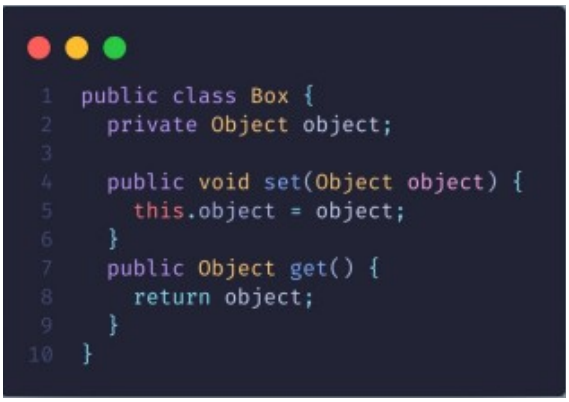
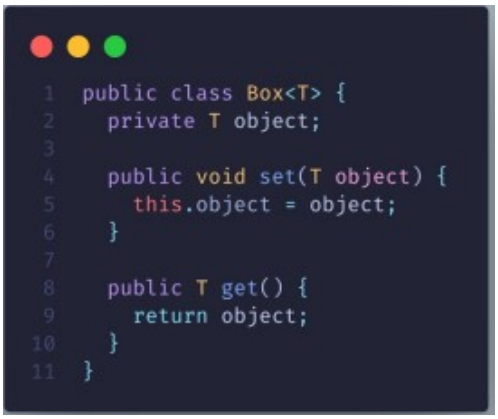
UD 01 - Streams, Ficheros y Expresiones Regulares

Clases genéricas

Se trata de una clase parametrizada sobre uno o más tipos. El tipo se asigna en tiempo de compilación



```
public class Box {
  private Object object;

  public void set(Object object) {
    this.object = object;
  }
  public Object get() {
    return object;
  }
}
```



```
public class Box<T> {
  private T object;

  public void set(T object) {
    this.object = object;
  }

  public T get() {
    return object;
  }
}
```

El código de la izquierda es más propenso a producir errores.

Otro ejemplo:

```
public class Par<T, S> {
  private T obj1;
  private S obj2;

  // Resto de la clase

}
```

Los nombres de tipos de parámetros más usados son:

- E (element, elemento)
- K (key, clave)
- N (number, número)
- T (type, tipo)
- V (value, valor)
- S, U, V, … (2º, 3º, 4º, … tipo)

Para instanciar un objeto genérico, tenemos que indicar los tipos dos veces.

```
Par<String, String> pareja2 = new Par<String, String>("Hola",
"Mundo");
```

Este estilo es muy verboso. Desde Java SE 7 tenemos el operador <>

```
Par<String, String> pareja2 = new Par<>("Hola", "Mundo");
```

Podemos indicar que el tipo parametrizado sea uno en particular (o sus derivados).

```
public class NumericBox<T extends Number>
public class StrangeBox <T extends A>
```

¿Qué es un tipo comodín <?> en Java?

Los tipos comodín (wildcard), aparecen con el concepto de programación genérica. En castellano también los puedes encontrar como tipo salvaje.

Imagina que tienes una caja. Esa caja puede contener diferentes tipos de objetos: manzanas, naranjas, libros, etc. Sin embargo, no sabemos exactamente qué tipo de objeto hay dentro hasta que abrimos la caja.

En Java, los tipos comodín son como esa caja. Representan un tipo desconocido, pero que cumple con ciertas restricciones. Se utilizan cuando no sabemos el tipo exacto de un objeto en tiempo de compilación, pero necesitamos realizar algunas operaciones con él.

¿Para qué sirven los tipos comodín?

Mayor flexibilidad: Permiten escribir código más genérico y reutilizable, ya que no están limitados a un tipo de dato específico.

Mayor seguridad: Ayudan a evitar errores de tiempo de ejecución relacionados con tipos incompatibles.

Colecciones: Son especialmente útiles al trabajar con colecciones como List, Set y Map, donde los elementos pueden ser de diferentes tipos.

```
List<?> myList; // Una lista que puede contener cualquier tipo de
objeto
```

En este ejemplo, myList puede contener una lista de números enteros, una lista de cadenas o incluso una lista de objetos de una clase personalizada. Sin embargo, no podemos agregar elementos directamente a myList porque no sabemos su tipo exacto.

Tipos de tipos comodín:

<?>: Tipo comodín sin restricciones. Puede representar cualquier tipo.

<? extends T>: Tipo comodín con límite superior. Solo puede representar tipos que sean subtipos de T.

<? super T>: Tipo comodín con límite inferior. Solo puede representar tipos que sean supertipos de T.

```
List<? extends Number> numbersList; // Solo puede contener números
List<? super Integer> integerList; // Puede contener Integer y sus
supertipos (como Number o Object)
```

Comparable y Comparator

Comparable y Comparator son dos interfaces en Java utilizadas para comparar objetos. Aquí están las principales diferencias:

Comparable

- La interfaz Comparable se encuentra en el paquete java.lang y tiene un único método llamado compareTo().

- Si quieres que los objetos de una clase sean comparables por defecto, entonces esa clase debe implementar Comparable.

- El método compareTo() compara el objeto actual con el objeto especificado y devuelve un entero negativo, cero o un entero positivo si el objeto actual es menor que, igual a, o mayor que el objeto especificado.

- Comparable es útil cuando tienes una única fuente de ordenamiento.

Comparator

- La interfaz Comparator se encuentra en el paquete java.util y tiene dos métodos: compare() y equals().

- Puedes crear una clase separada que implemente “Comparator”, y luego usar un objeto de esa clase para comparar objetos de otra clase, lo que significa que no necesitas modificar la clase que quieres comparar.

- El método compare() compara sus dos argumentos y devuelve un entero negativo, cero o un entero positivo si el primer argumento es menor que, igual a, o mayor que el segundo.

- Comparator es útil cuando tienes múltiples fuentes de ordenamiento. Por ejemplo, puedes tener un Comparator que ordene por nombre y otro que ordene por edad.

En resumen, Comparable es para una ordenación "natural" y por defecto de los objetos de una clase, mientras que Comparator es para ordenaciones personalizadas.

Otra explicación:

Muchas operaciones entre objetos nos obligan a compararlos: buscar, ordenar, … Los tipos primitivos y algunas clases ya implementan su orden (natural, lexicográfico). Para nuestras clases (modelo) tenemos que especificar el orden con el que las vamos a tratar

Comparable:

Se trata de un interfaz sencillo. Recibe un objeto del mismo tipo. Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor. Nos sirve para indicar el orden principal de una clase.

```
public interface Compararable<T> {
// 0 si es igual, 1 si es mayor, -1 si es menor.
public int compareTo(T o);
}
```

Comparator:

Se trata de un interfaz sencillo. Recibe dos argumentos. Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor. Nos sirve para indicar un orden puntual, diferente al orden principal de una clase.

```
public interface Comparator<T> {
public int compare(T o1, T o2);
}
```

Ejemplos:

Diferencia entre Comparable y Comparator en Java.

Comparable

```
import java.util.*;

class Student implements Comparable<Student> {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int compareTo(Student s) {
        return this.id - s.id;
    }

    public String toString() {
        return this.id + " " + this.name;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Alice"));
        list.add(new Student(1, "Bob"));
        list.add(new Student(2, "Charlie"));

        Collections.sort(list);

        System.out.println(list);
    }
}
```

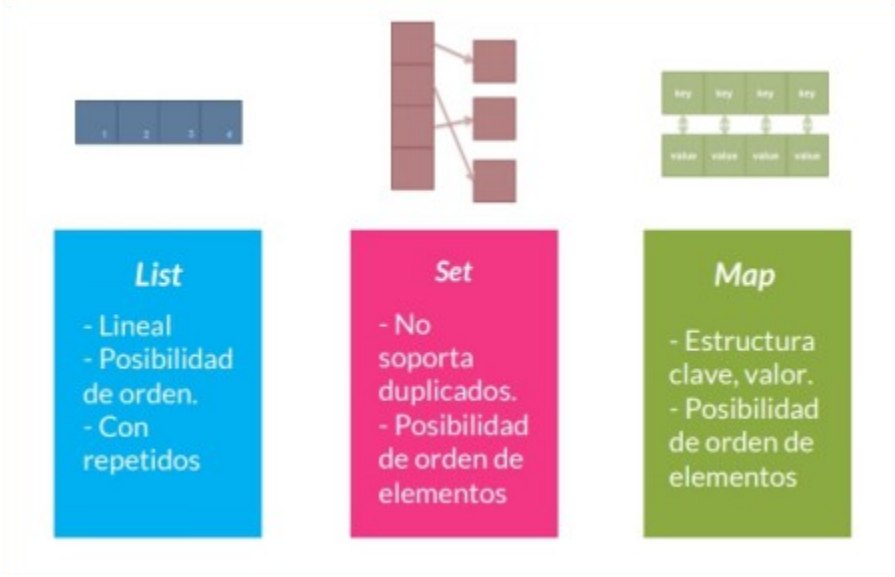
En este ejemplo, la clase Student implementa Comparable<Student>. Cuando llamamos a Collections.sort(list), los objetos Student en la lista se ordenan por su id porque eso es lo que especificamos en el método compareTo().

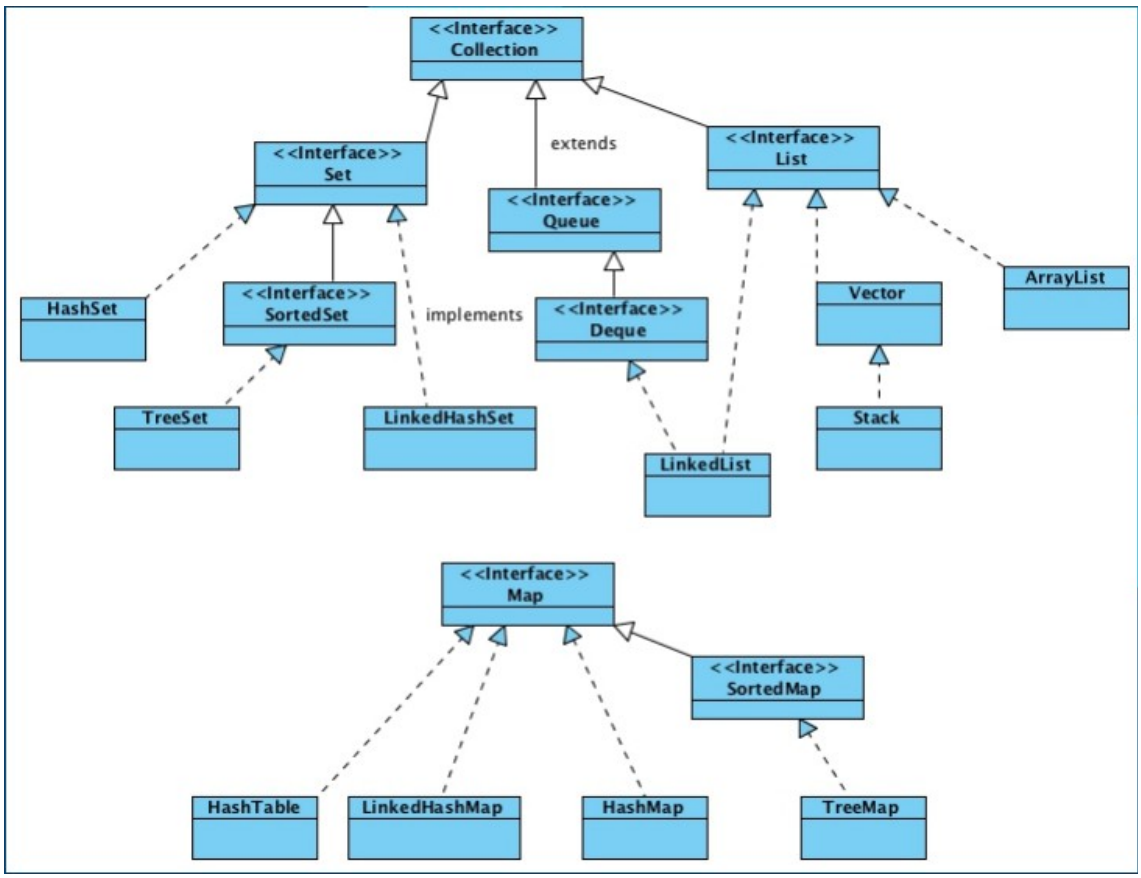
Comparator

```
import java.util.*;

class Student {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public String toString() {
        return this.id + " " + this.name;
    }
}

class StudentNameComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Alice"));
        list.add(new Student(1, "Bob"));
        list.add(new Student(2, "Charlie"));

        Collections.sort(list, new StudentNameComparator());

        System.out.println(list);
    }
}
```

En este ejemplo, la clase Student no implementa Comparable<Student>. En su lugar, creamos una clase separada StudentNameComparator que implementa Comparator<Student>. Cuando llamamos a Collections.sort(list, new StudentNameComparator()), los objetos Student en la lista se ordenan por su name porque eso es lo que especificamos en el método compare().

Colecciones





Ejercicios Básicos:

1 List

Ej1. Crear una lista de Strings y añadir elementos a ella. Luego, imprimir todos los elementos de la lista usando un bucle for-each.

Ej2. Dada una lista de números enteros, escribir una función que devuelva una nueva lista que contenga solo los números pares de la lista original.

Ej3. Dada una lista de Strings, escribir una función que devuelva la longitud del string más largo en la lista.

2 Set

Ej1. Crear un Set de Strings y añadir elementos a él. Luego, imprimir todos los elementos del Set. ¿Qué observas acerca del orden de los elementos?

Ej2. Dada una lista de números enteros, escribir una función que devuelva un Set que contenga solo los números únicos de la lista original.

Ej3. Dada una lista de Strings, escribir una función que devuelva un Set que contenga solo los Strings únicos de la lista original.

3 Map

Ej1. Crear un Map que asocie nombres de países con sus capitales. Luego, imprimir todos los pares de clave-valor del Map.

Ej2. Dada una lista de Strings, escribir una función que devuelva un Map donde las claves son los Strings únicos de la lista y los valores son el número de veces que cada String aparece en la lista.

Ej3. Dada una lista de estudiantes (donde cada estudiante es un objeto con propiedades como nombre, edad, grado, etc.), escribir una función que devuelva un Map donde las claves son los nombres de los estudiantes y los valores son los objetos de los estudiantes.

Funciones Lambda

Trabajando funcionalmente y de forma fluida.

Antes de nada debemos introducir la programación funcional en Java

En la programación Estructurada nos centramos en el QUÉ y el CÓMO, en la funcional nos centramos en el QUÉ

Expresiones Lambda: Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase. Su sintaxis básica se detalla a continuación:

(parámetros) -> { cuerpo-lambda}

El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.

Parámetros:

- ✓ Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
- ✓ Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.

Cuerpo de lambda:

- ✓ Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la clausula return en el caso de que deban devolver valores.
- ✓ Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la clausula return en el caso de que la función deba devolver un valor.

Ejemplos:

Ejemplo 1: Ordenar una lista de Strings por longitud

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Dog", "Elephant", "Cat",
"Rabbit", "Butterfly");

        Collections.sort(list, (s1, s2) -> s1.length() - s2.length());

        System.out.println(list);
    }
}
```

En este ejemplo, la función lambda (s1, s2) -> s1.length() - s2.length() se utiliza como argumento para el método Collections.sort(). Esta función lambda compara dos strings por su longitud.

Ejemplo 2: utiliza una función lambda para implementar una interfaz funcional personalizada:

```
public class Main {
    interface Greeting {
        void sayHello(String name);
    }

    public static void main(String[] args) {
        Greeting greeting = (name) -> System.out.println("Hello, " +
name);
        greeting.sayHello("Alice");
    }
}
```

En este ejemplo, definimos una interfaz funcional “Greeting” con un método “sayHello()”. Luego, en el método “main()”, creamos una instancia de “Greeting” utilizando una función lambda “(name) -> System.out.println("Hello, " + name)”. Esta función lambda toma un nombre e imprime un saludo personalizado.

Ejercicios:

1. Calculadora con funciones lambda: Crea una interfaz funcional “Calculator” con un método “calculate()”. Este método debe tomar dos números enteros y devolver un número entero. Luego, en tu método “main()”, crea varias instancias de “Calculator” utilizando funciones lambda para implementar operaciones como suma, resta, multiplicación y división. Finalmente, prueba tus calculadoras con algunos números.

2. Filtrado de lista con funciones lambda: Crea una lista de Strings que contenga varios nombres. Luego, utiliza una función lambda para filtrar esta lista y crear una nueva lista que solo contenga los nombres que comienzan con una letra específica (por ejemplo, "A"). Para hacer esto, puedes utilizar el método “removeIf()” de “ArrayList”, que toma un “Predicate” (que es una interfaz funcional que puedes implementar con una función lambda).