K key, V value)
remove(Object key)
remove(Object key, Object value)
replace(K key, V value)
size()
values()
```

Universidad Industrial de Santander
MisiónTIC2022

Resumen

- **Collection :** Es la interfaz raíz en la jerarquía de colecciones
- **Set:** Una colección que no permite duplicar elementos.
- **List:** Es una colección que permite agregar elementos repetidos
- **Map:** Es una estructura de datos creada a partir del uso de dos colecciones que asocia claves y valores. Además no puede contener claves duplicadas.

Iterador e iterador parametrizado

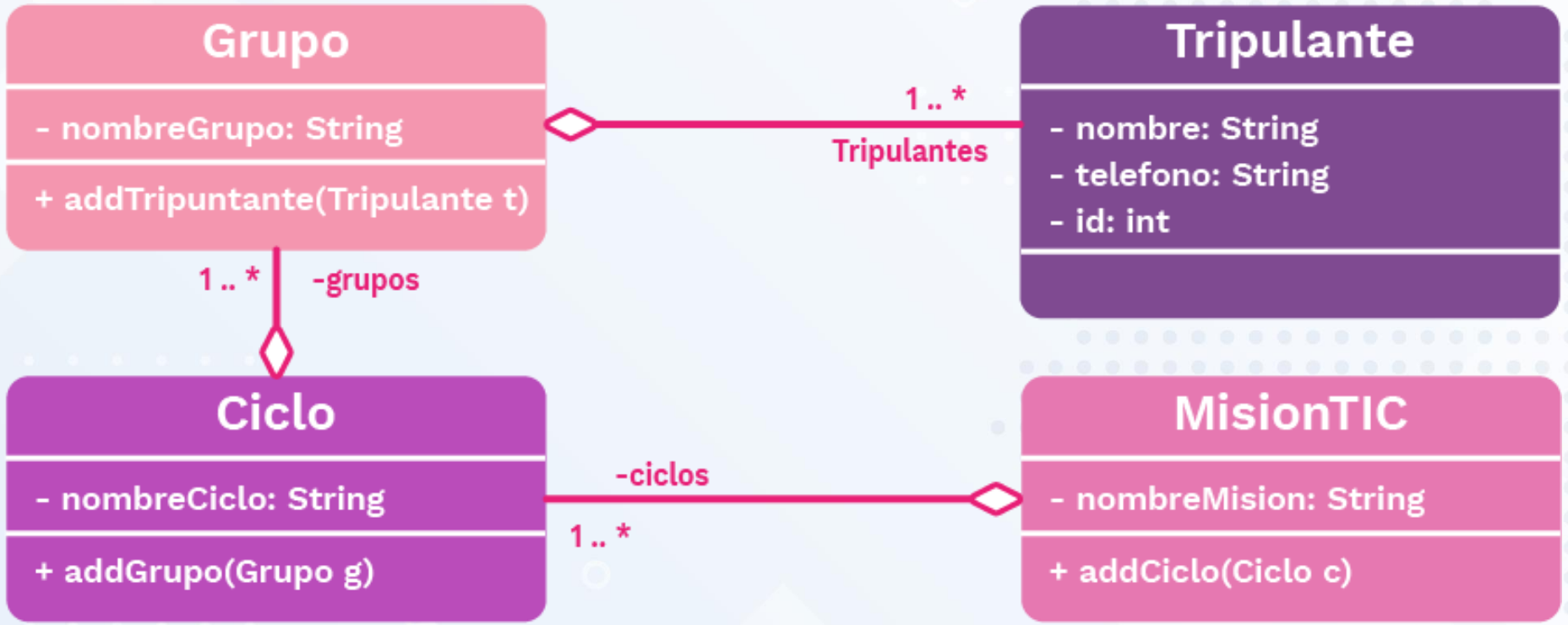
```
List ejemploListaSp = new ArrayList();
        ejemploListaSp.add(4.3);
        ejemploListaSp.add(3.5);
        Iterator iteradorSp = ejemploListaSp.iterator();
        while (iteradorSp.hasNext()) {
            Double ele = (Double) iteradorSp.next();
            System.out.println(ele);
        }

        List<Double> ejemploObjetoLista=  new ArrayList<>();
        ejemploObjetoLista.add(4.3);
        ejemploObjetoLista.add(3.5);

        Iterator<Double> iteradorListaP = ejemploObjetoLista.iterator();
        while (iteradorListaP.hasNext()) {
            Double ele = iteradorListaP.next();
            System.out.println(ele);
        }
```

4.7. Clases contenedoras predefinidas en los lenguajes de programación (patrón contenedor)

Este patrón es frecuentemente aplicado a diferentes contactos donde se requiere que un elemento sea contenido como atributo de una clase o como uno de los elementos del atributo de una clase

- Misión Tic compuesta por ciclos
- Ciclos compuestos por grupos
- Grupos compuestos por tripulantes

Diagrama UML de aplicación del patrón contenedor en un ejemplo



Para implementar este tipo de patrón se hace necesario:

- Una clase contenedora
- Una clase contenida
- Un atributo de tipo lista para agregar objetos en la clase contenedora
- Un método que permite agregar objetos en el atributo de tipo lista de la clase contenedora

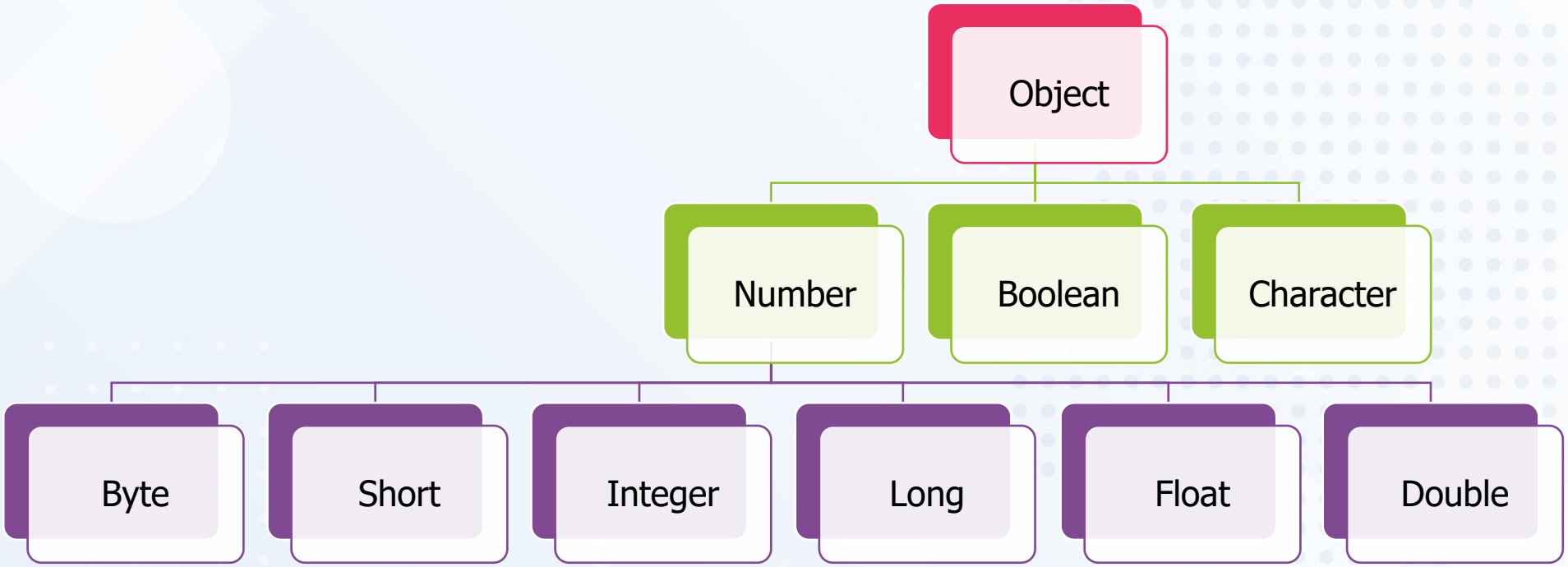
Clases envoltura o wrapper

Así como existen elementos de composición aplicados desde las relaciones entre clases, existen unas clases llamadas envoltura, clases contenedoras predefinidas del lenguaje Java que sirven como vehículo a las variables de tipo primitivo para ser almacenadas en estructuras de datos que contienen objetos.

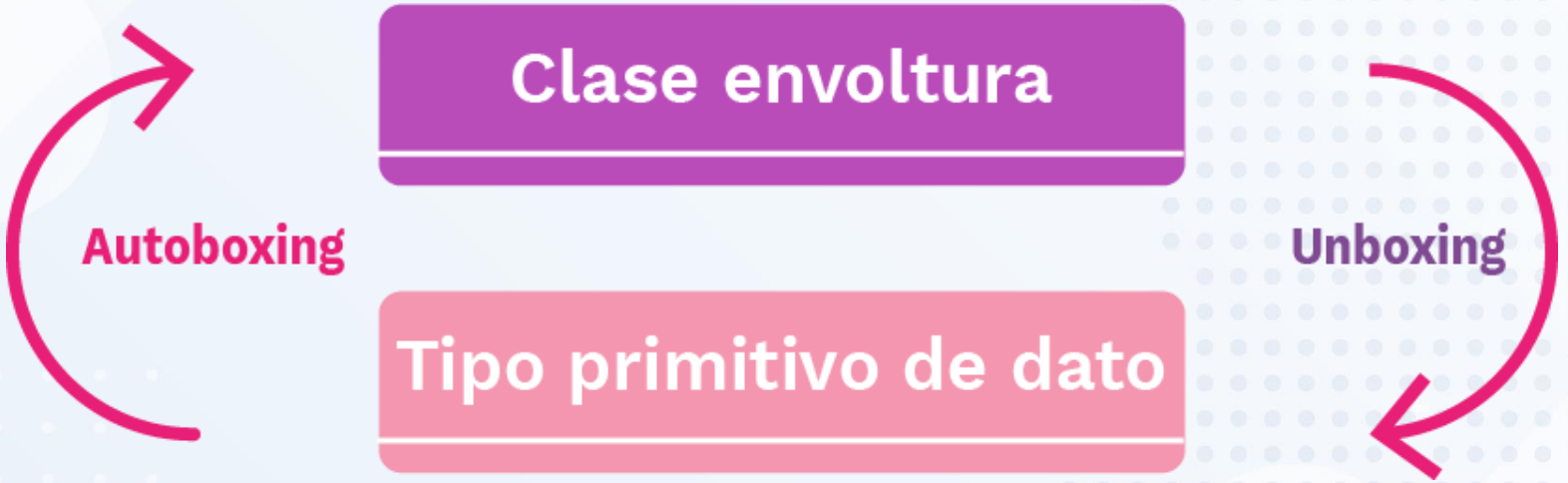
Para cada tipo primitivo existe su Clase tipo envoltura como se muestra a continuación:

Tipo primitivo de dato	Clase envoltura
char	Character
byte	Byte
short	Short
long	Long
int	Integer
float	Float
double	Double
boolean	Boolean

La figura mostrada a continuación representa la jerarquía de herencia de las clases envoltura



Algo importante a mencionar, es que los tipos primitivos son pasados en los argumentos de los métodos por valor, mientras que los objetos son pasados por referencia. Además que envolver estos datos primitivos da acceso a una variedad de métodos, permite realizar cambios de tipo de los primitivos envueltos. Por su parte, el proceso de envolver un dato se hace de manera automática a su clase envoltura sin especificarlo, pero para extraer el dato primitivo, se deben utilizar los métodos de la clase envoltura.


Clase envoltura
Autoboxing
Unboxing
Tipo primitivo de dato

Enums de Java

Un enum es una Tipo de "clase" especial que representa un grupo de constantes (variables inmutables), o un listado de objetos prefabricados listos para usar

Por ejemplo si para un programa se quisiera tener información sobre los continentes, lista para usar se puede crear una clase tipo ENUM con objetos etiquetados con el nombre de cada continente átomo se muestra en la siguiente implementación:

```
public class Main {
  enum Continente {
    AFRICA,
    EUROPA,
    ASIA,
    AMERICA,
    OCEANIA
  }

  public static void main(String[] args) {
    Continente myVar = Continente.AFRICA;
    System.out.println(myVar);
  }
}
```

Una vez etiquetados nuestros objetos dentro de la clase tipo ENUM, podemos realizar modificaciones para agregar atributos como la cantidad de países de cada uno, con su respectivo método get para tener acceso al atributo, y la implementación de un constructor que permita inicializarlos en el momento de su etiquetado y creación:

```
public class Main {
  enum Continente {
    AFRICA(54),
    EUROPA(50),
    ASIA(49),
    AMERICA(35),
    OCEANIA(15);
    private int numeroDePaises;
    Continente(int numeroDePaises){
    this.numeroDePaises=numeroDePaises;
    }
    public int getNumeroDePaises(){
    return numeroDePaises;
    }
  }

  public static void main(String[] args) {
    Continente myVar = Continente.AFRICA;
    System.out.println(myVar.getNumeroDePaises());
  }
}
```

Material complementario

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

https://bibliotecavirtual.uis.edu.co:4259/es/lc/uis/titulos/50117

Referencias Bibliográficas

Deitel, P. & Deitel, H. (2008). JAVA JAVA : COMO PROGRAMAR (7ed). Pearson.
Sarcar, V. (2016). Java design patterns. Apress.