

Polimorfismo via Interfaces

Programación Orientada a Objetos (CC2008) - 2026

Polimorfismo via Interfaces

Semestre 02, 2026

Interfaz

Una interfaz es un tipo que declara un conjunto de métodos sin implementarlos. Funciona como un contrato, y toda clase que la implementa se compromete a definir esos métodos.

```
public interface IVolador {  
    void volar();  
}
```

Características:

- Sus métodos son implícitamente `public` y `abstract`.
- No tiene atributos de instancia, solo constantes `public static final`.
- No tiene constructores y no se puede instanciar con `new`.
- Una clase implementa una interfaz con la palabra clave `implements`.
- Una clase puede implementar varias interfaces, aunque solo extiende una superclase.
- Desde Java 8 admite métodos `default` y `static` con cuerpo.

La herencia expresa una relación es-un. Una interfaz expresa una capacidad, una relación puede-hacer.

Tipo y habilidad

La lección anterior cerró con el límite de la herencia. Gengar es de tipo fantasma y veneno, y Java no permite que una clase extienda dos superclases.

```
class Gengar extends Pokemon implements IFantasma, IVeneno { }
```

Gengar hereda de `Pokemon` lo que todo Pokémon comparte, y sus dos tipos se suman como interfaces. Dragonite, de tipo dragón y volador, se resuelve con la misma técnica.

La superclase Pokemon

```
public abstract class Pokemon {
    protected String nombre;
    protected int nivel;
    protected int hp;

    public Pokemon(String nombre, int nivel, int hp) {
        this.nombre = nombre;
        this.nivel = nivel;
        this.hp = hp;
    }

    public abstract void atacar();

    public void mostrarFicha() {
        System.out.println(this.nombre + " | Nv. " + this.nivel
            + " | HP " + this.hp);
    }
}
```

```
}
```

Es la clase abstracta de la lección anterior. `PokemonElectrico` y `PokemonAgua` la extienden y definen su propio `atacar`.

Enunciado

Se extenderá la Pokédex con Pokémon de dos tipos. Dragonite es de tipo dragón y volador, y Gyarados es de tipo agua y volador. El juego necesita un movimiento de vuelo que funcione con cualquier Pokémon capaz de volar.

- Un Pokémon que no vuela, como Pikachu, no debe poder usar ese movimiento, y el error debe detectarse al compilar.
- El recorrido de los voladores no debe consultar la especie de cada Pokémon.

Restricciones de diseño:

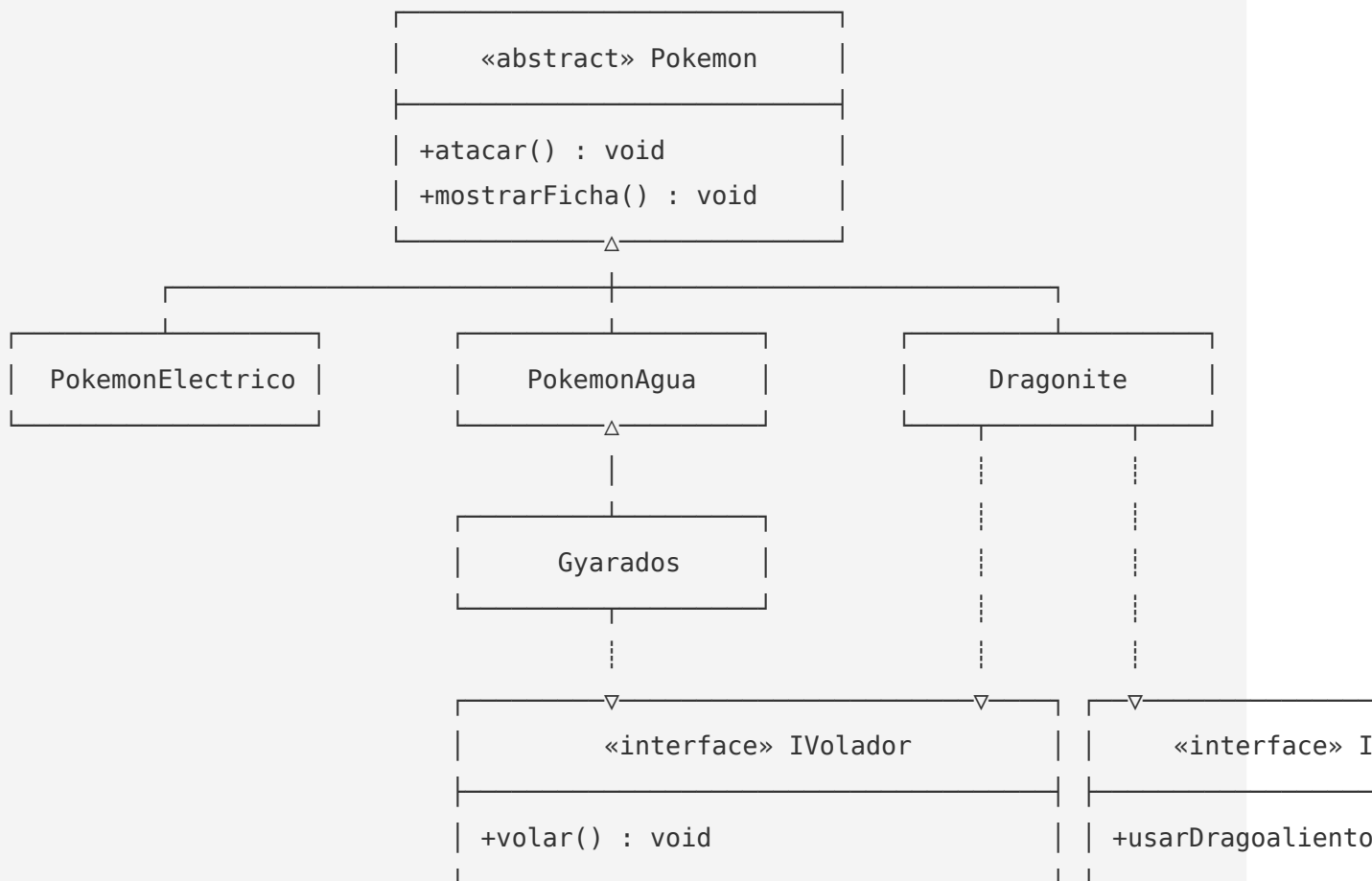
- Dragonite necesita dos tipos y Java permite extender una sola clase.
- Gyarados ya extiende a `PokemonAgua`, así que el vuelo no puede venir de otra superclase.
- El recorrido no pregunta si cada elemento es un Dragonite o un Gyarados.

Dragonite y Gyarados pertenecen a ramas distintas de la jerarquía, pero ambos vuelan. Esa capacidad común se modela con la interfaz `IVolador`.

Las interfaces `IVolador` e `IDragon`

```
public interface IVolador {  
    void volar();  
}
```

```
public interface IDragon {
    void usarDragoaliento();
}
```



Lectura del diagrama:

- La etiqueta **«interface»** marca el contrato.
- La línea punteada con triángulo indica implementación, y se lee como implementa a.
- La línea continua con triángulo indica herencia, igual que en la lección anterior.
- **IVolador** atraviesa la jerarquía. La implementan una subclase directa de **Pokemon** y una subclase de **PokemonAgua**.

Dragonite

```
public class Dragonite extends Pokemon
    implements IDragon, IVolador {

    public Dragonite(String nombre, int nivel, int hp) {
        super(nombre, nivel, hp);
    }

    @Override
    public void atacar() {
        System.out.println(this.nombre + " usa Hiperrayo.");
    }

    @Override
    public void usarDragoaliento() {
        System.out.println(this.nombre + " usa Dragoaliento.");
    }

    @Override
    public void volar() {
        System.out.println(this.nombre + " vuela sobre el océano.");
    }
}
```

Decisiones de diseño:

- Dragonite extiende una sola clase e implementa dos interfaces, separadas por coma.
- Debe definir `atacar`, que exige `Pokemon`, y los métodos de las dos interfaces, o no compila.
- `@Override` también aplica a los métodos que provienen de una interfaz.
- Los métodos se declaran `public`, porque un método de interfaz es público y la implementación no puede reducir su visibilidad.

Gyarados

```
public class Gyarados extends PokemonAgua implements IVolador {

    public Gyarados(String nombre, int nivel, int hp, int presion) {
        super(nombre, nivel, hp, presion);
    }

    @Override
    public void volar() {
        System.out.println(this.nombre + " se eleva sobre las olas.");
    }
}
```

Gyarados hereda `atacar` y `bucear` de `PokemonAgua` y solo agrega `volar`. Un Gyarados es un `PokemonAgua` y además cumple el contrato `IVolador`.

`PokemonAgua` no cambió ni una línea. Los demás Pokémon de agua siguen sin poder volar.

Los voladores

```
ArrayList<IVolador> voladores =
    new ArrayList<>();

voladores.add(new Dragonite(
    "Dragonite", 55, 91));
voladores.add(new Gyarados(
    "Gyarados", 40, 95, 120));

for (IVolador v : voladores) {
    v.volar();
}
```

```
// no compila, Pikachu no vuela
voladores.add(new PokemonElectrico(
    "Pikachu", 12, 35, 90));
```

La lista se declara con el tipo de la interfaz y acepta cualquier objeto cuya clase la implemente.

Dragonite y Gyarados conviven en la misma lista aunque sus superclases directas son distintas.

El `for` declara la variable como `IVolador`. Desde esa referencia solo son visibles los métodos del contrato, así que `v.atacar()` no compila.

Cada llamada a `volar` se resuelve en tiempo de ejecución según el objeto real, igual que con herencia.

`PokemonElectrico` no implementa `IVolador`, y el compilador rechaza agregar a Pikachu.

Salida:

```
Dragonite vuela sobre el océano.
Gyarados se eleva sobre las olas.
```

Métodos que reciben una interfaz

```
public static void usarVuelo(IVolador v, String destino) {
    v.volar();
    System.out.println("Llega a " + destino + ".");
}
```

```
usarVuelo(dragonite, "Ciudad Celeste");
usarVuelo(gyarados, "Isla Canela");
```

```
usarVuelo(pikachu, "Pueblo Paleta"); // no compila
```

El método acepta cualquier Pokémon que implemente `IVolador`, incluido uno que todavía no existe. Un Charizard que implemente la interfaz funciona sin modificar `usarVuelo`.

Clase abstracta e interfaz

Aspecto	Clase abstracta	Interfaz
Palabra clave	<code>extends</code>	<code>implements</code>
Cuántas por clase	una	varias
Atributos de instancia	sí	no, solo constantes
Constructores	sí	no
Métodos con cuerpo	sí	solo <code>default</code> y <code>static</code>
Relación que expresa	es-un	puede-hacer

`Pokemon` reúne el estado y el comportamiento que comparte toda la jerarquía, como el nombre, el nivel y la ficha. `IVolador` declara una capacidad que tienen solo algunos Pokémon, repartidos en ramas distintas.

Interfaces de la biblioteca estándar

Java define interfaces que sus propias clases y métodos utilizan de forma polimórfica. Una clase nueva que las implementa se integra con esas herramientas sin modificarlas.

- `Comparable<T>` : define el orden natural de los objetos de una clase.
- `Iterable<T>` : permite recorrer un objeto con el `for` mejorado.
- `Iterator<T>` : entrega los elementos de un recorrido uno a la vez.

La `T` es un parámetro de tipo. Indica con qué clase trabaja la interfaz, igual que en `ArrayList<Pokemon>` .

Comparable

Se solicita imprimir los Pokémon capturados ordenados de menor a mayor nivel.

```
Collections.sort(capturados); // no compila
```

`Collections.sort` no conoce la clase `Pokemon` ni sabe qué significa que un Pokémon vaya antes que otro. Solo ordena listas cuyos elementos implementen `Comparable` .

```
public interface Comparable<T> {  
    int compareTo(T otro);  
}
```

El contrato de `compareTo` :

- Negativo: `this` va antes que `otro` .
- Cero: ambos ocupan la misma posición en el orden.
- Positivo: `this` va después que `otro` .

El valor exacto no importa, solo su signo.

Pokemon implementa Comparable

```

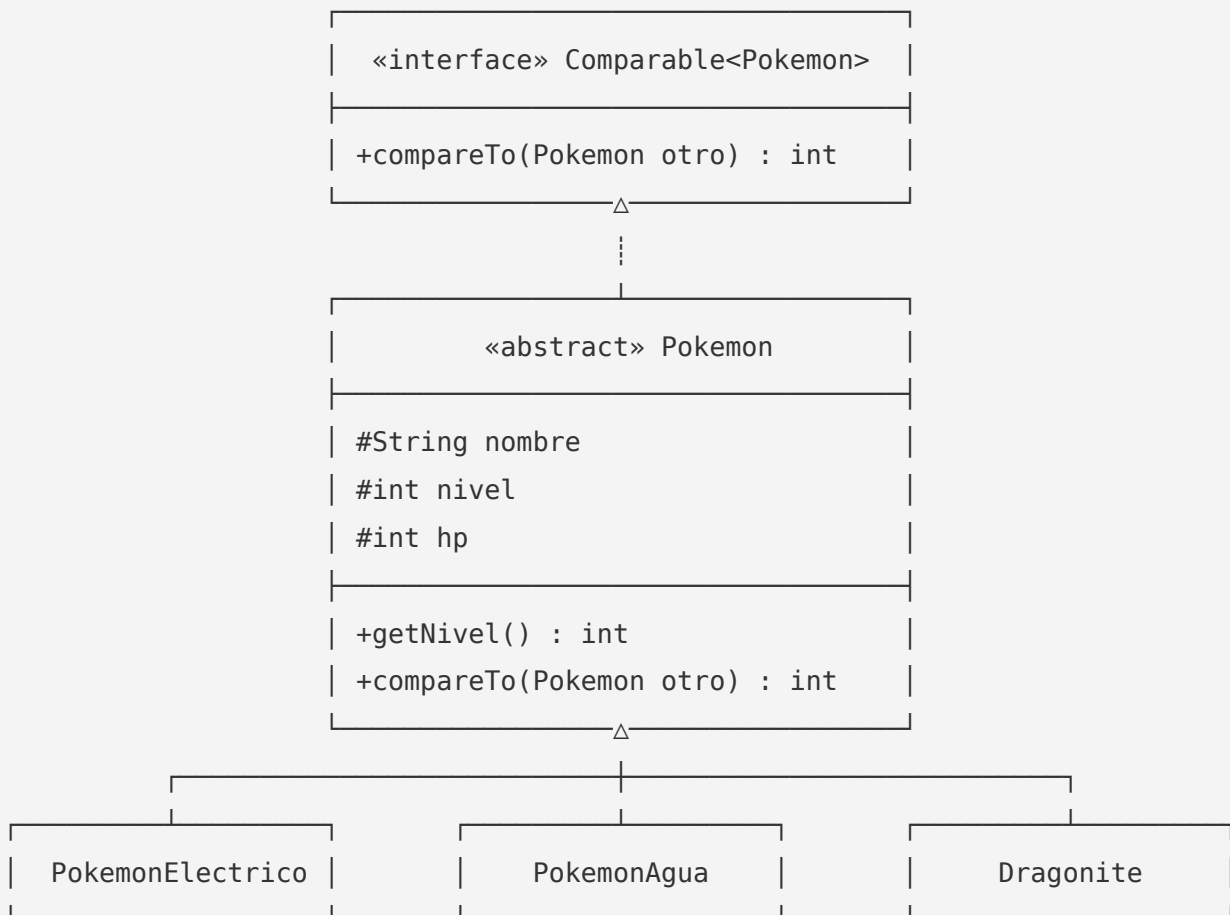
public abstract class Pokemon implements Comparable<Pokemon> {
    protected String nombre;
    protected int nivel;
    protected int hp;

    // constructor, atacar y mostrarFicha sin cambios

    public int getNivel() {
        return this.nivel;
    }

    @Override
    public int compareTo(Pokemon otro) {
        return Integer.compare(this.nivel, otro.getNivel());
    }
}

```



Decisiones de diseño:

- `Comparable<Pokemon>` indica que un Pokémon se compara con otro Pokémon, de modo que `compareTo` recibe un `Pokemon` y no un `Object`.
- La implementación vive en la superclase y todas las subclases la heredan, así que un Dragonite se compara con un Squirtle.
- `Integer.compare` devuelve el signo correcto sin restar.
- `compareTo` invoca `getNivel` del otro objeto en lugar de leer sus atributos.

Ordenar los capturados

```
ArrayList<Pokemon> capturados = new ArrayList<>();

capturados.add(new PokemonElectrico("Jolteon", 30, 65, 400));
capturados.add(new PokemonElectrico("Pikachu", 12, 35, 90));
capturados.add(new Dragonite("Dragonite", 55, 91));
capturados.add(new PokemonAgua("Squirtle", 10, 44, 60));

Collections.sort(capturados);

for (Pokemon p : capturados) {
    p.mostrarFicha();
}
```

Salida:

```
Squirtle | Nv. 10 | HP 44
Pikachu | Nv. 12 | HP 35
Jolteon | Nv. 30 | HP 65
Dragonite | Nv. 55 | HP 91
```

`Collections.sort` se escribió años antes que la clase `Pokemon`. Invoca `compareTo` a través de una referencia `Comparable`, y se ejecuta la versión que `Pokemon`

definió y que todas sus subclases heredaron.

Las clases `String`, `Integer` y `Double` ya implementan `Comparable`. Por eso una lista de nombres se ordena alfabéticamente con la misma llamada.

Desempate por nombre

Dos Pokémon del mismo nivel quedan en la posición que tenían antes de ordenar. Para un orden completo se agrega un segundo criterio.

```
public String getNombre() {
    return this.nombre;
}

@Override
public int compareTo(Pokemon otro) {
    int porNivel = Integer.compare(this.nivel,
                                    otro.getNivel());

    if (porNivel != 0) {
        return porNivel;
    }
    return this.nombre.compareTo(otro.getNombre());
}
```

El primer criterio decide mientras los niveles sean distintos. El segundo se consulta solo cuando el primero devuelve cero, y reutiliza el `compareTo` de `String`.

La resta en compareTo

```
@Override
public int compareTo(Pokemon otro) {
    return this.nivel - otro.getNivel();
}
```

```
}
```

Con enteros cercanos a los límites de `int` la resta se desborda, cambia de signo y el orden queda invertido.

Con atributos `double`, como un peso de 6.0 kg contra uno de 6.4 kg, la diferencia es -0.4 y la conversión a `int` la trunca a cero. Los dos Pokémon se consideran iguales aunque no lo son.

`Integer.compare` y `Double.compare` resuelven ambos casos.

Iterator e Iterable

Un equipo Pokémon admite como máximo seis integrantes. Se requiere una clase `Equipo` que los almacene en un arreglo privado de tamaño fijo, y el código cliente necesita recorrer el equipo para que cada integrante ataque.

El arreglo es `private`, así que el cliente no puede recorrerlo de forma directa. La primera alternativa es un método que devuelva el arreglo.

```
// dentro de Equipo
public Pokemon[] getMiembros() {
    return this.miembros;
}
```

```
// en el código cliente, con dos integrantes
Pokemon[] miembros = equipo.getMiembros();
for (int i = 0; i < miembros.length; i++) {
    miembros[i].atacar();    // falla en i = 2
}
miembros[0] = null;         // altera el equipo
```

Lo que cuesta exponer el arreglo:

- Las posiciones vacías son `null` , y recorrer hasta `length` lanza `NullPointerException` .
- El cliente recibe un alias del arreglo y puede modificar el equipo sin pasar por la clase.
- El cliente debe conocer la cantidad real de integrantes, que es un detalle interno.
- Si `Equipo` cambia el arreglo por un `ArrayList` , todo el código cliente deja de compilar.

Cada estructura se recorre de una forma distinta.

Estructura	Recorrido sin iterador
Arreglo	índice hasta la cantidad real
<code>ArrayList</code>	<code>get(i)</code> hasta <code>size()</code>
<code>HashSet</code>	no tiene índices
<code>Equipo</code>	depende de su implementación interna

Sin un mecanismo común, el código que recorre una colección queda atado a la forma en que esa colección guarda sus datos.

Iterador

Un iterador es un objeto que recorre una colección y entrega sus elementos uno a la vez, sin revelar cómo están almacenados.

1. La colección crea el iterador. `Equipo` conoce su arreglo y su cantidad, y se los entrega al iterador.

2. El cliente solo pregunta y pide. `hasNext` indica si quedan elementos y `next` entrega el siguiente. El cliente no ve índices, `null` ni el arreglo.

Lo que se gana:

- El arreglo sigue siendo `private` y nadie fuera de `Equipo` lo modifica.
- El iterador conoce la cantidad real y nunca entrega una posición vacía.
- `Equipo` puede cambiar su estructura interna sin romper el código cliente.
- `ArrayList`, `HashSet` y `Equipo` se recorren con las mismas dos llamadas.

El `for` mejorado usa ese protocolo por dentro, y por eso acepta cualquier objeto que implemente `Iterable`.

```
for (Pokemon p : equipo) {    // no compila
    p.atacar();
}
```

`Equipo` todavía no implementa `Iterable`. El `for` mejorado solo acepta arreglos y objetos que implementen esa interfaz.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}

public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

- `Iterable` : la colección que se puede recorrer. Su único método crea un iterador nuevo.
- `Iterator` : el recorrido en curso. Sabe en qué posición va y entrega el siguiente elemento.

- `hasNext` : indica si quedan elementos, sin avanzar.
- `next` : devuelve el siguiente elemento y avanza. Si ya no quedan lanza `NoSuchElementException` .

`Iterator` declara además `remove` , un método `default` que lanza `UnsupportedOperationException` si la clase no lo sobrescribe.

La clase Equipo

```
import java.util.Iterator;

public class Equipo implements Iterable<Pokemon> {
    private Pokemon[] miembros;
    private int cantidad;

    public Equipo() {
        this.miembros = new Pokemon[6];
        this.cantidad = 0;
    }

    public void agregar(Pokemon p) {
        this.miembros[this.cantidad] = p;
        this.cantidad++;
    }

    @Override
    public Iterator<Pokemon> iterator() {
        return new IteradorEquipo(this.miembros, this.cantidad);
    }
}
```

Decisiones de diseño:

- El arreglo y el contador son `private` y ninguna clase externa los ve.

- `iterator` crea un objeto nuevo en cada llamada, de modo que dos recorridos simultáneos no se interfieren.
- El iterador recibe una referencia al mismo arreglo, un alias, no una copia.
- El equipo no expone índices, así que el cliente no depende de que internamente se use un arreglo.

La clase `IteradorEquipo`

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class IteradorEquipo implements Iterator<Pokemon> {
    private Pokemon[] miembros;
    private int cantidad;
    private int actual;

    public IteradorEquipo(Pokemon[] miembros, int cantidad) {
        this.miembros = miembros;
        this.cantidad = cantidad;
        this.actual = 0;
    }

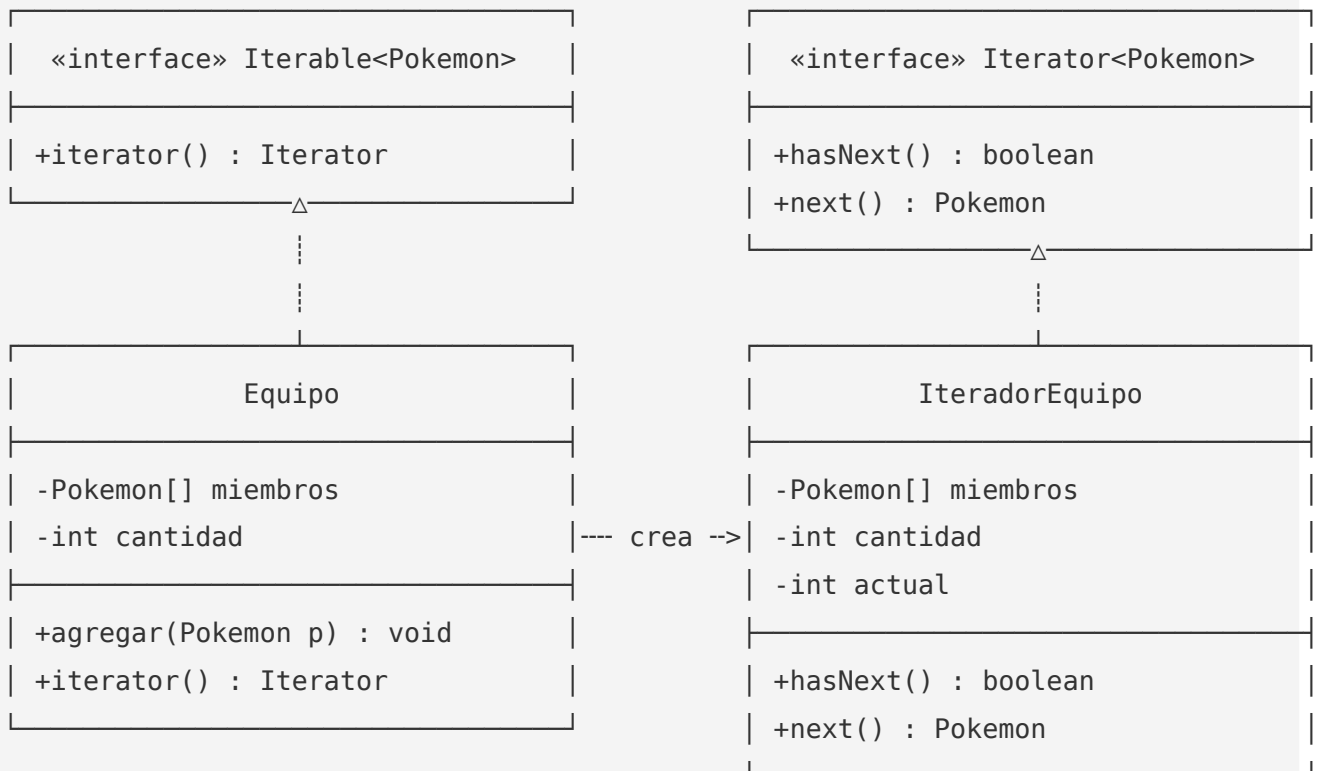
    @Override
    public boolean hasNext() {
        return this.actual < this.cantidad;
    }

    @Override
    public Pokemon next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Pokemon p = this.miembros[this.actual];
        this.actual++;
        return p;
    }
}
```

```

    }
}

```



Decisiones de diseño:

- `actual` guarda la posición del recorrido. Vive en el iterador y no en el equipo.
- `hasNext` solo consulta, nunca avanza, así que se puede llamar varias veces seguidas.
- `next` verifica antes de leer y lanza la excepción que exige el contrato.
- `remove` no se sobrescribe, de modo que el equipo no admite eliminar durante el recorrido.

El for mejorado por dentro

```
Equipo equipo = new Equipo();
equipo.agregar(new PokemonElectrico("Pikachu", 12, 35, 90));
equipo.agregar(new Dragonite("Dragonite", 55, 91));

for (Pokemon p : equipo) {
    p.atacar();
}
```

El compilador traduce el `for` mejorado a llamadas sobre la interfaz.

```
Iterator<Pokemon> it = equipo.iterator();
while (it.hasNext()) {
    Pokemon p = it.next();
    p.atacar();
}
```

- `equipo.iterator()` crea el iterador una sola vez, antes de la primera vuelta.
- `it.hasNext()` decide si el ciclo continúa.
- `it.next()` entrega el Pokémon y avanza la posición.
- El cuerpo del ciclo recibe cada Pokémon sin conocer el arreglo.

Salida:

```
Pikachu lanza 90 V.
Dragonite usa Hiperrayo.
```

El `for` mejorado depende solo de `Iterable` e `Iterator`. Funciona igual con `ArrayList`, `HashSet` o `Equipo`, porque las tres implementan la misma interfaz.

Métodos que reciben Iterable

```
public static double nivelPromedio(Iterable<Pokemon> grupo) {  
    int suma = 0;  
    int n = 0;  
    for (Pokemon p : grupo) {  
        suma += p.getNivel();  
        n++;  
    }  
    return (double) suma / n;  
}
```

```
nivelPromedio(equipo);          // un Equipo  
nivelPromedio(capturados);     // un ArrayList<Pokemon>
```

El método acepta cualquier colección de Pokémon que se pueda recorrer.

Eliminar durante un recorrido

Se solicita liberar a los Pokémon capturados con nivel menor que 15. La lista contiene a Pikachu, Jolteon, Squirtle y Dragonite, en ese orden.

```
for (Pokemon p : capturados) {  
    if (p.getNivel() < 15) {  
        capturados.remove(p);  
    }  
}
```

El programa lanza `ConcurrentModificationException`. El `for` mejorado usa un iterador por debajo, y la lista se modificó sin pasar por él. El iterador detecta el cambio en la siguiente llamada a `next`.

```
Iterator<Pokemon> it = capturados.iterator();  
while (it.hasNext()) {  
    Pokemon p = it.next();
```

```
    if (p.getNivel() < 15) {  
        it.remove();  
    }  
}
```

`it.remove()` elimina el último elemento entregado por `next` y mantiene al iterador sincronizado con la lista. Solo se puede invocar una vez por cada llamada a `next`. En la lista quedan Jolteon y Dragonite.

Errores comunes

Declarar con menor visibilidad un método de la interfaz. Escribir `void volar()` sin `public` en la clase no compila, porque el método de la interfaz es público.

Intentar instanciar una interfaz con `new IVolador()`. Una interfaz no tiene constructor.

Ordenar una lista cuyos elementos no implementan `Comparable`. `Collections.sort` sobre una `ArrayList<IVolador>` no compila.

Restar en `compareTo`. Con `double` la conversión trunca diferencias menores que uno, y con `int` el resultado puede desbordarse.

Avanzar dentro de `hasNext`. Si `hasNext` incrementa la posición, cada consulta salta un elemento.

Invocar `next` dos veces en la misma vuelta. Cada llamada avanza, así que el segundo valor corresponde a otro elemento y la última vuelta lanza `NoSuchElementException`.

Eliminar de la colección dentro de un `for` mejorado. Se usa `it.remove()` con un iterador explícito.

Buenas prácticas

Usar una interfaz para una capacidad que solo tienen algunas clases, o que atraviesa ramas distintas de la jerarquía.

Declarar variables, parámetros y colecciones con el tipo de la interfaz cuando solo se necesitan los métodos del contrato.

Implementar `compareTo` con `Integer.compare`, `Double.compare` o el `compareTo` de otra clase, nunca con una resta.

Guardar la posición del recorrido en el iterador, no en la colección.

Escribir `@Override` también en los métodos que implementan una interfaz.

Práctica

Sobre la Pokédex construida en clase, se solicita resolver los siguientes puntos sin modificar el recorrido de los voladores.

1. Implementar `Charizard`, que extiende a `PokemonFuego` e implementa `IVolador`, e incluirlo en la lista de voladores.
2. Declarar la interfaz `INadador`, implementarla en `PokemonAgua` y verificar que `Gyarados` también la cumple sin modificar su código.
3. Modificar `agregar` en `Equipo` para que lance una excepción cuando el equipo ya tenga seis integrantes.
4. Escribir un método estático que reciba un `Iterable<Pokemon>` y devuelva el Pokémon de mayor nivel, y probarlo con un `Equipo` y con un `ArrayList`.
5. Dibujar el diagrama de clases completo, con las interfaces, sus implementaciones y la jerarquía de `Pokemon`.