
Semana 16 de Septiembre

Colecciones

<recreo>

UML

Gabriela Pérez
gabriela.perez@lifa.info.unlp.edu.ar

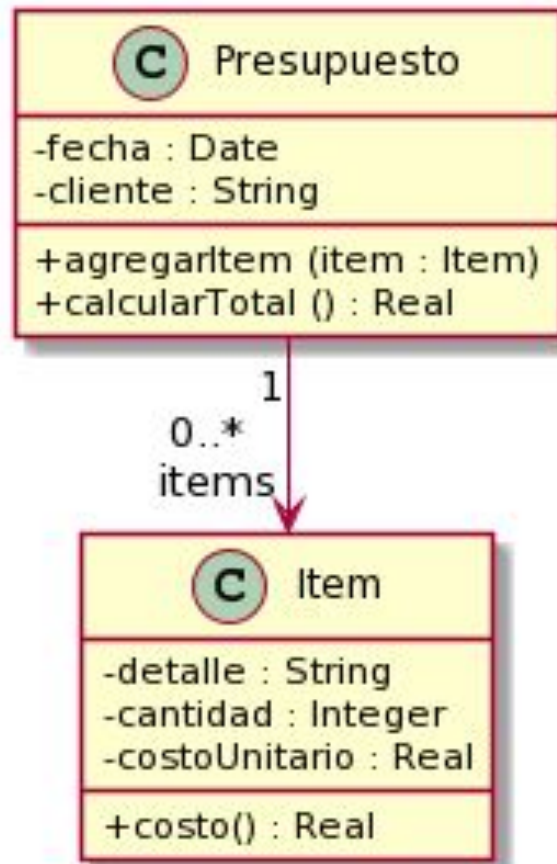


Colecciones

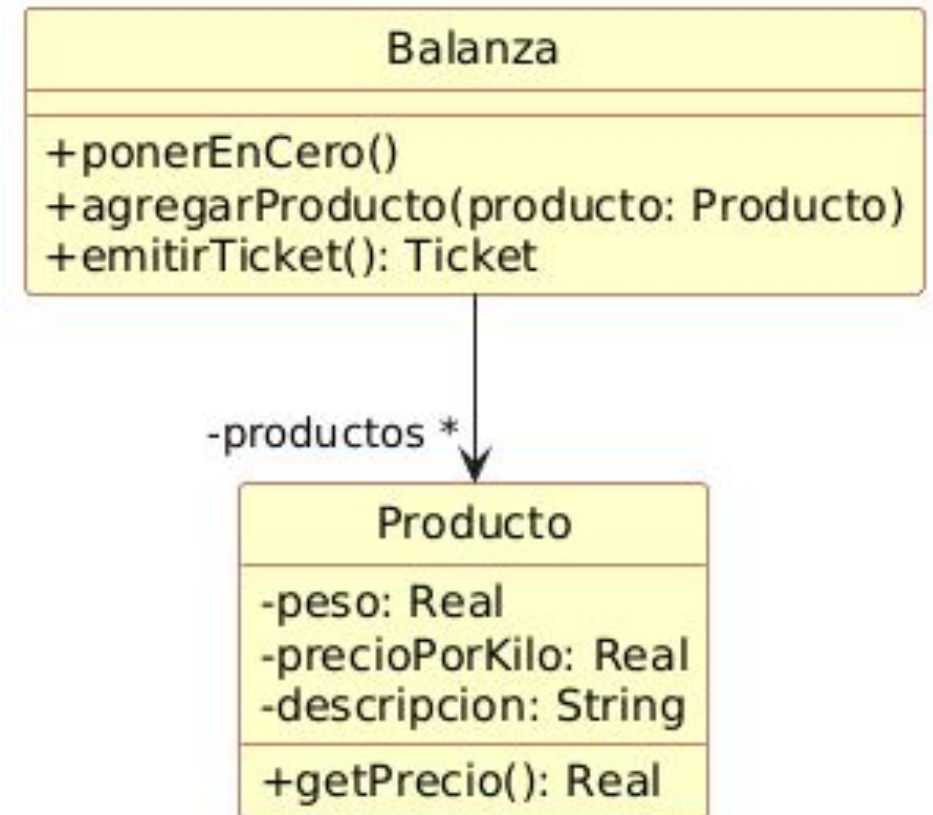


Ejercicios sobre Colecciones

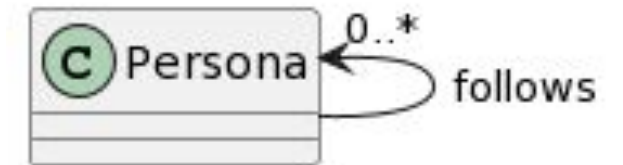
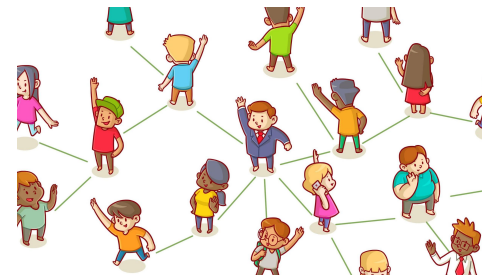
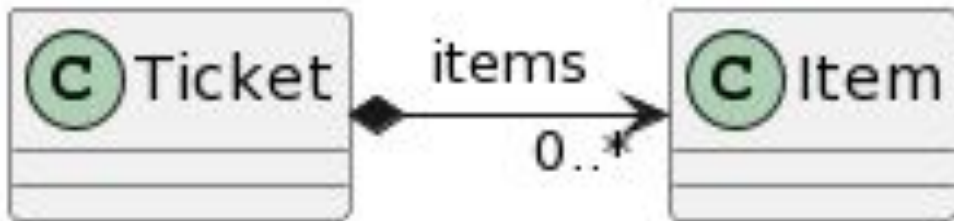
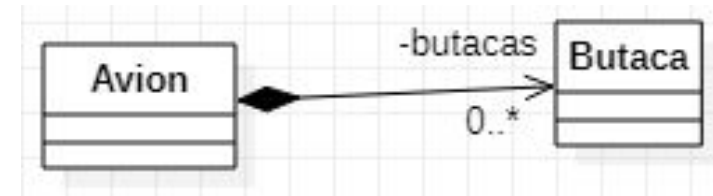
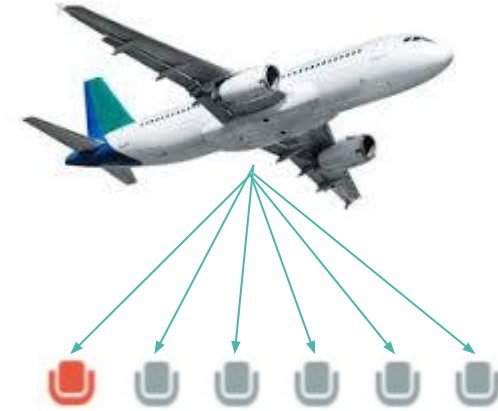
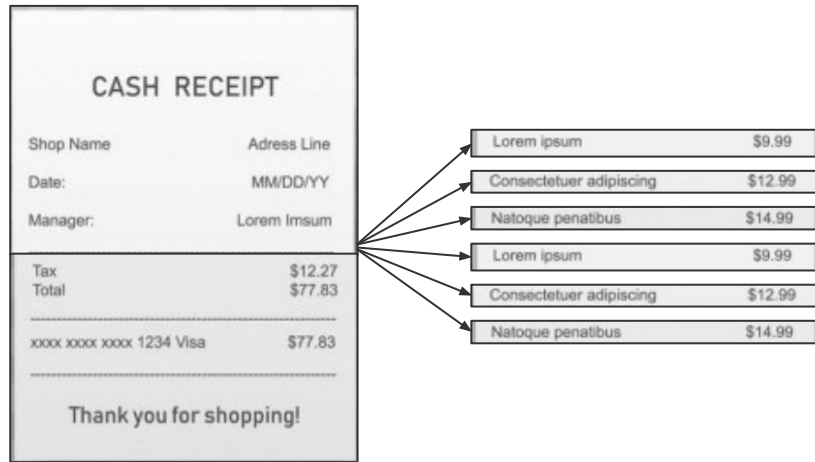
Ejercicio 3: Presupuestos



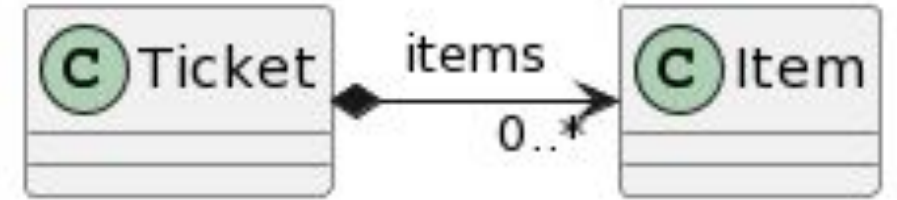
Ejercicio 4: Balanza mejorada



Relaciones 1 a muchos



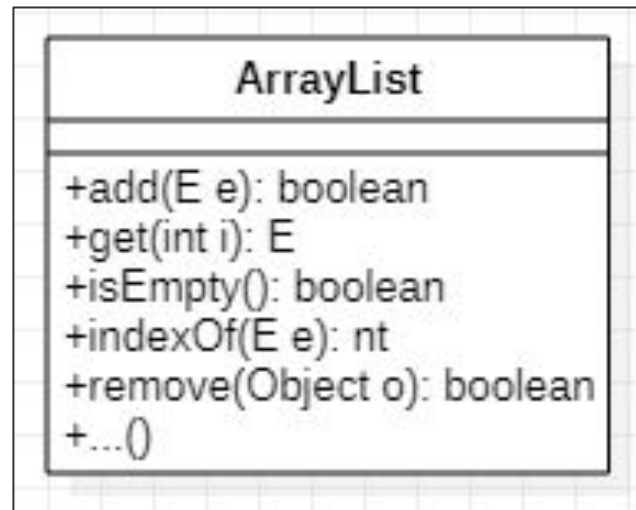
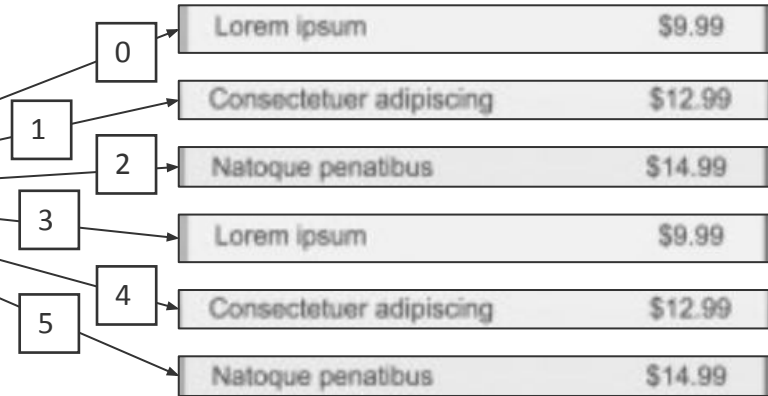
Colecciones como objetos



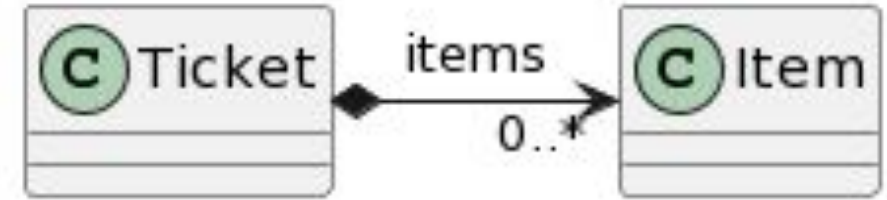
CASH RECEIPT	
Shop Name	Adress Line
Date:	MM/DD/YY
Manager:	Lorem Ipsum
<hr/>	
Tax	\$12.27
Total	\$77.83
<hr/>	
xxxx xxxx xxxx 1234 Visa	\$77.83
<hr/>	
Thank you for shopping!	

items

: ArrayList



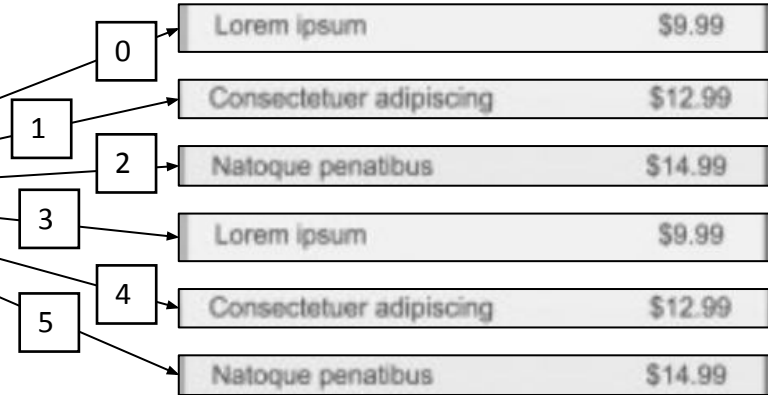
Colecciones como objetos



CASH RECEIPT	
Shop Name	Adress Line
Date:	MM/DD/YY
Manager:	Lorem lmsum
<hr/>	
Tax	\$12.27
Total	\$77.83
<hr/>	
xxxx xxxx xxxx 1234 Visa	\$77.83
<hr/>	
Thank you for shopping!	

items

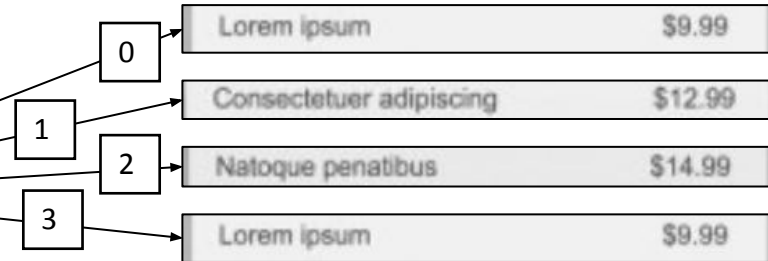
: ArrayList



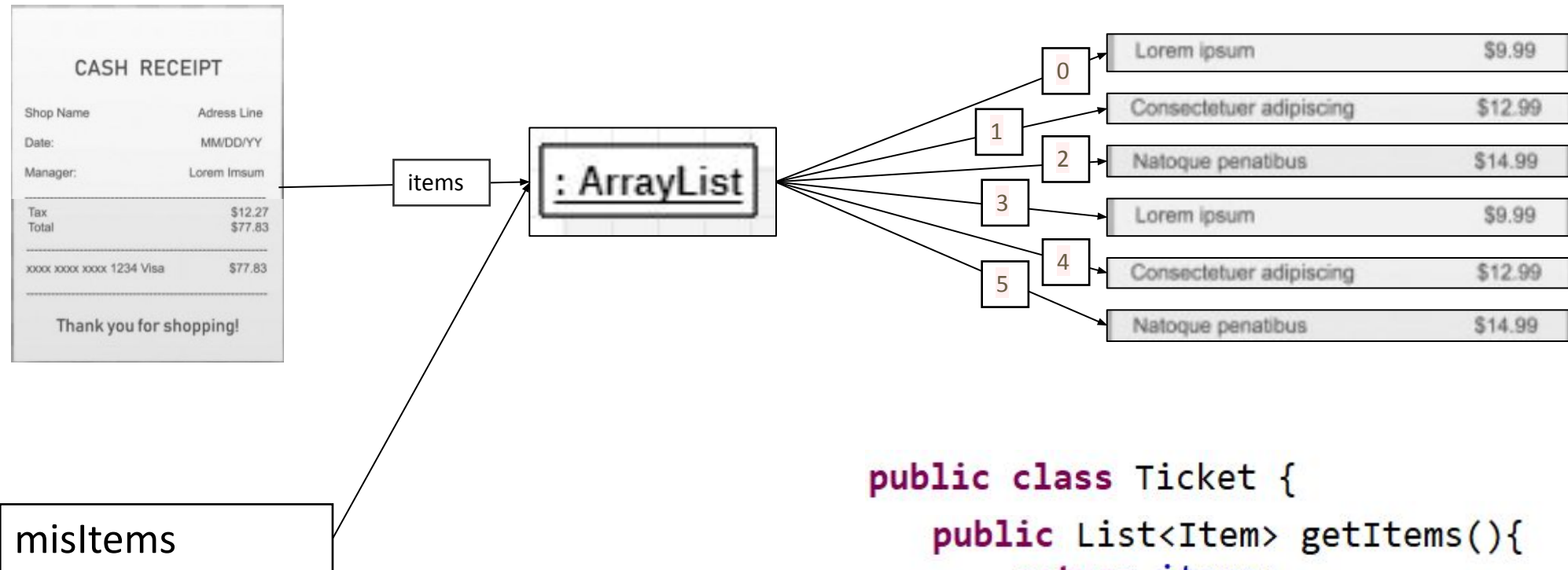
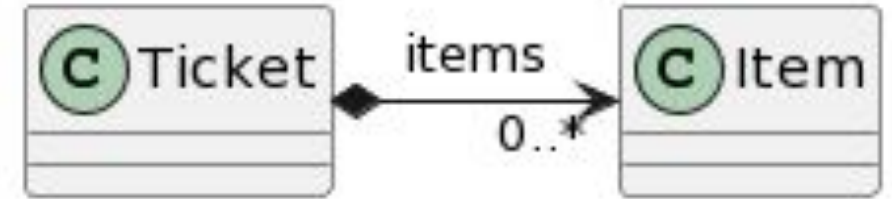
CASH RECEIPT	
Shop Name	Adress Line
Date:	MM/DD/YY
Manager:	Lorem lmsum
<hr/>	
Tax	\$12.27
Total	\$77.83
<hr/>	
xxxx xxxx xxxx 1234 Visa	\$77.83
<hr/>	
Thank you for shopping!	

items

: ArrayList



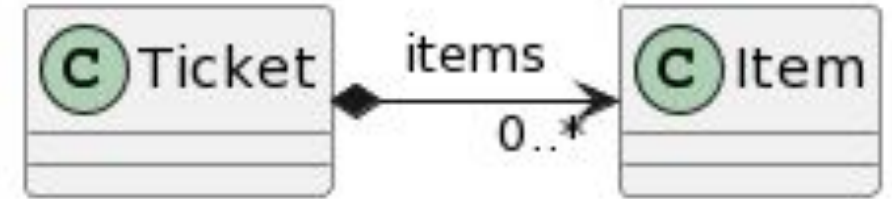
Colecciones como objetos



```
public class Ticket {  
    public List<Item> getItems(){  
        return items;  
    }  
}
```

```
List<Item> misItems = ticket.getItems();
```

Colecciones como objetos



CASH RECEIPT	
Shop Name	Adress Line
Date:	MM/DD/YY
Manager:	Lorem Ipsum
Tax	\$12.27
Total	\$77.83

xxxx xxxx xxxx 1234 Visa	\$77.83

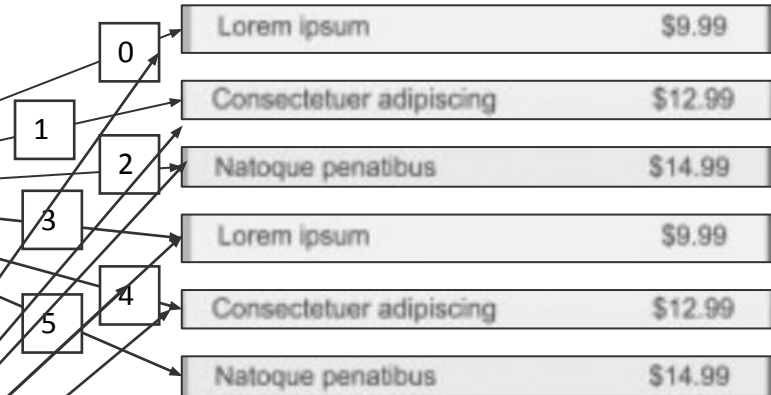
Thank you for shopping!	

items

: ArrayList

misItems

: ArrayList



```
public class Ticket {  
    public List<Item> getItems(){  
        return new ArrayList<>(this.items);  
    }  
}
```

```
List<Item> misItems = ticket.getItems();
```


Precaución

- Nunca modifico una colección que obtuve de otro objeto
- Cada objeto es responsable de mantener los invariantes de sus colecciones
- Solo el dueño de la colección puede modificarla
- Recordar que una colección puede cambiar luego de que la obtengo



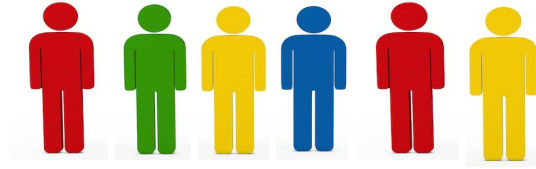
```
Ticket ticket = new Ticket(cliente);  
ticket.addItem(new Item(producto, 10));  
ticket.getItems().add(new Item(producto, 10));
```

Librería/framework de colecciones

- Todos los lenguajes OO ofrecen librerías de colecciones
 - Buscan abstracción, interoperabilidad, performance, reuso, productividad
- Las colecciones admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento
- La librería de colecciones de Java se organiza en términos de:
 - Interfaces: representa la esencia de distintos tipos de colecciones
 - Clases abstractas: capturan aspectos comunes de implementación
 - Clases concretas: implementaciones concretas de las interfaces
 - Algoritmos útiles (implementados como métodos estáticos)

Algunos tipos comunes de colecciones (interfaces)

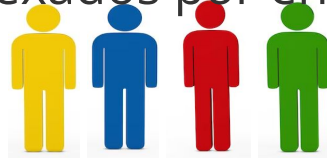
- List (`java.util.List`)



- Admite duplicados

- Sus elementos se están indexados por enteros de 0 en adelante (su posición)

- Set (`java.util.Set`)

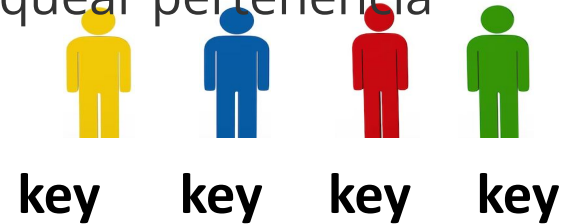


- No admite duplicados.

- Sus elementos no están indexados, ideal para chequear pertenencia

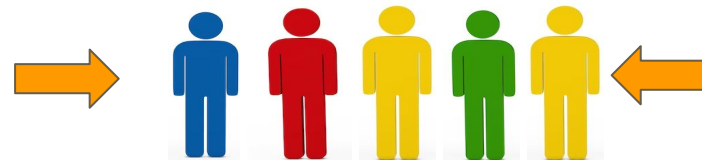
- Map (`java.util.Map`)

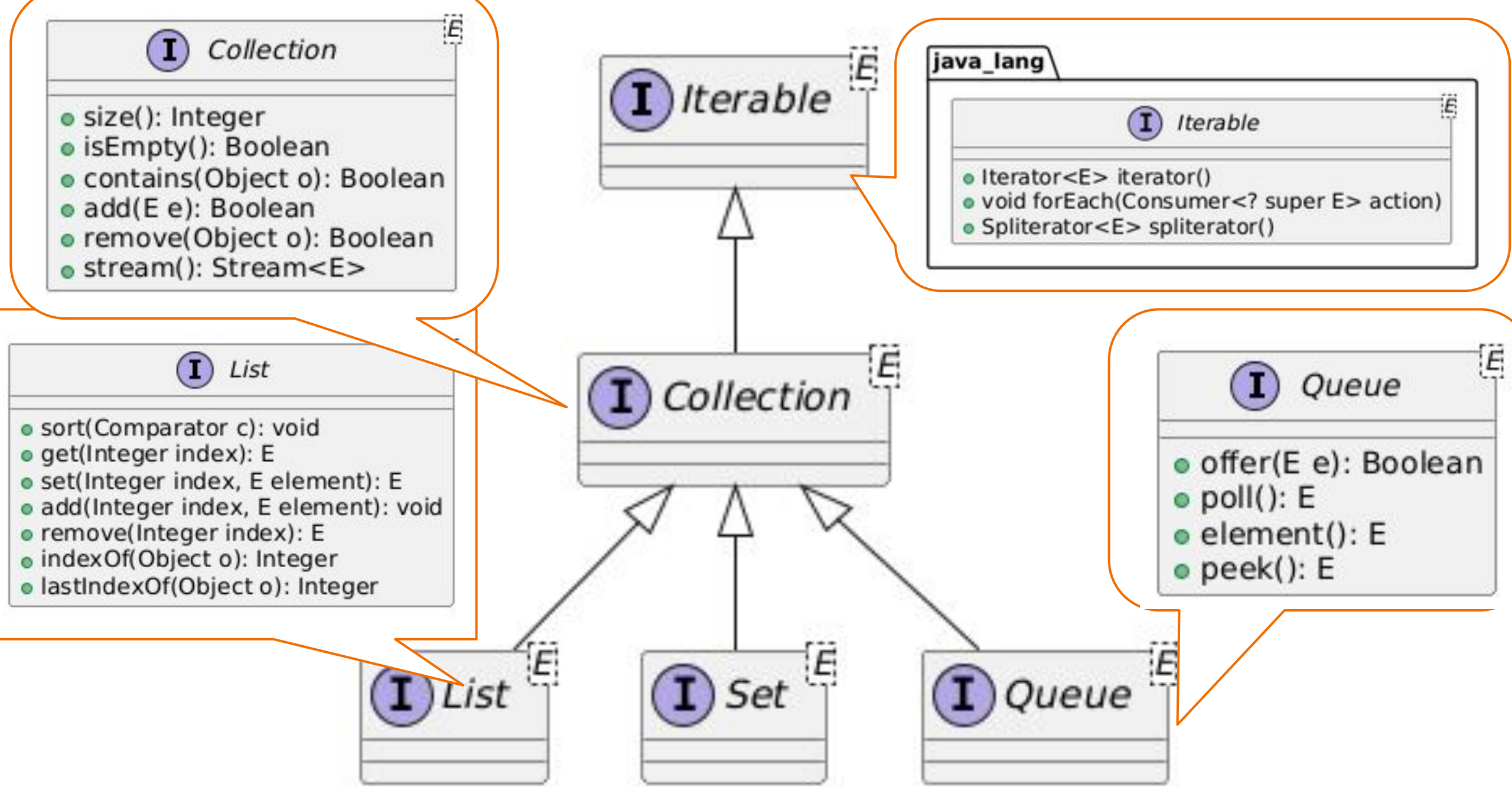
- Asocia objetos que actúan como claves a otros que actúan como valores

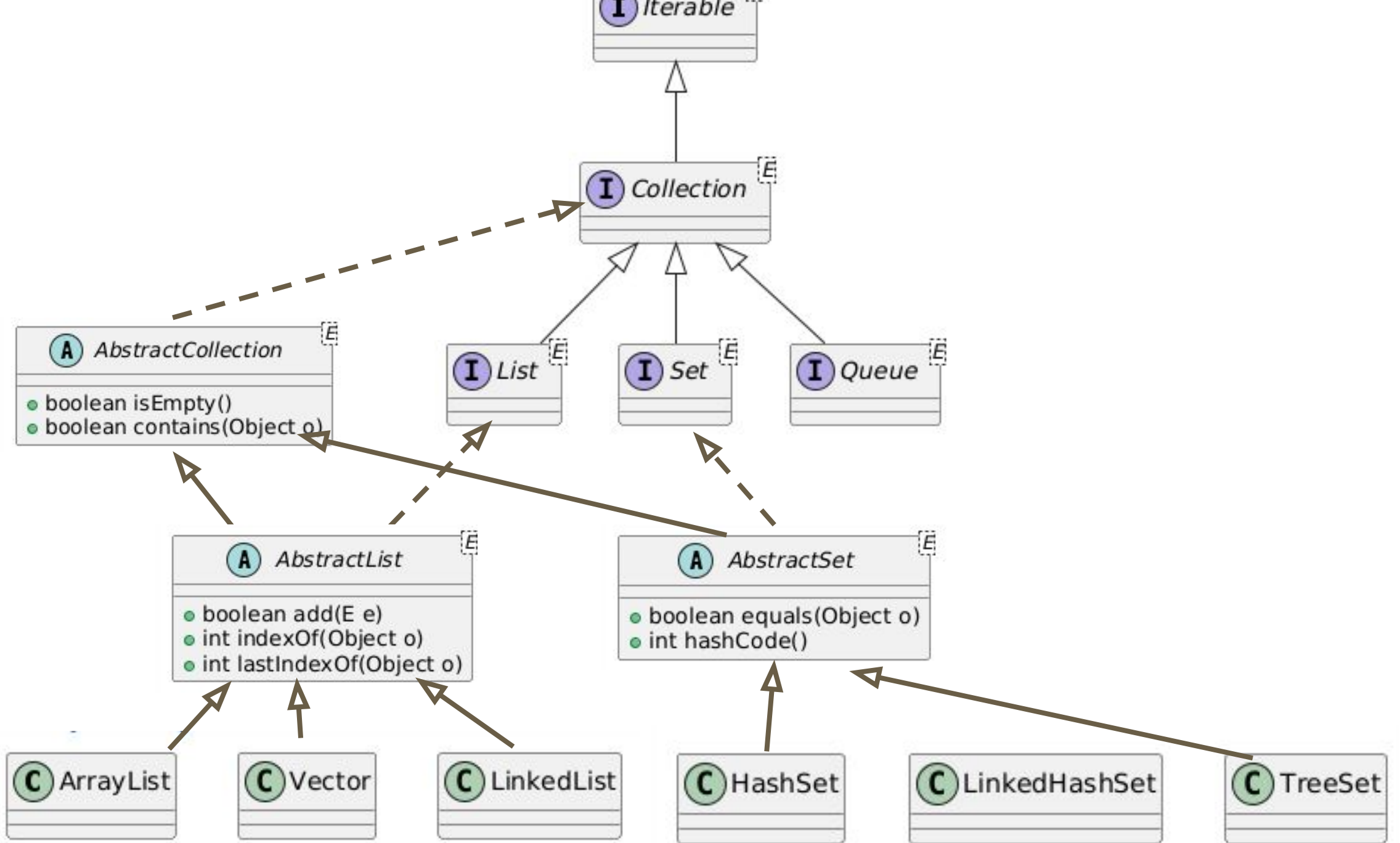


- Queue (`java.util.Queue`)

- Maneja el orden en que se recuperan los objetos (LIFO, FIFO, por prioridad, etc.)

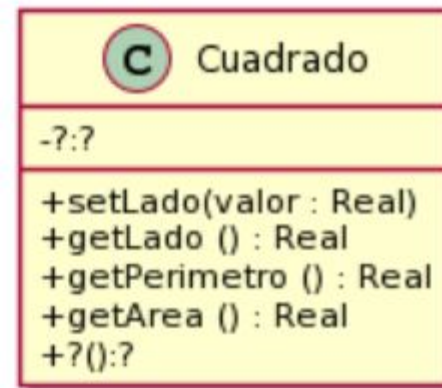






Generics y polimorfismo

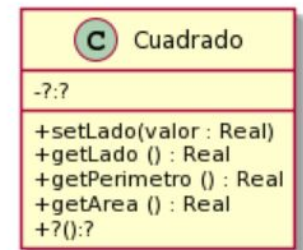
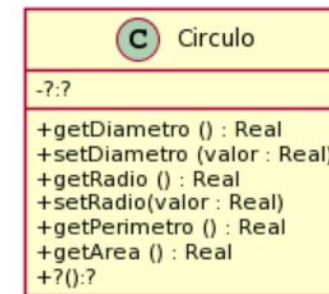
- Las colecciones admiten cualquier objeto en su contenido
- Cuanto mas sepa el compilador respecto al contenido de la colección, mejor podrá chequear lo que hacemos
- Contenido homogéneo da lugar a polimorfismo
- Al definir y al instanciar una colección indico el tipo de su contenido



Generics y polimorfismo

- Las colecciones admiten cualquier objeto en su contenido
- Cuanto mas sepa el compilador respecto al contenido de la colección, mejor podrá chequear lo que hacemos
- Contenido homogéneo da lugar a polimorfismo
- Al definir y al instanciar una colección indico el tipo de su contenido

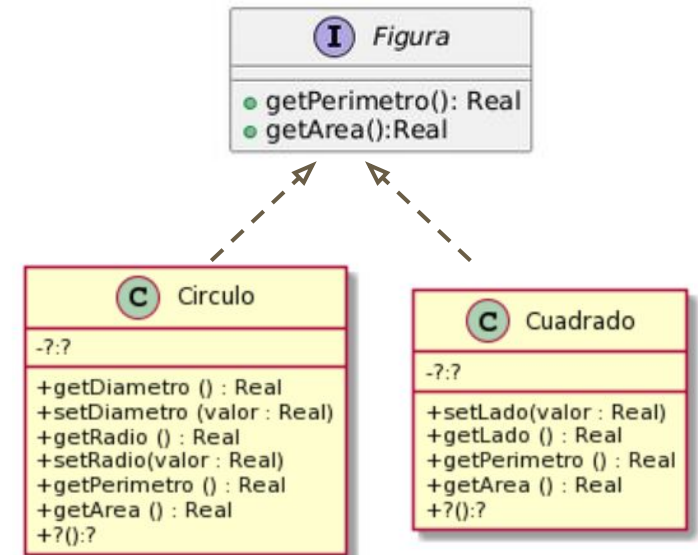
```
ArrayList figuras = new ArrayList();  
figuras.add(new Circulo());  
figuras.add(new Cuadrado());  
    figura = figuras.get(0);
```



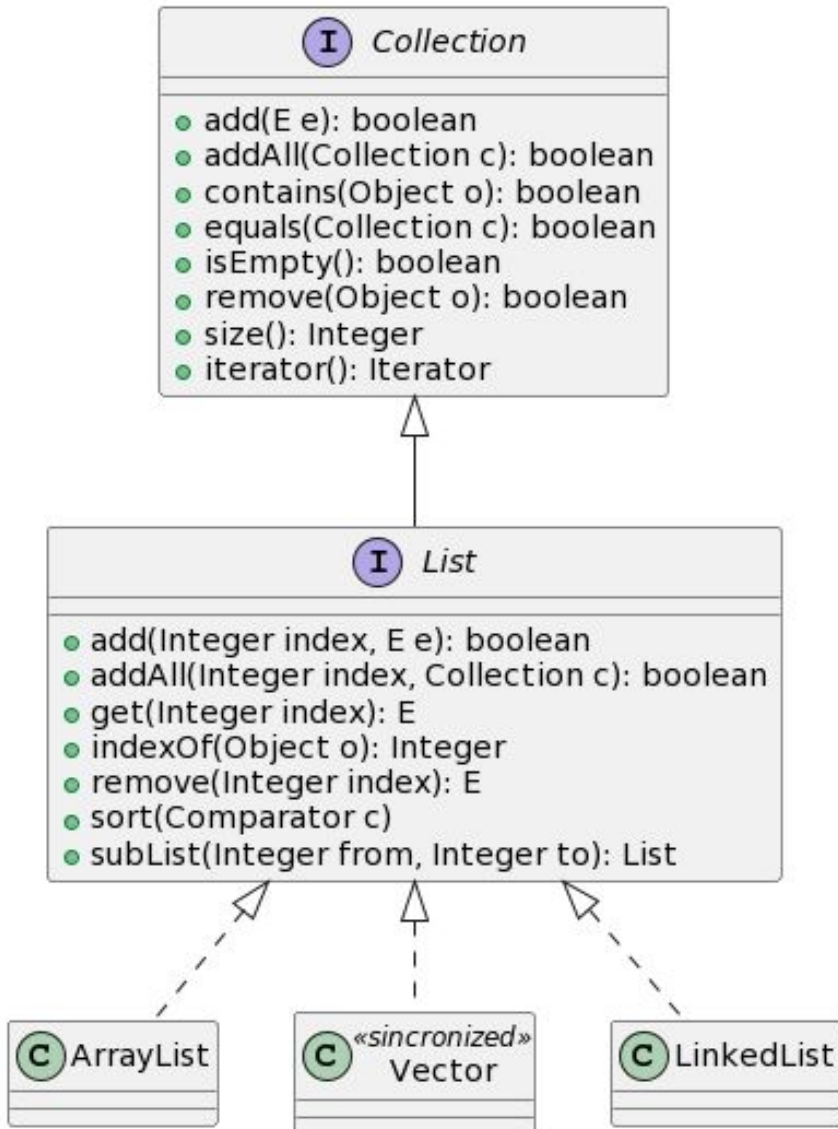
Generics y polimorfismo

- Las colecciones admiten cualquier objeto en su contenido
- Cuanto más sepa el compilador respecto al contenido de la colección, mejor podrá chequear lo que hacemos
- Contenido homogéneo da lugar a polimorfismo
- Al definir y al instanciar una colección indico el tipo de su contenido

```
ArrayList<Figura> figuras = new ArrayList<Figura>();  
figuras.add(new Circulo());  
figuras.add(new Cuadrado());  
Figura figura = figuras.get(0);
```



List



```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
```

```
public class Presupuesto {
    private LocalDate fecha;
    private String nombreCliente;
    private List<Item> items;
```

```
    public Presupuesto(String nombreCliente) {
        this.nombreCliente = nombreCliente;
        fecha = LocalDate.now();
        items = new ArrayList<>();
    }

    public void agregarItem(String detalle,
        int cantidad, double costoUnitario) {
        items.add(new Item(detalle, cantidad, costoUnitario));
    }

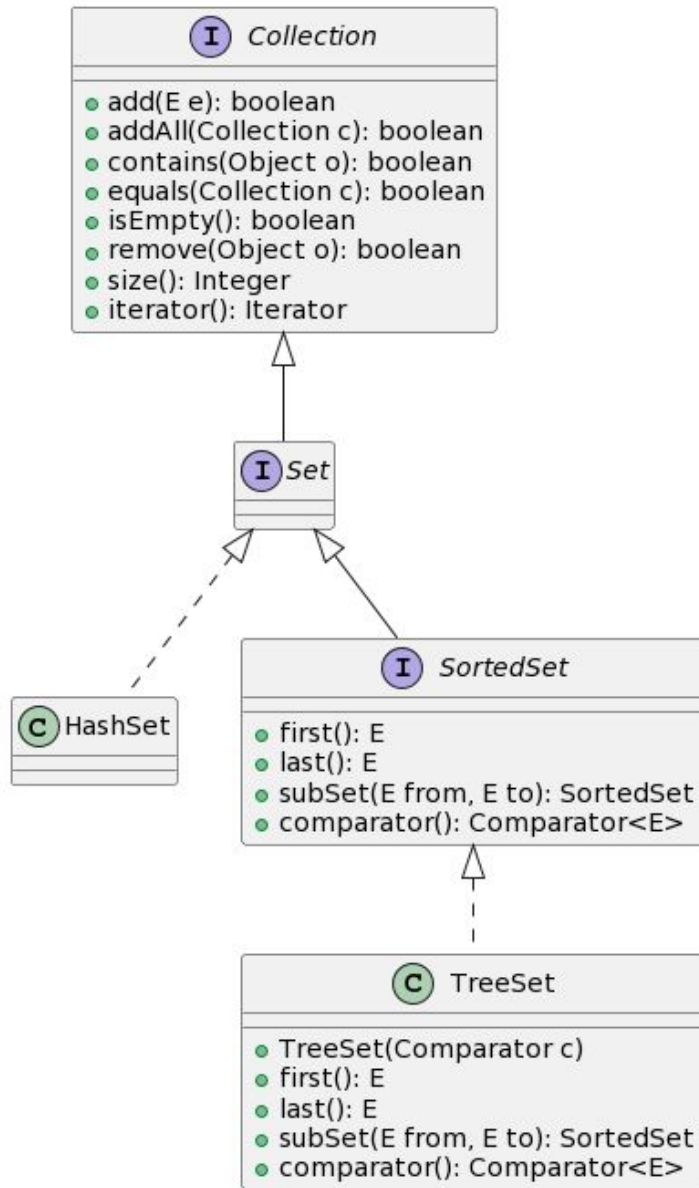
    public List<Item> getItems(){
        return items;
    }

    public int cantidadItems() {
        return items.size();
    }
}
```

Utilizamos la interfaz para darle el tipo a la variable

Inferencia de tipos: no es necesario indicarlo nuevamente

Set



```
import java.util.HashSet;
import java.util.Set;

public class Grupo {

    private String nombre;
    private Set<Persona> miembros;

    public Grupo() {
        miembros = new HashSet<>();
    }

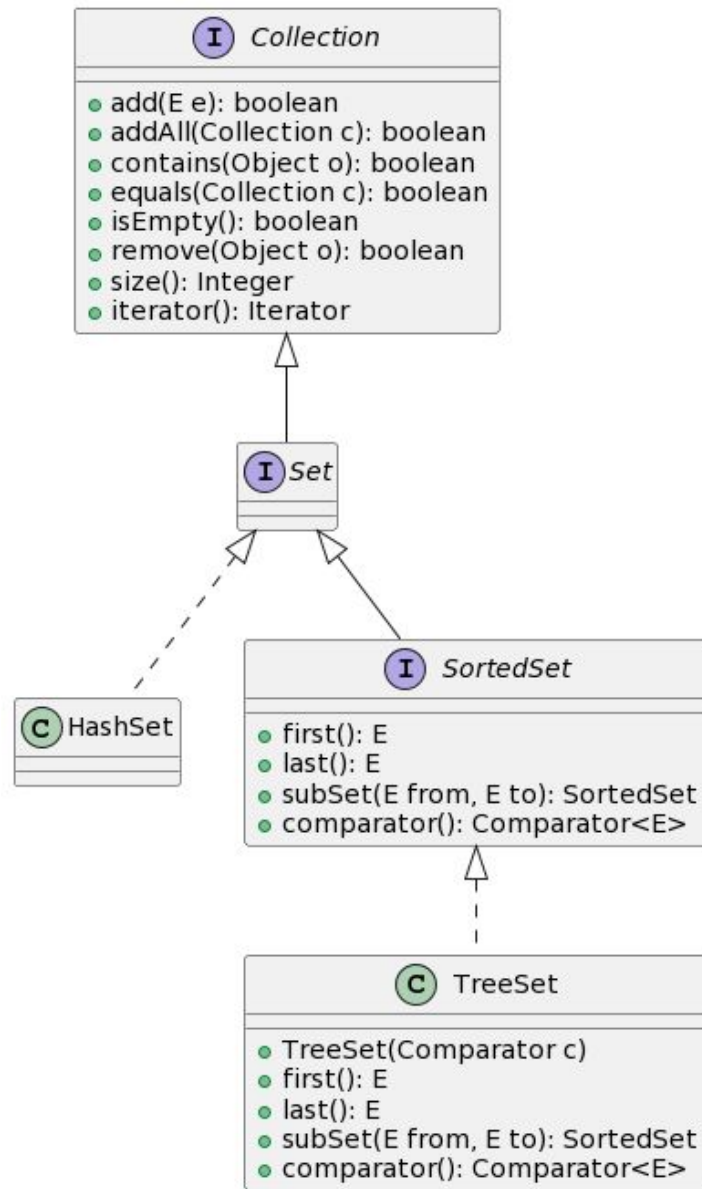
    public boolean agregarMiembro(Persona nuevo) {
        return miembros.add(nuevo);
    }

    public boolean esMiembro(Persona alguien) {
        return miembros.contains(alguien);
    }

    public int cantidad() {
        return miembros.size();
    }
}
```

Prestar atención cuando los objetos de la colección cambian, y ese cambio afecta al ***equals()*** y al ***hashCode()***

Set



```
import java.util.HashSet;
import java.util.Set;
```

```
public class Grupo {
```

```
    private String nombre;
```

```
    private Set<Persona> miembros;
```

```
public class Persona {
```

```
    private String nombre;
```

```
    private String apellido;
```

```
    private String dni;
```

```
    public Persona(String nombre, String apellido, String dni) {
```

```
        this.nombre = nombre;
```

```
        this.apellido = apellido;
```

```
        this.dni = dni;
```

```
    }
```

```
    @Override
```

```
    public int hashCode() {
```

```
        return this.dni.hashCode();
```

```
    }
```

```
    @Override
```

```
    public boolean equals(Object other) {
```

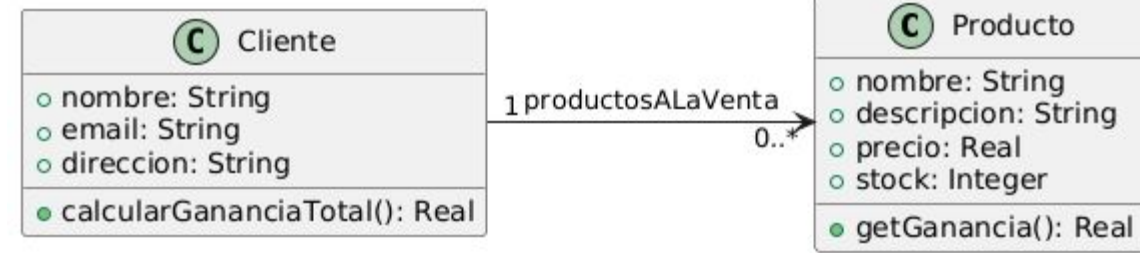
```
        ...
```

```
    }
```

En lo posible, delegar en una implementación de hashCode existente

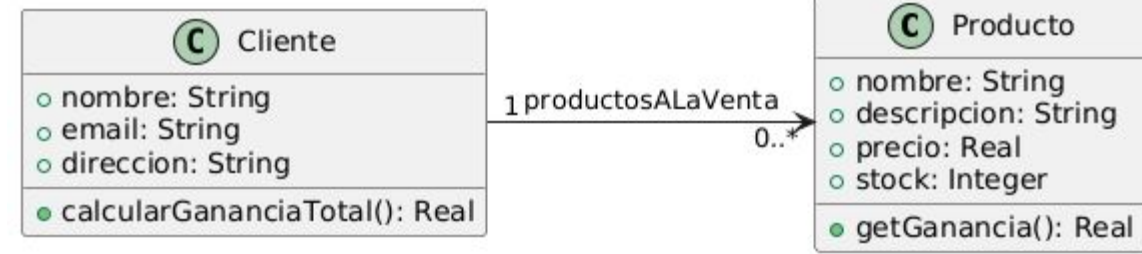
Operaciones sobre colecciones

Operaciones frecuentes



- Siempre (o casi) que tenemos colecciones repetimos las mismas operaciones:
 - Ordenar respecto a algún criterio
 - Recorrer y hacer algo con todos sus elementos
 - Encontrar un elemento (max, min, DNI = xxx, etc.)
 - Filtrar para quedarme solo con algunos elementos
 - Recolectar algo de todos los elementos
 - Reducir (promedio, suma, etc.)
- Nos interesa escribir código que sea independiente (tanto como sea posible) del tipo de colección que utilizamos

Ordenando colecciones - Comparable



producto_1
nombre = "Harina"
precio = 100

producto_2
nombre = "Aceite"
precio = 500

```
public class Producto implements Comparable<Producto>{
    @Override
    public int compareTo(Producto o) {
        return this.nombre.compareTo(o.getNombre());
    }
}
```

orden natural

negativo -> el primero es menor

0 -> son iguales

positivo -> el primero es mayor

```
Collections.sort(productos);
```

Quiero ordenar los productos por precio (?)

Método estático

Ordenando colecciones - Comparator

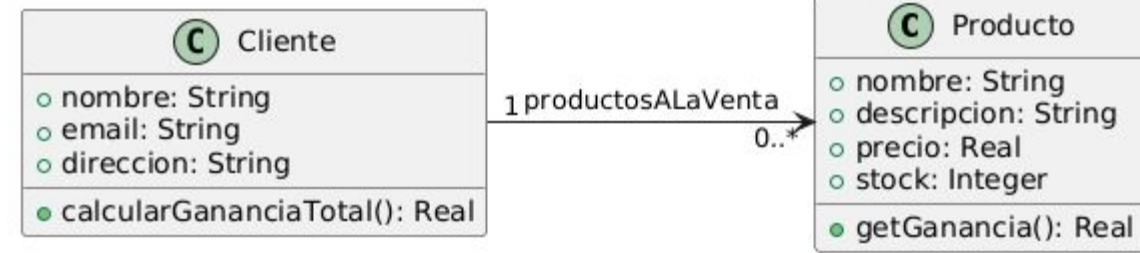
- En Java, para ordenar nos valemos de un Comparador
- Los TreeSet usan un comparador para mantenerse ordenados
- Para ordenar List , le enviamos el mensaje sort, con un comparador como parámetro

```
public class ComparadorProductoNombre implements Comparator<Producto> {  
  
    @Override  
    public int compare(Producto p1, Producto p2) {  
        return p1.getNombre().compareTo(p2.getNombre());  
    }  
  
    public class ComparadorProductoPrecio implements Comparator<Producto> {  
  
        @Override  
        public int compare(Producto p1, Producto p2) {  
            return Double.compare(p1.getPrecio(), p2.getPrecio());  
        }  
    }  
}
```

Collections.sort(productos, new ComparadorProductosNombre());

productos.sort(new ComparadorProductoNombre());

Recorriendo colecciones



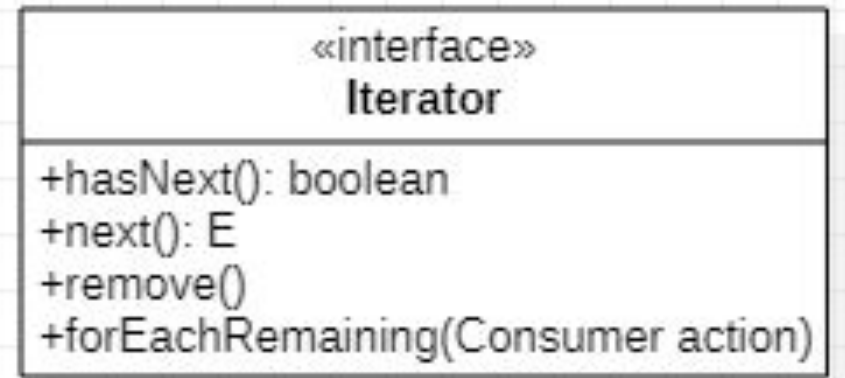
- Recorrer colecciones es algo frecuente
- El loop de control es un lugar más donde cometer errores
- El código es repetitivo y queda atado a la estructura/tipo de la colección

```
for (int i=0; i < clientes.size(); i++ ) {
    Cliente cli = clientes.get(i);
    for (int j=0; j < productos.size(); j++ ) {
        Producto prod = productos.get(j);
        // hacer algo con los clientes y los productos
    }
}
```

○ ¿ Funciona si la colección es un Set ?

Iterator (iterador externo)

- Todas las colecciones entienden iterator()
- Proporciona una manera de recorrer (o iterar) sobre los elementos de una colección de forma secuencial sin exponer su representación interna.
- Esto es útil para trabajar con diferentes tipos de colecciones (como listas, conjuntos y colas) de manera uniforme.
- Un Iterator encapsula:
 - Cómo recorrer una colección particular
 - El estado de un recorrido
- No nos interesa la clase del iterador (son polimórficos)



```
for (Iterator<Cliente> it = clientes.iterator(); it.hasNext(); ) {  
    Cliente cli = it.next();  
    // hago algo con el cliente  
}
```

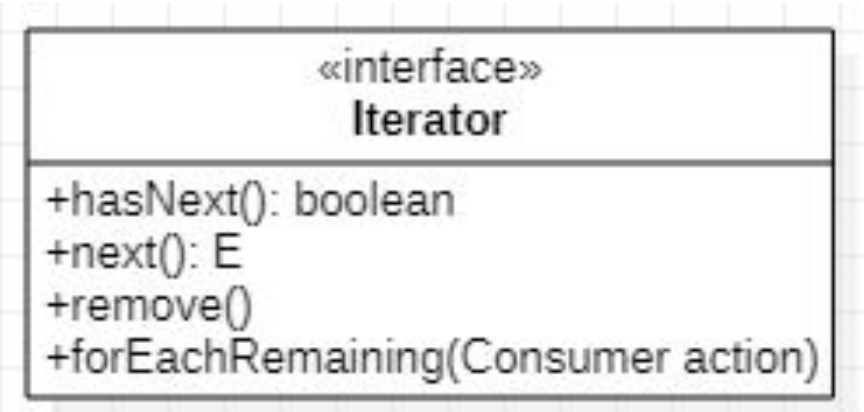
```
Iterator<Cliente> iterator =  
    clientes.iterator();  
while (iterator.hasNext()) {  
    Cliente cli = iterator.next();  
    //hago algo con el cliente  
}
```

Iterator (iterador externo)

- **bucle for-each**: Ofrece una sintaxis más limpia y menos propensa a errores.

No permite modificar la colección.

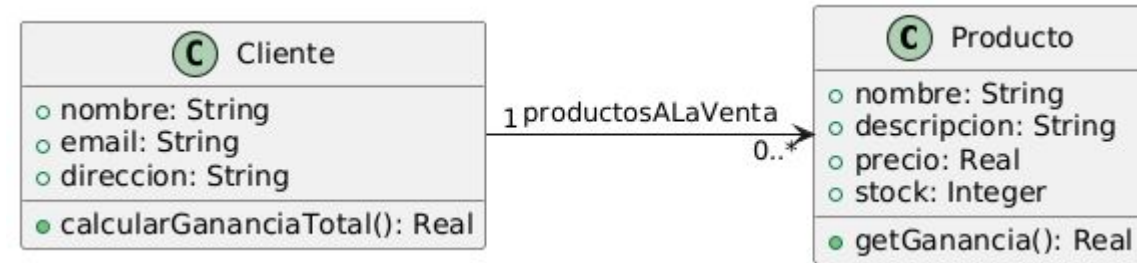
Usa internamente un **Iterator** para iterar sobre los elementos de una colección.



```
for (TipoElemento elemento : colección) {  
    // Operaciones con el elemento  
}
```

```
for (Cliente cli : clientes) {  
    // hago algo con el cliente  
}
```

¿Qué pasa si tengo que hacer varias operaciones seguidas?



Una a continuación de la otra....

Por ejemplo tengo que

- filtrar los productos cuyo precio sea más de 100 pesos
- ordenarlos por nombre
- armar una lista solo con los nombres y retornarla



Streams



Expresiones Lambda (clausuras / closures)

- Son métodos anónimos (no tienen nombre, no pertenecen a ninguna clase)
- Útiles para:
 - parametrizar lo que otros objetos deben hacer
 - decirle a otros objetos que me avisen cuando pase algo (callbacks)

```
clientes.iterator().forEachRemaining(c -> c.pagarLasCuentas());
```

```
clientes.forEach(c -> c.pagarLasCuentas());
```

```
JButton button = new JButton("Click Me!");  
button.addActionListener(e -> this.handleButtonAction(e));
```


Expresiones Lambda - sintaxis

(parámetros, separados, por, coma) -> { cuerpo lambda }

Ejemplos

```
c -> c.esMoroso()
```

```
(alumno1, alumno2) ->
```

```
Double.compare(alumno1.getPromedio(), alumno2.getPromedio())
```

1. Parámetros:

- Cuando se tiene un solo parámetro los paréntesis son opcionales.
- Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
- Opcionalmente se puede indicar el tipo, sino lo infiere.

2. Cuerpo de lambda

- Si el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y el return es implícito
- Si el cuerpo de la expresión lambda tiene varias líneas, es necesario usar llaves y debe escribirse el return.

Las operaciones hasta ahora: filtrar, coleccionar, reducir, encontrar ...

```
//Calcular el total de deuda morosa
```

```
double deudaMorosa = 0;  
for (Cliente cli : clientes) {  
    if (cli.esMoroso()) {  
        deudaMorosa += cli.getDeuda();  
    }  
}
```

```
//Generar facturas de pago para los deudores morosos
```

```
List<Factura> facturasMorosas = new ArrayList<>();  
for (Cliente cli : clientes) {  
    if (cli.esMoroso()) {  
        facturasMorosas.add(this.facturarDeuda(cli));  
    }  
}
```

- El iterador simplifica los recorridos pero ...

```
//Identificar el cliente moroso de mayor deuda
```

```
Cliente deudorMayor;  
double deudaMayor = 0;  
for (Cliente cli : clientes) {  
    if (cli.esMoroso()) {  
        if (deudaMayor < cli.getDeuda()) {  
            deudaMayor = cli.getDeuda();  
            deudorMayor = cli;  
        }  
    }  
}
```

Iteración externa

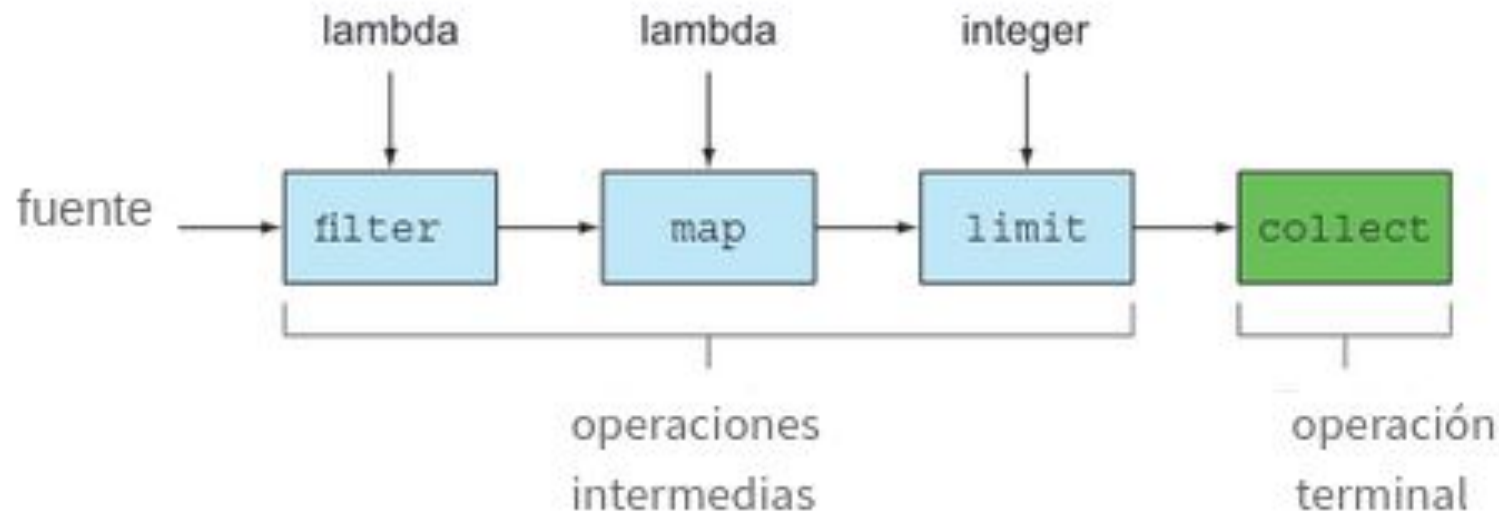
Stream

- Expresamos lo que queremos de una forma más abstracta y declarativa -> código más fácil de entender y mantener
- Las operaciones se combinan para formar pipelines (tuberías)
- No almacenan los datos, sino que proveen acceso a una fuente de datos subyacente (colección, canal I/O, etc.)
- Cada operación produce un resultado, pero **no modifica la fuente**
- Potencialmente sin final
- Consumibles: Los elementos se procesan de forma secuencial y se descartan después de ser consumidos
- La forma más frecuente de obtenerlos es vía el mensaje stream() a una colección

Iteración interna

Stream Pipelines

- Para construir un pipeline se encadenan envíos de mensajes
 - Una fuente, de la que se obtienen los elementos
 - Cero o más operaciones intermedias, que devuelven un nuevo stream
 - Operaciones terminales, que retornan un resultado
- La operación terminal guía el proceso. Las operaciones intermedias son Lazy: se calculan y procesan solo cuando es necesario, es decir, cuando se realiza una operación terminal que requiere el resultado.



Stream Pipelines - Algunos ejemplos

Operaciones intermedias

filter

de un stream obtengo otro con igual o menos elementos.

map

de un stream obtengo otro con el mismo número de elementos pero eventualmente de distinto tipo

limit

de un stream obtengo otro de un numero especifico de elementos

sorted

de un stream obtengo otro ordenado

Operaciones terminales

count | sum

average

findAny | findFirst

retorna Optional

collect

anyMatch | allMatch |

noneMatch

retorna boolean

min | max

retorna Optional

Optional

- se utiliza para representar un valor que podría estar presente o ausente en un resultado.
- son una forma de manejar la posibilidad de valores nulos de manera más segura y explícita.
- Algunos métodos en Streams, como `findFirst()` o `max()`, devuelven un `Optional` para representar el resultado.
- Luego, se puede utilizar métodos de `Optional` como `ifPresent()`, `orElse()`, `orElseGet()`, entre otros, para manipular y obtener el valor de manera segura.

Operación intermedia: filter()

- El mensaje filter retorna un nuevo stream que solo “deja pasar” los elementos que cumplen cierto predicado
- El predicado es una expresión lambda que toma un elemento y resulta en true o false

```
List<Alumno> ingresantesEnAnio2020 = alumnos.stream()  
    .filter(alumno -> alumno.getAnioIngreso() == 2020)  
    .collect(Collectors.toList());
```

Operación intermedia: map()

- El mensaje map() nos da un stream que transforma cada elemento de entrada aplicando una función que indiquemos
- La función de transformación (de mapeo) recibe un elemento del stream y devuelve un objeto

```
List<Factura> facturas = this.getFacturas();  
Set<String> cuits = facturas.stream()  
    .map(fact -> fact.getCuit())  
    .collect(Collectors.toSet());
```

Operación intermedia: sorted()

- Se usa para ordenar los elementos de la secuencia en un orden específico.
- Se puede usar para ordenar elementos en orden natural (si son comparables) o se debe proporcionar un comparador personalizado para especificar cómo se debe realizar la ordenación

```
List<Alumno> alumnosOrdenados = alumnos.stream()
    .sorted((a1, a2) ->
Double.compare(a1.getPromedio(), a2.getPromedio()))
    .collect(Collectors.toList());
```

Operación terminal: collect()

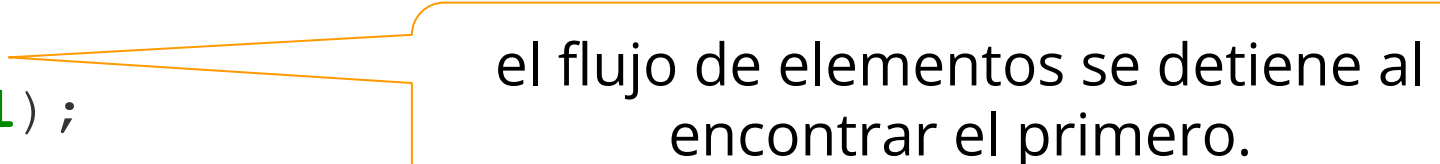
- El mensaje collect() es una operación terminal
- Es un “reductor” que nos permite obtener un objeto o colección de objetos a partir de los elementos del stream
- Recibe como parámetro un objeto Collector
 - Podemos programar uno, pero solemos utilizar los que “fabrica” Collectors (Collectors.toList(), Collectors.counting(), ...)

```
List<Factura> facturas = this.getFacturas();  
long aConsumidorFinal = facturas.stream()  
    .filter(fact -> fact.esConsumidorFinal())  
    .collect(Collectors.counting()); // podría ser count()
```


Operación terminal: findFirst()

- El mensaje findFirst() es una operación terminal
- Devuelve un Optional con el primer elemento del Stream si existe.
- Luego puedo usar)
 - `orElse()` que devuelve el valor contenido en el Optional si está presente. Si el Optional está vacío, entonces `orElse()` devuelve el valor predeterminado proporcionado como argumento.

```
Alumno primerAlumnoNombreConLetraM = alumnos.stream()  
    .filter(alumno -> alumno.getNombre().startsWith("M"))  
    .findFirst()  
    .orElse(null);
```



el flujo de elementos se detiene al encontrar el primero.

Filtrar, coleccionar, reducir, encontrar ...

```
//Calcular el total de deuda morosa
double deudaMorosa = 0;
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        deudaMorosa += cli.getDeuda();
    }
}
```

```
//Generar facturas de pago para los deudores morosos
List<Factura> facturasMorosas = new ArrayList<>();
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        facturasMorosas.add(this.facturarDeuda(cli));
    }
}
```

```
//Identificar el cliente moroso de mayor deuda
Cliente deudorMayor;
double deudaMayor = 0;
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        if (deudaMayor < cli.getDeuda()) {
            deudaMayor = cli.getDeuda();
            deudorMayor = cli;
        }
    }
}
```

Usando iteradores externos

```
//Calcular el total de deuda morosa
double deudaMorosa = clientes.stream()
    .filter(cli -> cli.esMoroso())
    .mapToDouble(cli -> cli.getDeuda())
    .sum();
```

```
//Generar facturas de pago para los deudores morosos
List<Factura> facturasMorosas = clientes.stream()
    .filter(cli -> cli.esMoroso())
    .map(cli -> this.facturarDeuda(cli))
    .collect(Collectors.toList());
```

```
//Identificar el cliente moroso de mayor deuda
Cliente deudorMayor = clientes.stream()
    .filter(cli -> cli.esMoroso())
    .max(Comparator.comparing(cli -> cli.getDeuda()))
    .orElse( other: null);
```

Usando Streams

Quando no es necesario usar streams ...

Los lenguajes de programación proporcionan nuevas características que están disponibles para ser utilizadas.

Es esencial estar dispuesto a explorar y aprender estas funcionalidades

- saber cuándo y cómo aplicarlas de manera efectiva
- reconocer cuándo no son apropiadas.

Siempre que me sea posible voy a usar alguna construcción de más alto nivel

- Son más concisas
- Están optimizadas y probadas

Cuando no es necesario usar streams ...

Sin embargo, es importante reconocer que **en algunos casos, no es la elección óptima**, y es necesario considerar otras soluciones más adecuadas a la situación específica.

- quiero ordenar una colección y que se mantenga ordenada por un criterio
- quiero eliminar los elementos que cumplen una condición
- cuando no requiero recorrer secuencialmente porque el algoritmo no lo requiere

Para llevarse

- Ojo con las colecciones de otro ... son una invitación a romper el encapsulamiento
- Observar la estrategia de diseño de encapsular lo que varía/molesta
- Seguir investigando los protocolos de las colecciones, de los streams (no hablamos de reduce), y de Collectors
 - No queremos reinventar la rueda

Preguntas