FUNDACIÓN ACCENTURE

accenture

JAVA II

3. CONCEPTOS RELACIONADOS CON LA POO

Profundiza en nuevos conceptos de Java que no has visto hasta ahora. Conoce en qué consiste la composición, las clases genéricas, las colecciones, y aprende a gestionar excepciones.

TABLA DE CONTENIDOS

JAVA I

3.1. COMPOSICIÓN

¿QUÉ ES LA COMPOSICIÓN?

La composición es un tipo de relación dependiente, donde un objeto más complejo está constituido por objetos más simples. Es decir, la composición es la **agrupación de varios objetos como atributos de otro objeto de la clase.**

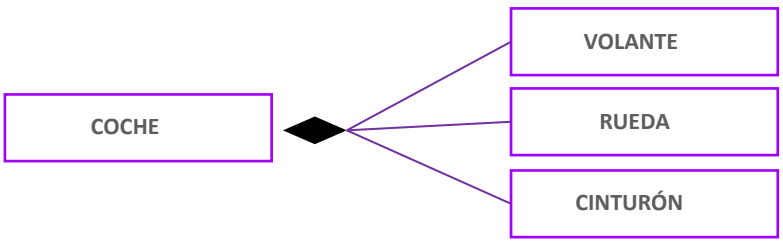
UN EJEMPLO DE COMPOSICIÓN

Pensemos en la **clase Coche**, de ella creamos nuestro objeto (c1) que a la vez estará compuesto de diferentes objetos: volante, cinturón, rueda, etc.

La composición nos permite poder reutilizar los objetos: volante, cinturón, etc. del objeto coche c1 para crear otro coche c2. Es decir, podemos **reutilizar código y ahorrar tiempo**. El tiempo de vida de nuestro coche c1 estará condicionado por el tiempo de vida de los objetos que lo componen. Al igual que en la vida real, si los mecanismos del coche quedan obsoletos, no hay coche.

Y... ¿Dónde están los objetos ruedas o cinturones? En nuestra clase Coche no, porque esta clase sirve para crear coches, no ruedas o cinturones. Estos objetos se encontrarán en sus propias clases. Por ello, la relación que estos objetos tienen con nuestra clase Coche no es de herencia. No es una relación "es un", sino **"tiene un..."**.

En este caso la relación se representará con un rombo de color negro.



PREMISA COMPOSICIONES EN JAVA	NUESTRO COCHE
La composición crea una relación "tiene un..." entre objetos.	El coche tiene un volante, cinturón, rueda.
Los objetos que componen a la clase contenedora deben existir desde el principio.	El volante lo creo antes de crear el coche.
No debería una clase contenedora existir sin alguno de sus objetos componentes.	Para que funcione el coche debe tener volante.
Objetos componentes (partes) como la clase contenedora, nacen y mueren al mismo tiempo. Tienen el mismo tiempo de vida.	Si el volante se estropea no funciona el coche.

3.2. GENÉRICOS

¿QUÉ ES UN GENÉRICO?

El término **genéricos** hace referencia a los **tipos parametrizados**. Nos permiten **crear clases, interfaces y métodos** en los que el **tipo de datos** sobre los que operan se especifica como **parámetro** (solo funcionan con tipos de referencia, no se pueden usar primitivos). Una clase, interfaz o método que funciona con uno o varios tipos como parámetro se denomina genérico.

Un genérico nos permite escribir un **código que se pueda reutilizar** para objetos de diversos tipos. De esta manera, se evita crear una clase para cada tipo de objeto que vamos a manejar. Además, nos permite crear clases, interfaces y métodos en los que se desconoce el tipo de dato con el que quieres trabajar.

UN EJEMPLO DE GENÉRICO

Imagina la **clase Bolsa**. En función de los objetos que contenga, podrá ser una bolsa de caramelos o una bolsa de frutos secos. Pues bien, debemos declarar la clase Bolsa de tipo Genérico y esto lo hacemos con la letra **T** entre corchetes angulares **<>**. Y llamaremos "contenido" a la variable que cambiará según el tipo de bolsa.

De esta manera cuando construyamos un objeto de esta clase será el momento de especificar el tipo de bolsa que deseamos.

Ten en cuenta que... cualquier identificador válido puede ser utilizado, pero **T** es el tradicional. Además, se recomienda que los nombres de los parámetros de tipo sean de un solo carácter y mayúsculas. Otros nombres de parámetros de uso común son **V** y **E**.

APLICACIÓN EN JAVA

1. Primero indicamos que esta clase utiliza un tipo de parámetro (**T**). Después, declaramos T como un atributo privado de la clase y usamos el método **getter** para obtener el valor de T. Por último, usamos el método **setter** para modificar el valor de T. Veamos cómo sería:

```
public class Bolsa<T> {
    private T contenido;
    public T getContenido() {
        return contenido;
    }
    public void setContenido(T nuevoValor){
        this.contenido = nuevoValor;
    }
}
```

Parámetro: un parámetro representa un dato que ofrece una función con un fin específico. En el caso de los genéricos, el tipo de parámetro será el valor que le demos.

2. Creamos la clase y el **método main** para iniciar el programa. A continuación**, invocamos a la clase genérica (Bolsa)** e indicamos el tipo de dato que va a manejar entre <>. En nuestro caso será un **String** (frutos secos o caramelos) y le llamamos g1. Usaremos el método **setter** para cambiar el valor de g1 a Caramelos.

```
public class UsoBolsa{
     public static void main (String[] args){
            Bolsa<String> g1 = new Bolsa<>();
            g1.setContenido("Caramelos");
            System.out.printIn(g1.getContenido());
     }
}
```

3. Finalmente mandaremos la instrucción de imprimir el valor del contenido de la bolsa, y **la consola devolverá "Caramelos".**

USO DE GENÉRICOS

EN CLASES

Hasta ahora habíamos visto que para definir un atributo en una clase debíamos especificar su tipo. Pues con el uso de genéricos podemos crear o definir una clase sin especificar el tipo del atributo y, en el momento de instanciar, se especifica el tipo que queremos que tome el atributo pasado como parámetro.
También se puede declarar más de un parámetro de tipo en un tipo genérico, para ello simplemente se usa una lista separada por comas, como en el siguiente ejemplo.

```
public class GenericoMultiple<K,V> {
       private K variable1;
       private V variable2;
       public K getVariable1() {
              return variable1;
       }
       public void setVariable1(K variable1) {
              this.variable1 = variable1;
       }
       public V getVariable2() {
              return variable2;
       }
       public void setVariable2(V variable2) {
              this.variable2 = variable2;
       }
}
class DemoGenMultiple{
       public static void main(String[] args) {
              GenericoMultiple<String, Integer> gm = new GenericoMultiple<>();
              gm.setVariable1("texto");
              gm.setVariable2(20);
       }
}
```

EN MÉTODOS

Un método genérico es aquel que se adapta o que podemos utilizar con cualquier tipo. Los métodos genéricos no necesitan ser creados en una clase genérica.
Por ejemplo, vamos a crear la clase MisMatrices y en ella creamos el método genérico getElementos(), que nos va a devolver un String indicando la longitud del array que le pasemos como parámetro. Al declararlo como genérico, podrá recibir cualquier tipo de array.

```
public class MisMatrices {
	public static <T> String getElementos(T[] a) {
		return "El array tiene " + a.length + " elementos.";
	}
	public static void main(String[] args) {
		//Creamos un array con nombres
		String nombres[]={"Pablo", "Rebeca", "Ana"};
		//Creamos un array con números
		Integer numeros[]={7, 9, 2 6};
		//Utilizamos el mismo método con un array numérico y con otro array de texto
		System.out.println(MisMatrices.getElementos(nombres));
		System.out.println(MisMatrices.getElementos(numeros));
	}
}
```

EN INTERFACES

Una interfaz genérica se declara de la misma manera que una clase genérica. Además, si una clase implementa una interfaz genérica, esa clase también debe ser genérica, ya que también debe recibir el tipo como parámetro.
Por ejemplo, creamos la interfaz Contenedor con un método contenido() que verificará si un elemento especifico está dentro de un objeto. A continuación, creamos la clase MiClase que implementa la interfaz Contenedor.

```
interface Contenedor <T> {
	public abstract boolean contenido (T obj);
}
class MiClase <T> implements Contenedor <T> {
	T[] array; //Array del tipo genérico
	MiClase(T[] o){
		array=o;
	}
	//Implementación del método contenido() de la interfaz Contenedor
	public boolean contenido (T o) {
		for (T x: array)
			if (x.equals(o)) return true;
		return false;
	}
}
class DemoIFGen{
	public static void main(String[] args) {
		Integer x[]={1,2,3};
		MiClase<Integer> ob = new MiClase<Integer>(x);
		if (ob.contenido(2))
			System.out.println("2 está en el array");
		else
			System.out.println("2 NO está en el array");
	}
}
```

GENÉRICOS DE TIPO LIMITADO

A la hora de crear una clase genérica, el tipo de parámetro puede ser reemplazado por cualquier tipo de clase, sin ningún tipo de limitación, pero en algunas situaciones se tiene la necesidad de **delimitar los tipos de datos permitidos para una clase**.

Por ejemplo, se quiere crear una clase que realice operaciones matemáticas con diferentes tipos de números, necesitamos limitar el tipo de dato para que permita manejar tipos del tipo numérico. (Byte, Short, Integer, Long…), y de esta manera evitar que se usen cualquier otro tipo de dato.

Para limitar el tipo de datos permitidos en una clase se utiliza la palabra **extends.**

```
    public class Bolsa<T extends Number>
```

A la hora de poner nuestro programa en funcionamiento solo se podrá trabajar con tipos de datos numéricos. Por ejemplo, la Bolsa puede tener valores enteros (Integer) o valores decimales (Double), pero no podrá tomar un valor de tipo texto (String).

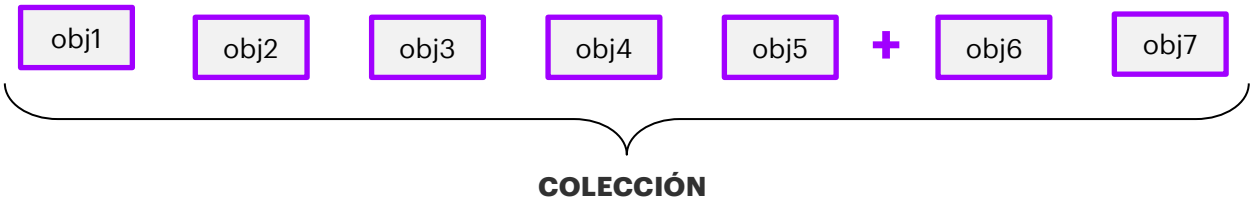
```
public class UsoBolsa{
    public static void main (String[] args) {

       Bolsa<Integer> g1 = new Bolsa<>();
       Bolsa<Double> g2 = new Bolsa<>();
       Bolsa<String> g3 = new Bolsa<>(); //Error
    }
}
```

3.3. COLECCIONES

¿QUÉ SON LAS COLECCIONES?

Las colecciones son similares a los **Arrays**, la diferencia es que la Colección es un **almacén de objetos dinámicos**, es decir que el almacén puede crecer o disminuir durante la ejecución del programa, mientras que un array tiene un tamaño fijo.

Por ejemplo... Imaginemos que hemos creado una colección para que almacene 5 objetos y necesitamos que durante la ejecución del programa almacene 2 objetos más, gracias a esta funcionalidad de Java, esa colección es capaz de crecer.



VENTAJAS DE LAS COLECCIONES

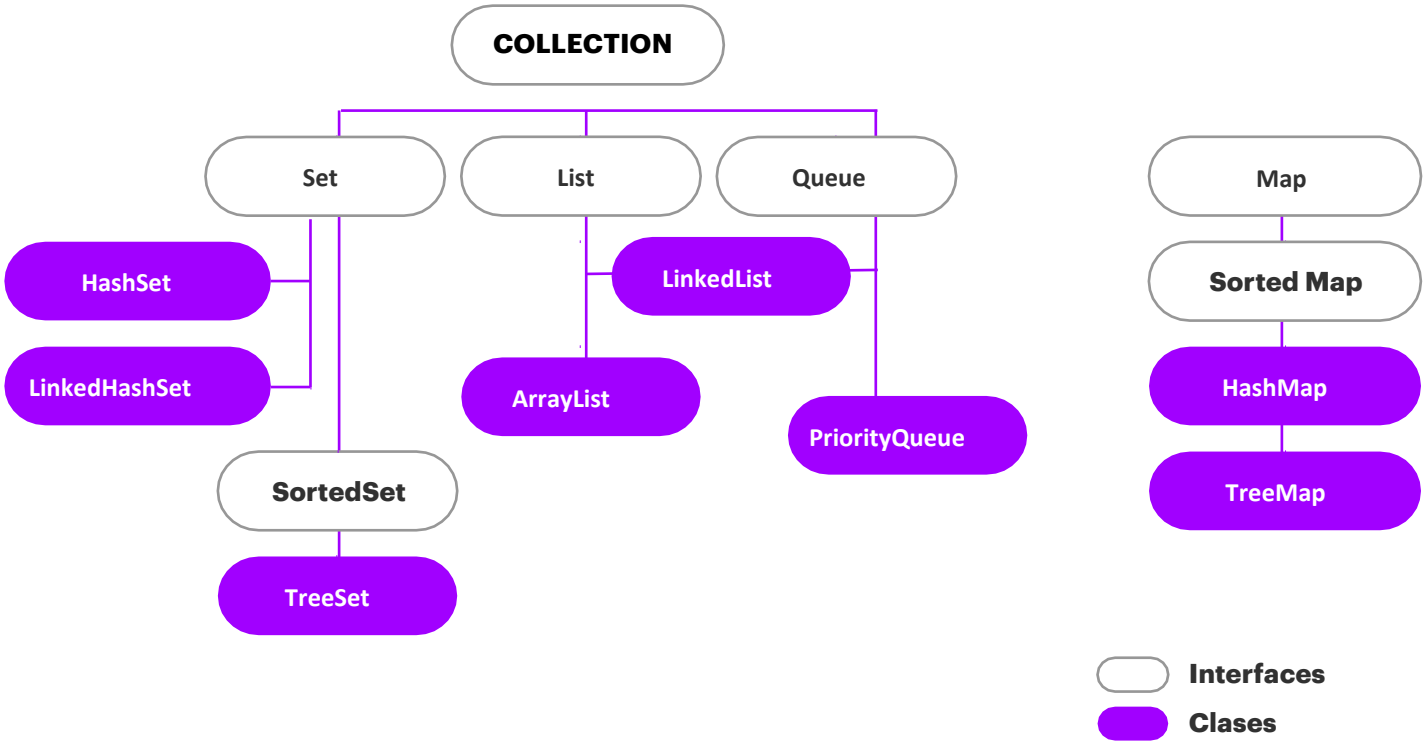
- ✓ Pueden cambiar de tamaño dinámicamente (se pueden insertar y eliminar objetos)
- ✓ Pueden ir provistas de ordenamiento (podemos ordenar los objetos que tenemos dentro)

Array: contenedor de objetos o de datos de tipo primitivo como cadenas de texto. Tiene un tamaño fijo establecido por la cantidad de elementos que lo componen.

TIPOS DE COLECCIONES

Para trabajar con colecciones hay que hacer uso del ecosistema de **Interfaces y clases de la API de Java** (se encuentran dentro del paquete java.util). Además, cada tipo de interfaz maneja diferentes tipos de clases, pero todas ellas extienden de la interface **Collection**, y cada una de estas diferentes interfaces y clases aportan distintas funcionalidades.

- ✓ Las interfaces especifican el comportamiento abstracto de cada colección.
- ✓ Las clases tendrán la implementación del comportamiento de cada colección.



A continuación, vamos a entrar en detalle en cada una de estas interfaces...

TIPOS DE INTERFACES DE COLECCIÓN

1. INTERFAZ *SET*

Es una colección **sin ordenar** que **no permite elementos duplicados** y puede contener, como máximo, **un elemento nulo**. Esta interfaz contiene, únicamente, los métodos heredados de Collection.

Es importante destacar que, para comprobar si los elementos están duplicados, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos **equals** y **hashCode**.

Alguno de los métodos básicos para trabajar con la interface ***Set*** son:

- Objeto.**size**(): Devuelve el número de elementos que contiene.
- Objeto.**isEmpty**(): Verifica si se encuentra vacío.
- Objeto.**add**(elemento): Inserta un elemento en el conjunto siempre que no esté repetido. Devuelve true o false para indicar si se ha podido insertar.
- Object.**remove**(elemento): Elimina el elemento que se especifica si se encuentra en el set. Devuelve true o false para indicar si el elemento está o no presente y si fue eliminado.

Como es una interfaz, necesita una implementación. Aquí tenemos las 3 implementaciones más comunes para la interfaz ***Set***:

- **HashSet**: Los elementos (no repetidos) no están ordenados, es decir, esta implementación almacena los elementos como en una bolsa sin ordenar. Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones.
- **LinkedHashSet**: Los elementos (no repetidos) no están ordenados, pero funcionan como punteros, es decir, esta implementación almacena los elementos en función del orden de inserción.
- **TreeSet**: Los elementos se almacenan distribuidos como una jerarquía de árbol, es decir, esta implementación almacena los elementos ordenándolos en función de sus valores. Los elementos almacenados deben implementar la interfaz **Comparable** que veremos más adelante.

Declaración:

Set <tipo> nombreConjunto = new tipoclase <tipo>();

Para nuestro ejemplo: `Set <String> titulares = new HashSet<String>();`

Aplicación en Java:

Para nuestro ejemplo, el siguiente programa usando la interfaz ***Set*** nos devolvería "false" ya que se están repitiendo los nombres de los titulares. El programa podrá ejecutarse cuando los elementos de la colección no estén repetidos.

```
public class CuentaCorriente{
      public static void main (String[] args) {
            Set <String> titulares = new HashSet<String>();
            System.out.print(titulares.add("Laura"));
            System.out.print(titulares.add("Miguel"));
            //Al no existir "Laura" y "Miguel", se insertan y
            devuelven true
            System.out.print(titulares.add("Laura"));
            System.out.print(titulares.add("Miguel"));
            //Al existir "Laura" y "Miguel", no se insertan y
            devuelven false
            System.out.printIn("N° elementos: " + titulares.size());
            //Mostraría en pantalla "N° elementos: 2"
       }
}
```

2. INTERFAZ ***LIST***

Es una colección **ordenada** de elementos que, a diferencia de la interfaz **Set**, **sí permite elementos duplicados**. Es como un array pero de dimensión variable, es decir, el acceso es por la posición del índice. Al permitir hacer más cosas, también se vuelve menos eficiente (***Set*** es más rápido).

Los métodos básicos para trabajar con estos elementos son los siguientes:

- Object.**add**(elemento): Inserta un elemento en la lista.
- Object.**add**(indice, elemento): Inserta un elemento en la posición que se indica.
- Object.**get**(indice): Obtiene el elemento de una posición concreta.
- Object.**indexOf**(elemento): Nos dice cuál es el índice de dicho elemento dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.
- Object.**remove**(indice): Elimina el elemento que se encuentre en una posición concreta.
- Object.**set**(indice, elemento): Establece el elemento en una posición concreta sobrescribiendo el elemento que hubiera anteriormente en dicha posición.

Como es una interfaz, necesita una implementación, aquí tenemos las 2 implementaciones más comunes para ***List***:

- **ArrayList**: Esta es la implementación típica. Se basa en un array redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de las situaciones.
- **LinkedList**: Esta implementación se basa en una lista doblemente enlazada de los elementos teniendo, cada uno de los elementos, un puntero al anterior y al siguiente elemento.

Declaración:

List <tipo> nombreList = new tipoclase<tipo>();

Para nuestro ejemplo: `List <String> listaAsistentes = new ArrayList<String>();`

Aplicación en Java:

Para nuestro ejemplo, el siguiente programa usando la interfaz ***List*** sí nos permitirá tener una colección con elementos repetidos. En este caso, hay dos asistentes al evento con el mismo nombre “María”.

```
public class Evento{
	public static void main (String[] args) {
		List <String> listaAsistentes = new ArrayList<String>();
		listaAsistentes.add(“Pedro”);
		listaAsistentes.add(“María”);
		listaAsistentes.add(“Antonio”);
		listaAsistentes.add(“María”);
		//Los 4 se dan de alta a pesar de que se repite “María”
		System.out.printIn(“N° elementos: “ + listaAsistentes.size());
		//Mostraría en pantalla “N° elementos: 4”

	}
}
```

3. INTERFAZ ***QUEUE***

Son colecciones de **tipo cola**, que no permiten el acceso aleatorio (no puedo acceder al elemento 10 o 12 de una colección de 20 elementos), permitiéndose sólo acceder a elementos que se encuentren al principio o al final de la cola (elementos 1 y 20). Según vaya creciendo la colección o vayamos eliminando elementos, estas posiciones primera y última pueden cambiar.

Los métodos básicos para trabajar con estos elementos son los siguientes:

- Object.**add**(elemento): Inserta el elemento especificado en esta cola.
- Object.**element**(): Recupera, pero no elimina, el encabezado de esta cola.
- Object.**peek**(): Recupera, pero no elimina, el encabezado de esta cola o devuelve un valor nulo si esta cola está vacía.
- Object.**remove**(): Recupera y elimina el encabezado de esta cola.
- Object.**poll**(): Recupera y elimina el encabezado de esta cola, o devuelve un valor nulo si esta cola está vacía.

Como es una interfaz, necesita una implementación, aquí tenemos las 2 implementaciones más comunes para la interfaz ***Queue***:

- **LinkedList**: Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.
- **PriorityQueue**: Una variante de una cola clásica la implementa la clase PriorityQueue. Cuando se agregan elementos a la cola se organiza según su valor, por ejemplo, si es un número se ingresan de menor a mayor.

Declaración:

Queue <tipo> nombreConjunto = new tipoclase <tipo>();

Para nuestro ejemplo: `Queue <String> colaPersonas = new LinkedList<String>();`

Aplicación en Java:

Para nuestro ejemplo, el siguiente programa usando la interfaz Queue nos permite crear una colección de personas que se encuentran en cola.

```
public class Carniceria{
       public static void main (String[] args) {
          Queue <String> colaPersonas = new LinkedList<String>();
          colaPersonas.add("Juan");
          colaPersonas.add("Ana");
          colaPersonas.add("Luis");
       System.out.printIn("N° elementos: colaPersonas.size());
       //Mostraría en pantalla "N° elementos: 3"
       }
}
```

4. INTERFAZ *MAP*

Aunque muchas veces se hable de los mapas como una colección, en realidad no lo son, ya que NO heredan de la interfaz Collection.

Entonces, ¿qué es una interfaz ***Map***? Es una colección que contiene parejas (par de valores), una clave y un elemento, es decir, ***Map*** es un objeto que relaciona una clave (key) con un valor. Contendrá un conjunto de claves, y a cada clave se le asociará un determinado valor. Tanto la clave como el valor puede ser de cualquier tipo o cualquier objeto.

Los métodos básicos para trabajar con estos elementos son los siguientes:

- Object.**get**(clave): Nos devuelve el valor asociado a la clave indicada.
- Object.**put**(clave, valor): Inserta una nueva clave con el valor especificado.
- Object.**remove**(clave): Elimina una clave y su valor asociado.
- Object.**keySet**(): Nos devuelve el conjunto de claves registradas.
- Object.**size**(): Nos devuelve el número de parejas (clave,valor) registradas.

Pero, ¿qué es una clave en un ***Map***? La clave es asignada al elemento cuando se agrega a ***Map*** y puede ser de cualquier tipo primitivo o un objeto. La clave se usa para recuperar elementos de **Map** y actúa como una referencia al valor, por lo que **no puede estar duplicada**.

Como es una interfaz, necesita una implementación, aquí tenemos las 3 implementaciones más comunes para la interfaz ***Map***:

- **HashMap**: Es como tener parejas de datos en un saco y se recupera el dato por el valor de la key. Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones.
- **LinkedHashMap**: Esta implementación almacena las claves en función del orden de inserción.
- **TreeMap**: Los elementos se almacenan distribuidos como una jerarquía de árbol, es decir, esta implementación almacena los elementos ordenándolos en función de sus valores. Las claves almacenadas deben implementar la interfaz Comparable.

Declaración:

Map <tipoClave, tipoValor> nombreMap = new tipoclase<>();

Para nuestro ejemplo: `Map <Integer, String> listadoAlumnos = new HashMap<>();`

Aplicación en Java:

Para nuestro ejemplo, el siguiente programa usando la interfaz ***Map*** no nos permite repetir las claves (1,2,3) pero sí los valores (hay dos alumnos con nombre Juan). En este caso, para pedir el nombre del alumno número 2, usamos el método get y especificamos la clave (2). Entonces la consola nos devuelve el valor "Juan".

```
public class Colegio{
	public static void main (String[] args) {
		Map <Integer, String> listaAlumnos = new HashMap<>();
		listaAlumnos.add(1,"Juan");
		listaAlumnos.add(2,"Juan");
		listaAlumnos.add(3,"Laura");
		String valor = listaAlumnos.get(2);
		System.out.printIn("Alumno asociado a 2 es" + valor);
		//Mostraría en pantalla "Alumno asociado a 2 es Juan"
	}
}
```

¿CÓMO ORDENAR COLECCIONES?

Java nos permite crear un criterio para ordenar los elementos de una colección o un array.

Se pueden ordenar fácilmente cuando se trata de número o palabras siguiendo el orden numérico o el orden del alfabeto, sin embargo, cuando se trata de objetos (Personas, Coches, etc.), no hay un orden natural definido, para ello podemos utilizar dos interfaces que se usan para comparar o para definir el orden natural de un objeto para su comparación:

- **Interfaz Comparable:** permite implementar un criterio de orden natural para los objetos a ordenar, implementando el método **compareTo().**

- **Interfaz Comparator:** permite usar más de un criterio para ordenar los elementos, implementando el método **compare().**

¿Sabías que para ordenar una colección en Java disponemos, en la clase *java.util.Collections,* del método sort(List) que permite ordenar cualquier colección que implemente List (Collections.sort(list))?, pero para que este método compile y funcione, el elemento de la colección debe tener definido un orden natural.

INTERFAZ ***COMPARABLE***

Esta interfaz se encuentra en el paquete java.Lang y puede ser implementada por cualquier clase en la que se quiera definir un criterio de orden natural a través de su método abstracto ***compareTo***.
Si se crea una clase que implemente de esta interface, deberá sobrescribir el método compareTo(Object). Este método compara el objeto que llama con el objeto especificado como parámetro, el cual permite ordenar un objeto según un atributo especificado (String, double, int, etc.), en un orden ascendente o descendente. Este método debe retornar un valor entero que debe interpretarse de la siguiente forma:

- Si retorna un **cero** quiere decir que son **iguales**, es decir, el valor de la instancia es igual al valor del parámetro Object.
- Si retorna un **valor negativo** quiere decir que es **menor**, es decir, el valor de la instancia es menor que el valor del parámetro Object.
- Si retorna un **valor positivo** quiere decir que es **mayor**, es decir, el valor de la instancia es mayor que el valor del parámetro Object.

Ejemplo.

1. Imaginemos que tenemos la clase persona con sus atributos edad y nombre. Y nosotros queremos ordenar nuestra lista por edad.

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

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

2. Tenemos que implementar la interfaz **Comparable** sobre la clase Persona, para lo que tendremos que sobrescribir el método **`public int compareTo(Persona p)`** e indicar ordenación por edad. Este método nos devolverá diferente valor al comparar cada objeto de la lista con el objeto de referencia, que en nuestro caso hemos llamado "Persona p".

```
public class Persona implements Comparable<Persona> {
       private int edad;
       private String nombre;

       public Persona (int edad, String nombre){
           this.edad = edad
           this.nombre = nombre;
       }
       //Ordenar por edad.
       @Override
       public int compareTo(Persona p){
           return this.getEdad()-p.getEdad();
       }
}
```

3. Finalmente creamos nuestra clase con la lista de personas que queremos ordenar por edad. Para la interfaz ***Comparable*** se usa la clase **ArrayList**, y el método **Collections.sort** que ordena los elementos según los criterios del compareTo definido en la clase Persona.

```
public class ListaPersonas{
       public static void main (String[] args) {
              Persona p1 = new Persona (20, "Pedro");
              Persona p2 = new Persona (10, "María");
              Persona p3 = new Persona (50, "Ramón");
              Persona p4 = new Persona (30, "Sofía");

              List<Persona> listaPersonas = new ArrayList<>();
              listaPersonas.add(p1);
              listaPersonas.add(p2);
              listaPersonas.add(p3);
              listaPersonas.add(p4);

              Collections.sort(listaPersonas);
              for (Personas p: listaPersonas){
                     System.out.println(p);
              }
       }
}
```

De esta manera estaremos pidiendo a la consola nos devuelve la **lista de personas ordenada de menor a mayor edad:**

Consola:

Persona [edad=10; nombre="María"]

Persona [edad=20; nombre="Pedro"]

Persona [edad=30; nombre="Sofía"]

Persona [edad=50; nombre="Ramón"]

INTERFAZ ***COMPARATOR***

Esta interfaz se encuentra en el paquete java.Lang y puede ser implementada por cualquier clase en la que se quiera **definir el criterio de orden natural** a través de su método abstracto ***compare***. Mientras que la interfaz ***Comparable*** nos obliga a implementar el método ***compareTo*** para indicar el orden natural de los elementos de esa clase, la interfaz ***Comparator*** nos obliga a implementar el método ***compare*** para indicar o definir el orden natural de los elementos.
Para el uso de esta interfaz es necesario **crear una clase nueva** que implemente dicha interfaz y sobrescriba el método ***compare***, de tal forma que la clase original no se ve afectada como en el caso del uso de la Interfaz ***Comparable*** que sí afecta a la clase original.
Si se crea una clase que implemente de esta interface, deberá sobrescribir el método compare(Object o1, Object o2). Este método compara dos objetos pasados como parámetros, en un orden ascendente o descendente. Este método debe retornar un valor entero para todos los casos, de la siguiente forma:

- Si retorna un **cero** quiere decir que son **iguales**, es decir, el valor de las dos instancias pasadas como parámetro son iguales.
- Si retorna un **valor negativo** quiere decir que es **menor**, es decir, el valor de la primera instancia es menor que el valor de la segunda instancia pasada como parámetro.
- Si retorna un **valor positivo** quiere decir que es **mayor**, es decir, el valor de la primera instancia es mayor que el valor de la segunda instancia pasada como parámetro.

Ejemplo.

1. Siguiendo nuestro ejemplo anterior, tenemos la clase original Persona. Nosotros crearemos una clase nueva en la que implementaremos la interfaz ***Comparator***, y en este caso indicaremos que queremos ordenar las personas por edad, pero de mayor a menor. Para ello sobrescribimos el método **`public int compare(Persona p1, Persona p2){`**

   La clase original de la que partimos sería:

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

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

2. Creamos una clase nueva para que ordene la clase original Persona (implementamos la **interfaz Comparator).** Y sobrescribimos el **método compare()** para ordenar por edad, pero en este caso de mayor a menor.

```
public class OrdenarEdad implements Comparator<Persona> {
	//Ordenar por edad
	@Override
	public int compare(Persona p1, Persona p2){
		return p2.getEdad()-p1.getEdad();
	}
}
```

3. Finalmente creamos nuestra clase igual que hacíamos en el ejemplo anterior, pero en este caso al método **Collection.sort** se le pasa un segundo parámetro para indicarle que tipo de criterio se quiere aplicar.

```
public class ListaPersonas{
	public static void main (String[] args) {
		Persona p1 = new Persona (20,"Pedro");
		Persona p2 = new Persona (10,"María");
		Persona p3 = new Persona (50,"Ramón");
		Persona p4 = new Persona (30,"Sofía");

		List<Persona> listaPersonas = new ArrayList<>();
		listaPersonas.add(p1);
		listaPersonas.add(p2);
		listaPersonas.add(p3);
		listaPersonas.add(p4);

		Collections.sort(listaPersonas, new OrdenarEdad());
		for(Personas p: listaPersonas){
			System.out.printIn(p);
		}
	}
}
```

De esta manera estaremos pidiendo a la consola nos devuelve la **lista de personas ordenadas** según el método compare() de la clase que hemos creado "OrdenarEdad" que ordena **por edad de mayor a menor.**

Consola:

Persona [edad=50; nombre="Ramón"]

Persona [edad=30; nombre="Sofía"]

Persona [edad=20; nombre="Pedro"]

Persona [edad=10; nombre="María"]

3.4. ERRORES Y EXCEPCIONES

¿QUÉ SON LOS ERRORES EN JAVA?

Los errores son problemas críticos causados por un **código incorrecto o mal diseñado** que hacen que las **aplicaciones fallen y se detengan**. En Java **los errores en tiempo de ejecución** se denominan **excepciones** y ocurren cuando se produce un error en alguna de las instrucciones de nuestro programa, es decir, cada excepción es un objeto que define una condición anormal que interrumpe el flujo de un programa.

En Java (al igual que en otros lenguajes de programación), existen muchos tipos de excepciones y enumerar cada una de ellas sería casi una labor infinita. En lo referente a las excepciones hay que decir que se aprenden progresivamente a medida que nos encontramos con ellas y aprendemos a solucionarlas.

¿QUÉ TIPOS DE ERRORES HAY EN JAVA?

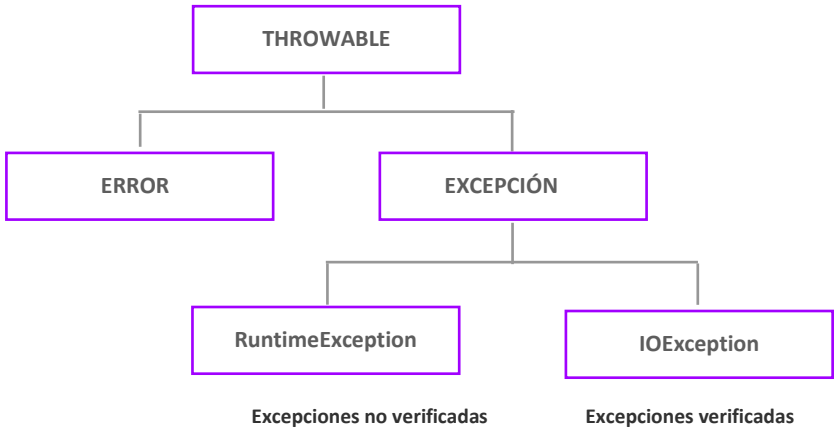
Al programar, la API de Java nos puede proporcionar dos errores diferentes según el momento:

- → **En tiempo de compilación**: errores de sintaxis que cometemos nosotros al escribir el código (se nos olvide un punto y coma (;) o cerrar unas llaves {}, etc.)
- → **En tiempo de ejecución**: errores o excepciones porque la lógica no es correcta, es decir, una vez compilado se ejecuta y nos da error. Estos pueden ser de dos tipos:
  - ***ERROR***: errores graves en la máquina virtual de Java.
  - ***EXCEPTION***: condición excepcional que altera la ejecución de un programa (errores no críticos que pueden ser tratados y Java nos va a avisar para que los podamos arreglar).

A continuación, nos centraremos en los errores en tiempo de ejecución, y más en concreto en las excepciones.

ESQUEMA DE TIPOS DE ERRORES

Cuando en Java se produce un Error o Excepción, se crea un ***objeto "Throwable"*** que mantendrá la información sobre el fallo y nos proporcionará los métodos necesarios para arreglarlo.



Este objeto será de un tipo u otro según el tipo de fallo (**Error o Excepción**).

Dentro de las Excepciones (errores no críticos que podemos controlar) podremos encontrar dos subtipos: **RuntimeException** e **IOException**, que detallaremos en los próximos puntos.

DIFERENCIAS ENTRE ERROR Y EXCEPCIÓN

Cuando hablamos de un **Error** nos referimos a **problemas graves** en nuestra aplicación. Este problema no puede ser resuelto de ninguna manera por lo que el programa normalmente se detiene. Mientras una excepción y sus subclases indican situaciones en donde una aplicación debería tratar de resolver un "error" de manera transparente al usuario lanzando advertencias.

Al producirse un **error** o una **excepción** en Java, se crea un objeto de una determina clase (como hemos visto en el Esquema de tipos de errores), que mantendrá la información sobre el error o excepción producido y nos proporcionará los métodos necesarios para obtener dicha información.

Cuando el error o excepción hereda de la clase **ERROR**, normalmente quiere decir que ha ocurrido un error de *Hardware*, por ejemplo, si una estructura de control crea un bucle infinito, colapsando la pila de llamadas, entonces se genera un error en la consola llamado "StackOverflowError".

¿QUÉ SON LAS EXCEPCIONES?

Las excepciones son condiciones especiales que **cambian el flujo normal de la ejecución** del programa. Cuando un error hereda de la clase **Exception**, indican situaciones que una aplicación debería tratar de forma razonable. El manejo de excepciones en Java se centra en todos los objetos que heredan de la clase **Exception** o sus subclases.

Veamos cuales son estas excepciones:

EXCEPCIONES VERIFICADAS (CHECKED) - IOException

Errores de entrada y salida que suelen ser **ajenos al programador**, normalmente debidos a factores externos. Estos errores o excepciones heredan de la clase **IOException**.

Ejemplo: Imaginemos que en nuestro programa estamos usando un objeto de un fichero externo que importaremos, y otra persona mueve ese fichero de sitio. Nuestro programa no encontrará ese fichero al buscarlo y nos dará error.

EXCEPCIONES NO VERIFICADAS (UNCHECKED) - RuntimeException

Errores generalmente causados por **código de programa o lógica incorrectos**, como parámetros no válidos. Estos errores o excepciones heredan de la clase **RuntimeException**.

Ejemplo: Imaginemos que queremos acceder fuera de los límites de un array o algo tan simple como tratar de dividir por cero.

MANEJO DE EXCEPCIONES

Las excepciones se pueden manejar de varias maneras:

- **Capturar**: Se captura la excepción que se ha producido y se ejecutan las sentencias asignadas para esa excepción. Para capturar las excepciones usaremos el bloque **try... catch...finally**. Captura tanto las excepciones checked (comprobadas) como las unchecked (no comprobadas).

```
try{
//sentencias a ejecutar
//aquí tendremos las sentencias propensas a lanzar las excepciones.
}catch (Exception e){
//sentencias que se ejecutarán en caso de
//producirse alguna excepción
}finally{
//sentencias que se ejecutarán, tanto si se
//produce una excepción como sino
}
```

- **Propagar**: se produce una excepción, pero derivamos la gestión de la excepción a la pila de llamadas.

- **Lanzar**: se produce una condición determinada, y decidimos forzar una excepción para que sea gestionada usando la palabra reservada **throw new**. Como cualquier excepción, está en algún momento tendrá que ser capturada y gestionada.

```
Public void metodo (int argumento) throws IllegalArgumentException{
       If (argumento > 18) {
              this.argumento = argumento;

       }else {
              throw new IllegalArgumentException ("Argumento no válido");
       }
}
```

¿CÓMO CREAR UNA EXCEPCIÓN PROPIA?

A pesar de todas las excepciones que nos proporciona la API de Java, también podemos crear **excepciones personalizadas** para poder llegar a cubrir algún tipo de error personalizado por nosotros. Para esto, **debemos crearnos nuestra propia clase** que heredará de la clase IOException o RuntimeException según el tipo que sea.

Veamos un ejemplo: vamos a crear una excepción de tipo no verificada que informará al programador que nuestro programa nunca va a dividir entre 1.

Primero tenemos que crear la clase con la excepción que queremos usar, en nuestro caso le hemos llamado **ExcepcionDividirUno** que heredará de **RuntimeException** (no verificada).
A continuación, la excepción debe tener dos **constructores**, uno que no recibe argumento y otro que sí (recibe un argumento de tipo String, el mensaje del error para el programador). Para este último llamamos al método **super** para agregar el mensaje.

```
public class ExcepcionDividirUno extends RuntimeException{

     public ExcepcionDividirUno(){ //constructor sin argumentos
     }
     public ExcepcionDividirUno(String texto){ //constructor con argumentos
            super (texto);
     }

}
```

Por último, en nuestro programa (clase División) podremos hacer uso de esta excepción usando la palabra reservada **throws** para lanzar la excepción con dicha clase. Vamos a comprobar que funciona, creando una condición: "si b=1 lance la nueva excepción" usando las palabras reservadas **throw new** "si no, divida las cantidades".

Finalmente ejecutaremos el método main para resolver el programa.

```
public class Division{
       public void dividir(){  //método que divide
              divide (10,1);
       }
       public divide(int a, int b)throws ExceptionDividirUno{
              if (b==1){ //si b=1 lance la excepción
                     throw new ExceptionDividirUno("No voy a dividir entre uno");
              }else{ //si b es distinto de cero, que divida a/b
                     System.out.printIn (a/b);
              }
       }
       public static void main (String args[]){ //método main para ejecutar la división
              new Division().dividir();
       }
}
```

La consola nos devuelve: "No voy a dividir entre uno"
(Si los valores hubiesen sido (10, 2) la consola nos hubiese devuelto el valor 5).

Constructor: son un tipo de métodos especiales que inician los atributos de la clase cada vez que se crea un objeto con valores coherentes. Sirven para establecer el estado inicial de un objeto.

3.5. PREGUNTAS FRECUENTES

- **¿Cómo ordena la interfaz Comparable una colección de nombres por orden natural?**

  Ordenará los elementos por orden alfabético.

- **¿Qué parámetro puedo usar para identificar una clase genérica?**

  Debe de ser un solo carácter y en mayúscula (no puede ser un tipo primitivo, por ejemplo, un número). Según las convenciones los nombres de los parámetros usados comúnmente son los siguientes: T, E, U, V. Aunque podrías usar la letra que quieras, estas son las más utilizadas para identificar genéricos en programación.

FUNDACIÓN ACCENTURE

accenture