# Modulo 2

Docente: Henry Antonio Mendoza Puerta
MitoCode

# Interfaces

# Interfaces

- A partir de Java 8, las interfaces pueden tener "default methods”.
- Esto tiene implicaciones conceptuales y prácticas, que serán discutidas más adelante en este capítulo.
- Primero vamos a trabajar con la definición "clásica" de interfaces. Luego vamos a añadir el concepto de default methods.

# Interfaces

- La interfaz es un tipo que define un conjunto de operaciones que una clase debe implementar.
- La interfaz establece un contrato que la clase debe cumplir.

```
interface Shape {

    double area();
    double perimeter();
}
```

- ¿Para qué interfaces?.
    - Para crear sistemas con bajo acoplamiento y flexibles.

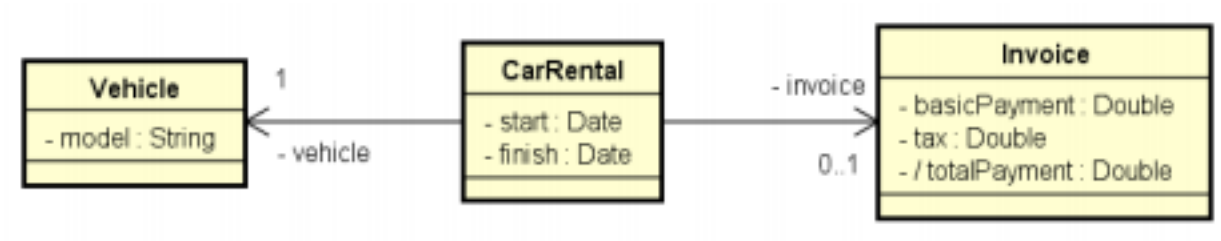
# Ejercicio

Una empresa de coches cobra un valor por hora para alquileres de hasta 12 horas. Sin embargo, si la duración del alquiler supera las 12 horas, el arrendamiento será cobrado en base a un valor diario. Además del valor del alquiler, se añade el precio del valor del impuesto conforme a las reglas del país que, en el caso de Brasil, es el 20% para valores hasta 100.00, o 15% para valores por encima de 100.00.

Hacer un programa que lee los datos del alquiler (modelo del coche, instante inicial y final de la alquiler), así como el valor por hora y el valor diario de alquiler. El programa debe entonces generar la nota de pago (que contiene valor del arrendamiento, valor del impuesto y valor total del pago) e informar los datos en la pantalla. Vea los ejemplos.

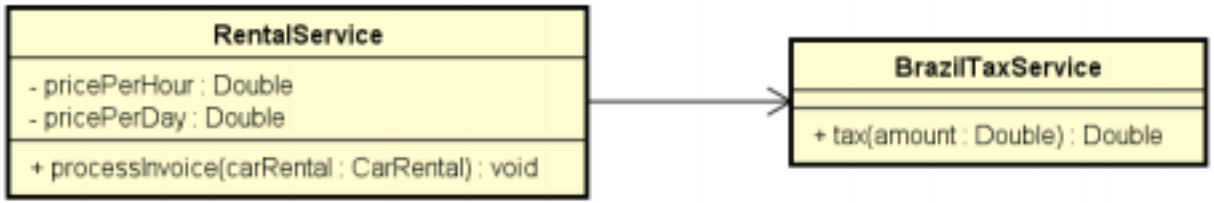
# Ejercicio

## Domain layer design



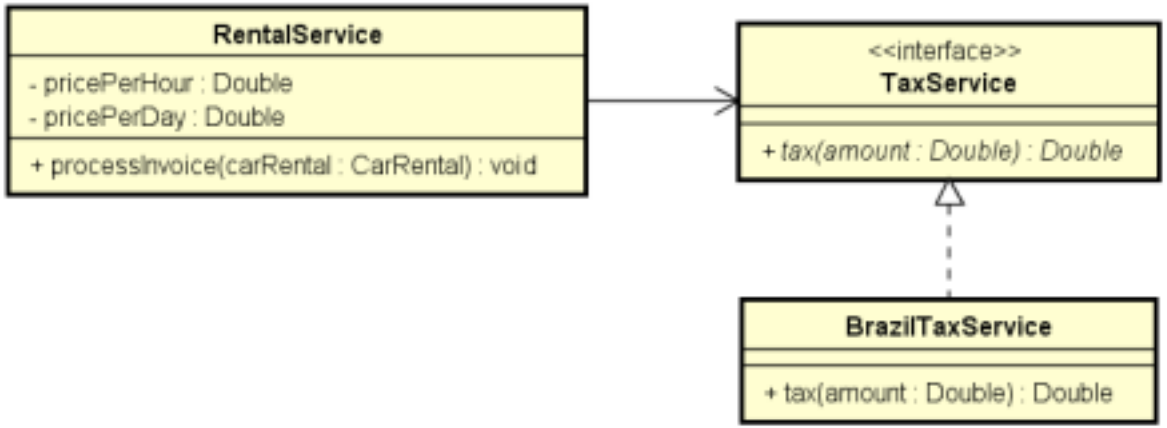
# Ejercicio

## Service layer design **(no interface)**



# Ejercicio

## Service layer design



# Ejemplo

```
Enter rental data
Car model: Civic
Pickup (dd/MM/yyyy hh:mm): 25/06/2018 10:30
Return (dd/MM/yyyy hh:mm): 25/06/2018 14:40
Enter price per hour: 10.00
Enter price per day: 130.00
INVOICE:
Basic payment: 50.00
Tax: 10.00
Total payment: 60.00
```

*Calculations:*

*Duration = (25/06/2018 14:40) - (25/06/2018 10:30) = 4:10 = 5 hours*
*Basic payment = 5 * 10 = 50*

*Tax = 50 * 20% = 50 * 0.2 = 10*

# Ejemplo

```
Enter rental data
Car model: Civic
Pickup (dd/MM/yyyy hh:mm): 25/06/2018 10:30
Return (dd/MM/yyyy hh:mm): 27/06/2018 11:40
Enter price per hour: 10.00
Enter price per day: 130.00
INVOICE:
Basic payment: 390.00
Tax: 58.50
Total payment: 448.50
```

*Calculations:*

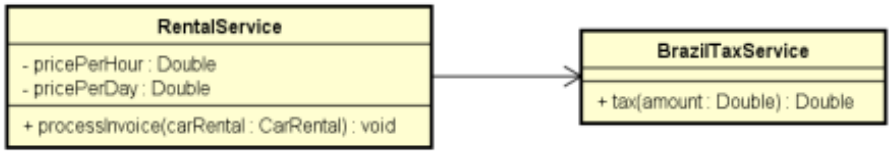
*Duration = (27/06/2018 11:40) - (25/06/2018 10:30) = 2 days + 1:10 = 3 days*
*Basic payment = 3 * 130 = 390*

*Tax = 390 * 15% = 390 * 0.15 = 58.50*

# Inyección de dependencia

# Inyección de dependencia

- Acoplamiento fuerte.
- La clase RentalService conoce la dependencia concreta.
- Si la clase concreta cambia, es necesario cambiar la clase RentalService



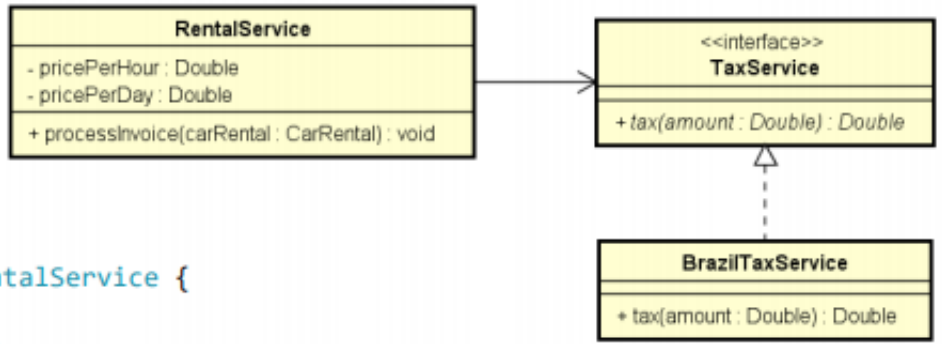

```
class RentalService {

    (...)

    private BrazilTaxService taxService;
```

# Inyección de dependencia

- Acoplamiento débil.
- La clase RentalService no conoce la dependencia concreta.
- Si la clase concreta cambia, la clase RentalService no cambia nada




```
class RentalService {

    (...)

    private TaxService taxService;
```

# Inyección de dependencia por medio de un constructor

```
class Program {
    static void Main(string[] args) {

        (...)

        RentalService rentalService = new RentalService(pricePerHour, pricePerDay, new BrazilTaxService());
```

upcasting

```
class RentalService {

    private TaxService taxService;

    public RentalService(double pricePerHour, double pricePerDay, TaxService taxService) {
        this.pricePerHour = pricePerHour;
        this.pricePerDay = pricePerDay;
        this.taxService = taxService;
    }
```

# Inversión de control

- Inversión de control
  - Estándar de desarrollo que consiste en retirar de la clase la responsabilidad de instanciar sus dependencias.
- Inyección de dependencia
  - Es una forma de realizar la inversión de control: un componente externo instancia la dependencia, que luego se inyecta en el objeto "padre". Puede ser implementada de varias formas: Constructor, Clase de instancia (builder / factory), Framework.

# Ejercicio

Una empresa desea automatizar el procesamiento de sus contratos. El procesamiento de un contrato consiste en generar los pagos para ese contrato, sobre la base del el número de meses deseado ( cuotas). La empresa utiliza un servicio de pago en línea para realizar el pago. Los servicios de pago en línea típicamente cobran un **interés mensual,** así como una **tasa por pago (payment fee)**. Por el momento, el servicio contratado por la empresa es el de Paypal, que aplica intereses simples del 1%, más una tasa de pago del 2%.

Hacer un programa para leer los datos de un contrato (número del contrato, fecha del contrato, y el valor total del contrato). A continuación, el programa debe leer el número de meses para (en el caso de que se trate de un contrato de arrendamiento, y de ahí generar los registros a ser pagadas (fecha y valor), siendo el primer tramo que se pagará un mes después de la fecha del contrato, la segunda parte dos meses después del contrato y así sucesivamente. Mostrar los datos de los pagos generados en la pantalla.

# Ejercicio

```
Enter contract data
Number: 8028
Date (dd/MM/yyyy): 25/06/2018
Contract value: 600.00
Enter number of installments: 3
Installments:
25/07/2018 - 206.04
25/08/2018 - 208.08
25/09/2018 - 210.12
```

**Calculations (1% monthly simple interest + 2% payment fee):**

**Quota #1:**
200 + 1% * 1 = 202
202 + 2% = 206.04

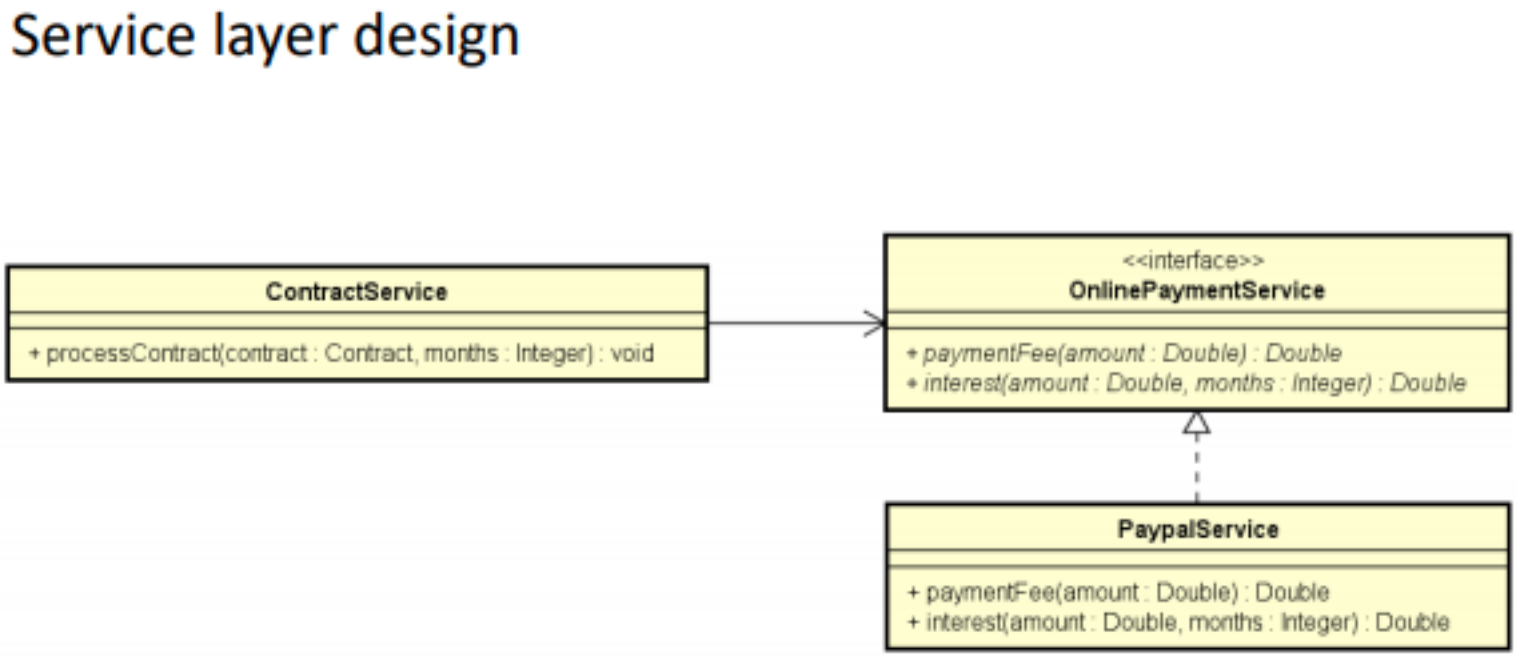
**Quota #2:**
200 + 1% * 2 = 204
204 + 2% = 208.08

**Quota #3:**
200 + 1% * 3 = 206
206 + 2% = 210.12

Ejercicio

## Domain layer design (entities)

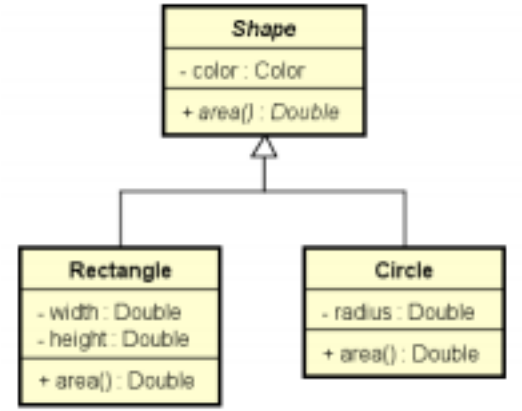
Ejercicio

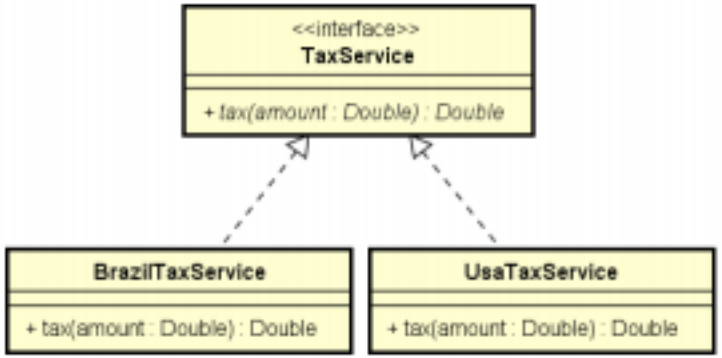

Service layer design
ContractService
+ processContract(contract : Contract, months : Integer) : void
<<interface>>
OnlinePaymentService
+ paymentFee(amount : Double) : Double
+ interest(amount : Double, months : Integer) : Double
PaypalService
+ paymentFee(amount : Double) : Double
+ interest(amount : Double, months : Integer) : Double

# Heredar vs. cumplir con el contrato

# Aspectos comunes entre herencia e interfaces

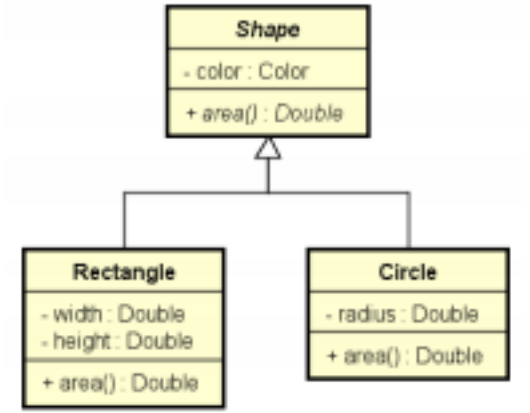
- Relación es-uno.
- Generalización / especialización.
- Polimorfismo.

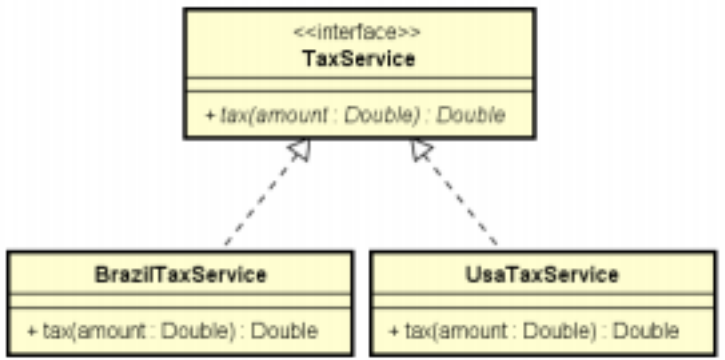


# Diferencia fundamental

- Herencia => reuso.
- Interfaz => contrato que se debe cumplir.



# Interface Comparable

# Interface Comparable

https://docs.oracle.com/javase/10/docs/api/java/lang/Comparable.html

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

# Ejercicio

Haga un programa para leer un archivo que contenga funcionarios (nombre y salario) formato .csv, almacenándolos en una lista. A continuación, ordenar la lista por nombre y mostrar el resultado en la pantalla.

```
Maria Brown,4300.00
Alex Green,3100.00
Bob Grey,3100.00
Anna White,3500.00
Alex Black,2450.00
Eduardo Rose,4390.00
Willian Red,2900.00
Marta Blue,6100.00
Alex Brown,5000.00
```

# Ejercicio

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

```java
System.out.println("maria".compareTo("alex"));
System.out.println("alex".compareTo("maria"));
System.out.println("maria".compareTo("maria"));
```

**Output:**

```
12
-12
0
```

https://docs.oracle.com/javase/10/docs/api/java/lang/Comparable.html

---

**Method compareTo:**

**Parameters:**
o - the object to be compared.

**Returns:**
a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

# Default methods

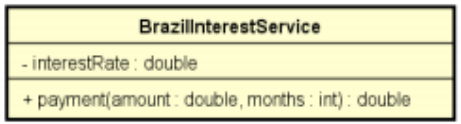
# Default methods

- Desde Java 8, las interfaces pueden contener métodos concretos.
- La intención básica es proveer la implementación estándar para los métodos, para evitar:
  - Repetición de implementación en toda clase que implemente la interfaz.
  - La necesidad de crear clases abstractas para proveer reutilización de la implementación

# Ejercicio

- Hacer un programa para leer una cantidad y la duración en meses de un año préstamo. Informar el valor que se pagará después de transcurrido el plazo del de acuerdo con las reglas de interés. La regla de cálculo de los intereses son de 2% al mes.

```
Amount: 200.00
Months: 3
Payment after 3 months:
212.24
```




**Calculations:**

$$Payment = 200 * 1.02 * 1.02 * 1.02 = 200 * 1.02^3 = 212.2416$$

$$Payment = amount * (1 + interestRate / 100)^N$$

# Ejercicio

- ¿Qué sucede si otro servicio de interés de otro país?

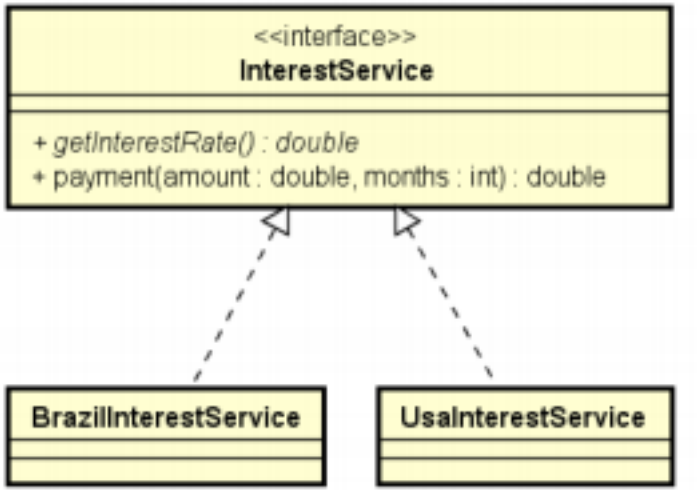
Amount: 200.00
Months: 3
Payment after 3 months:
206.06

| UsaInterestService |
| --- |
| - interestRate : double |
| + payment(amount : double, months : int) : double |

Calculations: $Payment = 200 * 1.01 * 1.01 * 1.01 = 200 * 1.01^3 = 206.0602$

$Payment = amount * (1 + interestRate / 100)^N$

# Ejercicio


<<interface>>
InterestService
+ getInterestRate() : double
+ payment(amount : double, months : int) : double
BrazilInterestService
UsaInterestService

## Consideraciones importantes

- Sí: ahora las interfaces pueden proveer reuso.
- Sí: ahora tenemos una forma de herencia múltiple.
- Interfaces siguen siendo muy diferentes de las clases abstractas. las interfaces no tienen características tales como constructores y atributos.

# Listas

# Listas

- Lista es una estructura de datos:
  - Homogénea (datos del mismo tipo).
  - Ordenada (elementos accedidos por medio de posiciones).
  - Inicia vacía, y sus elementos se asignan bajo demanda.
  - Cada elemento ocupa un "nodo" (o nodo) de la lista.
- Tipo (interface): List.
- Clases que implementan: ArrayList, LinkedList, etc.

# Listas

- Ventajas:
  - Tamaño variable.
  - Facilidad para realizar operaciones de inserción, eliminación, obtener.
- Clases que implementan: ArrayList, LinkedList, etc.
- Desventajas
  - Acceso secuencial a los elementos.

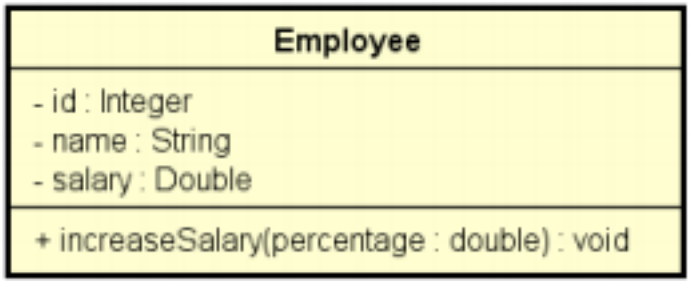
# Listas

- Tamaño de la lista: size()
- Obtener el elemento de una posición: get(position)
- Insertar elemento en la lista:add(obj), add(int,obj).
- Quitar elementos de la lista: remove(obj), remove(int),removeIf(predicate)
- Encontrar posición de elemento:indexOf(obj), lastIndexOf(obj)
- Filtrar lista basada en predicado: List result = list.stream().filter(x -> x > 4).collect(Collectors.toList());
- Encontrar primera instancia basada en predicado.

```
Integer result = list.stream().filter(x -> x > 4).findFirst().orElse(null);
```

# Ejercicio

Hacer un programa para leer un número entero N y luego los datos (id, nombre y salario) de N empleados. No debe haber repetición de id. A continuación, realizar el aumento de X por ciento en el salario de un determinado empleado. Para ello, el programa debe leer un id y el valor X. Si el id informado no existe, mensaje y abortar la operación. Al final, mostrar la lista actualizada de los funcionarios, según los ejemplos. Recuerde aplicar la técnica de encapsulación para no permitir que el salario pueda ser cambiado libremente. Un salario sólo puede incrementarse sobre la base de una operación de aumento por porcentaje dado.



# Ejercicio

```
How many employees will be registered? 3

Emplyoee #1:
Id: 333
Name: Maria Brown
Salary: 4000.00

Emplyoee #2:
Id: 536
Name: Alex Grey
Salary: 3000.00

Emplyoee #3:
Id: 772
Name: Bob Green
Salary: 5000.00

Enter the employee id that will have salary increase : 536
Enter the percentage: 10.0

List of employees:
333, Maria Brown, 4000.00
536, Alex Grey, 3300.00
772, Bob Green, 5000.00
```

```
How many employees will be registered? 2

Emplyoee #1:
Id: 333
Name: Maria Brown
Salary: 4000.00

Emplyoee #2:
Id: 536
Name: Alex Grey
Salary: 3000.00

Enter the employee id that will have salary increase: 776
This id does not exist!

List of employees:
333, Maria Brown, 4000.00
536, Alex Grey, 3000.00
```

# Generics

# Generics

Generics permiten que las clases, interfaces y métodos puedan ser parametrizados por tipo. Sus beneficios son

- Reuso
- Type safety
- Performance

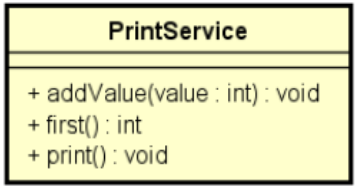
Uso común : Colecciones

```
List<String> list = new ArrayList<>();
list.add("Maria");
String name = list.get(0);
```

# Ejercicio (reuso)

Se desea hacer un programa que lee una cantidad N, y luego N números enteros. Al final, imprima estos números de forma organizada de acuerdo ejemplo. A continuación, indicar cuál fue el primer valor informado.

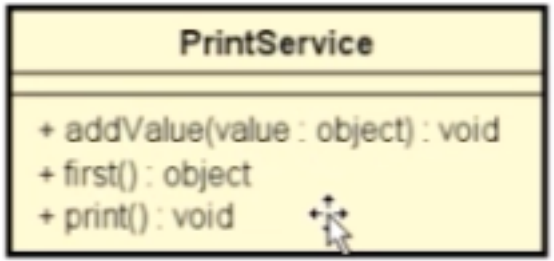
```
How many values? 3
10
8
23
[10, 8, 23]
First: 10
```



# Ejercicio (type safety & performance)

Se desea hacer un programa que lee una cantidad N, y luego N números enteros. Al final, imprima estos números de forma organizada de acuerdo ejemplo. A continuación, indicar cuál fue el primer valor informado.

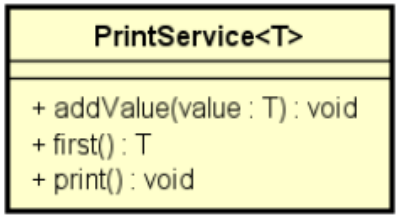
```
How many values? 3
10
8
23
[10, 8, 23]
First: 10
```



# Ejercicio (Solución con Generics)

Se desea hacer un programa que lee una cantidad N, y luego N números enteros. Al final, imprima estos números de forma organizada de acuerdo ejemplo. A continuación, indicar cuál fue el primer valor informado.

```
How many values? 3
10
8
23
[10, 8, 23]
First: 10
```



# MitoCode Network
# Descubre tu potencial

www.mitocode.com