

# Tema 3 – Herencia en Java – Parte 3

---

Programación Orientada a Objetos

Grado en Ingeniería Informática

# Contenido

---

- Clases abstractas.
- Interfaces.

# Caso de estudio

---

- ❑ Queremos desarrollar una aplicación para la gestión de **bolsas de empleo**.
- ❑ En una bolsa se inscriben *candidatos* a la espera de optar a *ofertas* de empleo.
- ❑ La funcionalidad principal es la gestión de los *llamamientos*: generar la lista de candidatos que pueden elegir entre las ofertas de la bolsa.

# Caso de estudio

---

- En la aplicación se gestionan dos **tipos de bolsas**:
  - *Bolsa conservadora*:
    - El tamaño de la lista de candidatos del llamamiento puede ser hasta el doble que las ofertas.
    - Ante una renuncia, el candidato pasa al final de la lista de candidatos.
  - *Bolsa estricta*:
    - Limita el tamaño de la lista de candidatos de un llamamiento.
    - Si un candidato renuncia a un llamamiento, es dado de baja en la bolsa.

# Caso de estudio

---

- A continuación se analiza el concepto *Bolsa Conservadora*.
- Propiedades:
  - Nombre.
  - Lista de candidatos.
  - Lista de ofertas.
- Funcionalidad:
  - Altas y baja en la lista de candidatos.
  - Altas de ofertas de empleo.
  - Generación de un llamamiento: *hasta el doble de ofertas*.
  - Renuncia de un candidato al llamamiento: *mover al final de la lista de candidatos*.

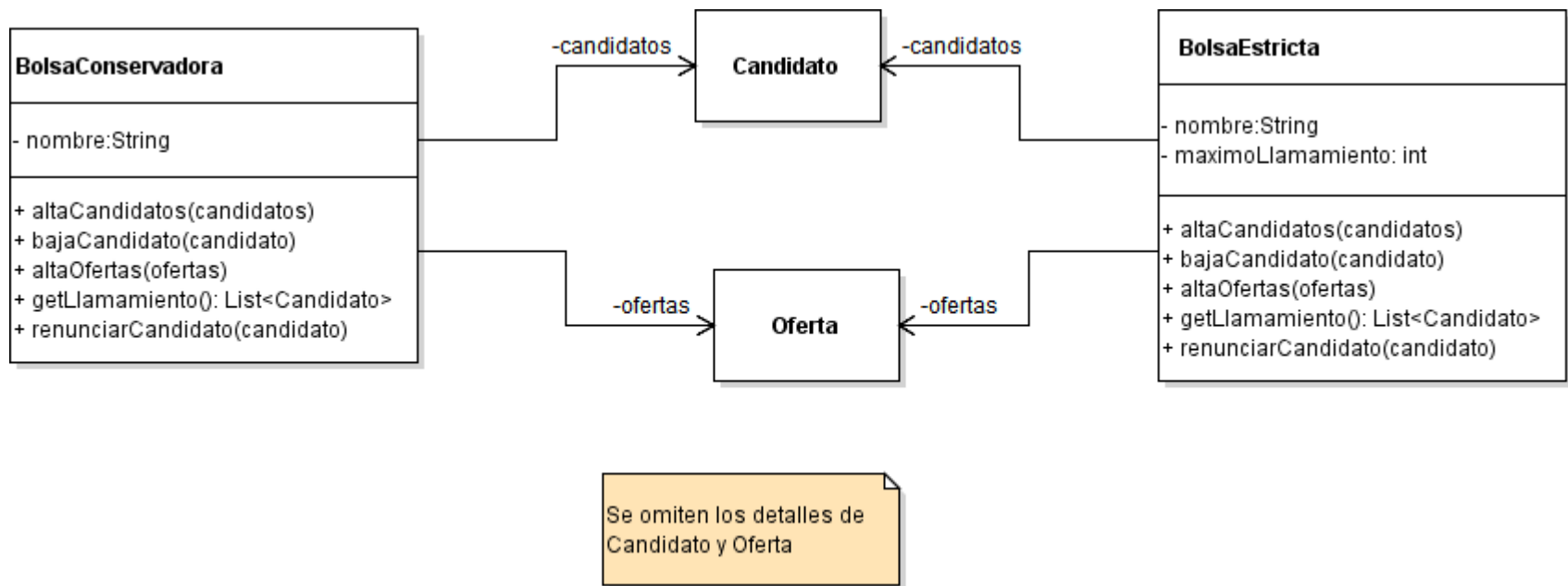
# Caso de estudio

---

- A continuación se analiza el concepto *Bolsa Estricta*.
- Propiedades:
  - Nombre.
  - Lista de candidatos.
  - Lista de ofertas.
  - **Límite máximo tamaño del llamamiento**
- Funcionalidad:
  - Altas y baja en la lista de candidatos.
  - Altas de ofertas de empleo.
  - Generación de un llamamiento: *hasta el límite máximo*.
  - Renuncia de un candidato al llamamiento: *dar de baja al candidato en la bolsa*.

# Caso de estudio

## □ Diseño inicial



# Caso de estudio

---

- ❑ El análisis de los dos conceptos muestra que comparten la mayor parte de propiedades y funcionalidad.
- ❑ **Propiedades comunes:**
  - Nombre.
  - Lista de candidatos.
  - Lista de ofertas.
- ❑ **Funcionalidad común:**
  - Altas y baja de la lista de candidatos.
  - Altas de ofertas de empleo.
  - Generación de un llamamiento: *distintos límites*.
  - Renuncia de un candidato al llamamiento: *diferente tratamiento*.



# Caso de estudio

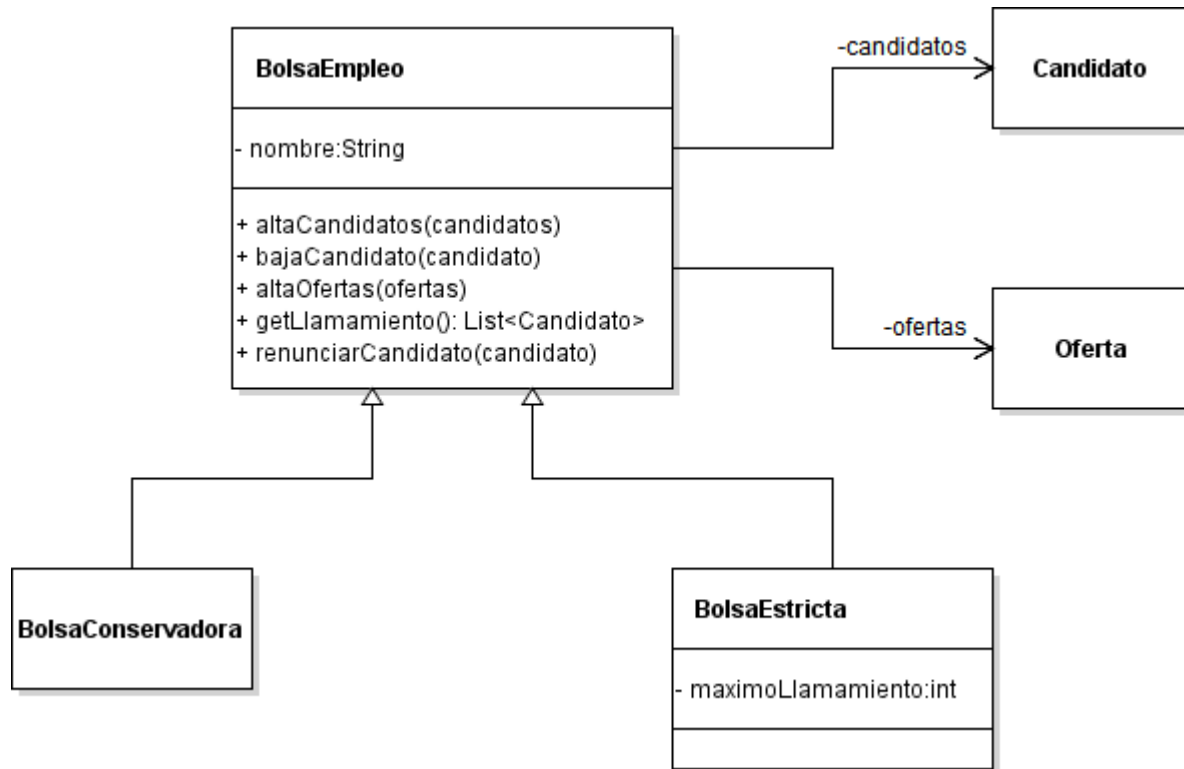
---

- ❑ La herencia nos permite **factorizar** las propiedades y funcionalidad común en un nuevo concepto (*Bolsa de Empleo*).
- ❑ *Bolsa Estricta* y *Bolsa Conservadora* podrían heredar de *Bolsa de Empleo* para reutilizar las características comunes.

➔ Este modo de aplicar herencia se denomina **Generalización**.

# Caso de estudio

- Segundo diseño: introducción de la herencia por **generalización**



# Caso de estudio

---

```
public class BolsaEmpleo {  
  
    private final String nombre;  
    private LinkedList<Candidato> candidatos;  
    private LinkedList<Oferta> ofertas;  
  
    public BolsaEmpleo(String nombre) {  
  
        this.nombre = nombre;  
        this.candidatos = new LinkedList<Usuario>();  
        this.ofertas = new LinkedList<Oferta>();  
    }  
  
    public String getNombre() { // ... }  
    public LinkedList<Oferta> getOfertas() { // ... }  
    public LinkedList<Candidato> getCandidatos() { // ... }  
  
    // ...  
}
```

# Caso de estudio

---

```
public class BolsaEmpleo {  
    // ...  
  
    public void altaCandidatos(Candidato... candidatos) {  
        Collections.addAll(this.candidatos, candidatos);  
    }  
  
    public boolean bajaCandidato(Candidato candidato) {  
        return this.candidatos.remove(candidato);  
    }  
  
    public void addOfertas(Oferta... ofertas) {  
        Collections.addAll(this.ofertas, ofertas);  
    }  
  
    // ...  
}
```

# Caso de estudio

---

- ❑ En la versión anterior de la clase `BolsaEmpleo` queda pendiente la funcionalidad sobre la generación de un llamamiento y la renuncia de un candidato al llamamiento.
- ❑ El comportamiento de ambas operaciones es diferente en los subtipos.
- ❑ **Cuestiones:**
  - ¿Debemos declarar estas operaciones en la clase `BolsaEmpleo`?
    - ❑ Sí, son operaciones comunes a toda bolsa de empleo.
  - ¿Qué código podríamos darle a esas operaciones?

# Caso de estudio

---

## □ Primera aproximación:

- Podríamos ofrecer una implementación vacía o por defecto a estas operaciones para que los subtipos las redefinan.
- Ejemplo: método de renuncia de un candidato

```
public void renunciarLlamamiento(Candidato candidato) {  
  
    // Implementación vacía  
}
```

## ■ Problemas:

- ¿Es correcto crear objetos de la clase `BolsaEmpleo` con métodos incompletos?
- ¿Qué sucedería si un subtipo olvida redefinir el método?

# Clases abstractas

---

- ❑ La Programación Orientada a Objetos introduce el concepto de **clase abstracta** como solución a los problemas anteriores.
- ❑ Una clase abstracta se caracteriza por:
  - No pueden construirse objetos de esa clase.
  - Permite definir **métodos abstractos** (sin código).
- ❑ Caso de estudio:
  - Si definimos `BolsaEmpleo` como clase abstracta evitamos construir objetos con métodos incompletos.

```
public abstract class BolsaEmpleo {  
    // ...  
}
```

# Clases abstractas

---

- Caso de estudio:
  - El concepto de clase abstracta no es suficiente para evitar el segundo problema: los subtipos pueden olvidar redefinir el método incompleto (*renunciar al llamamiento*).
- Un **método abstracto** es un método que solo declara la signatura (cabecera), nada de la implementación.
  - NO es equivalente a un método vacío.
- **Importante:** Toda clase que declare algún método abstracto, necesariamente debe ser abstracta, esto es, no se pueden construir objetos de ella.
  - **Nota:** una clase abstracta puede no tener métodos abstractos.



# Clases abstractas

---

## □ Caso de estudio:

- Declaramos el método `renunciarLlamamiento` como abstracto.

```
public abstract void renunciarLlamamiento(Candidato candidato);
```

- Un método abstracto **obliga** a los subtipos a implementar el método.
  - Un método abstracto se hereda, como cualquier otro.
  - Una clase con algún método abstracto, debe ser abstracta.
  - En definitiva, si queremos tener objetos de los subtipos debemos implementar los métodos abstractos.

# Clases abstractas

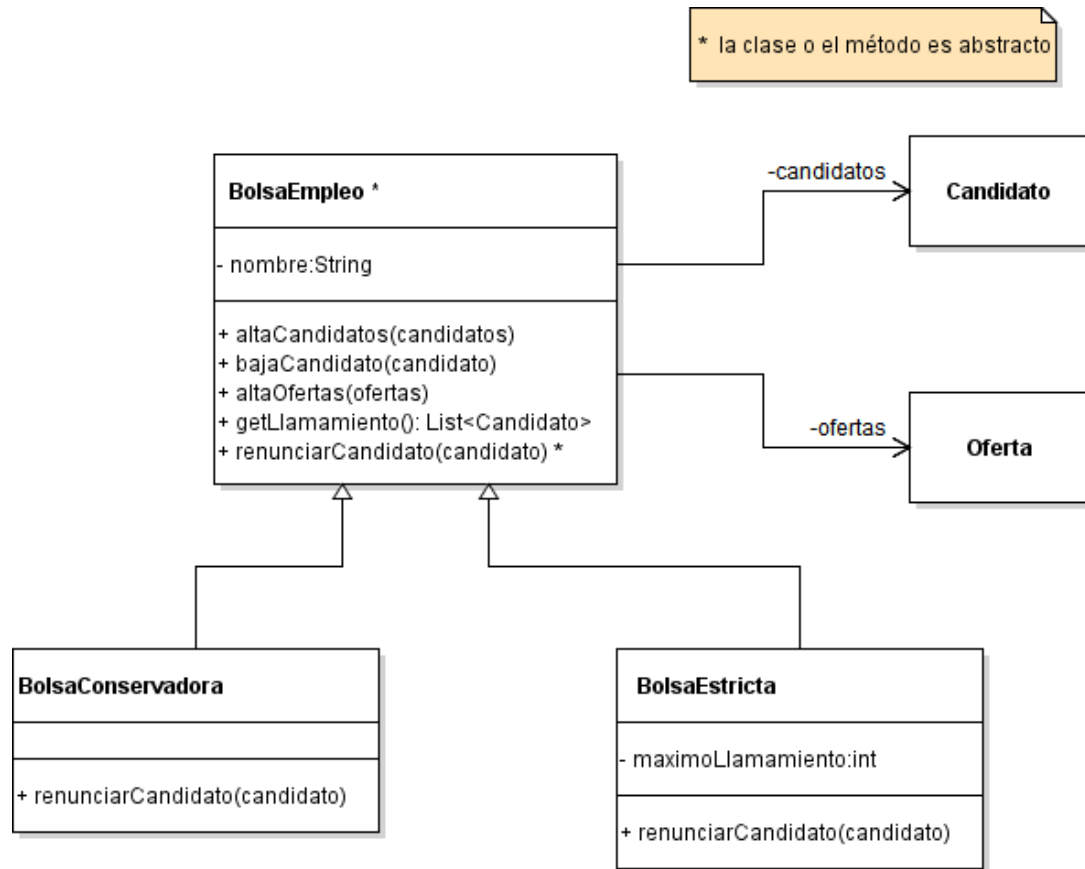
---

- Las clases y métodos abstractos resuelven los problemas que surgen al aplicar herencia por generalización (caso de estudio).
- El concepto de método abstracto es una herramienta de programación que permite a una clase establecer requisitos de implementación en las clases descendientes.
- Una clase que quiera ser efectiva (no abstracta) tiene la obligación de implementar todos los métodos abstractos que hereda.

```
public class BolsaEstricta extends BolsaEmpleo {  
    // ...  
  
    public void renunciarLlamamiento(Candidato candidato) {  
        bajaCandidato(candidato);  
    }  
}
```

# Clases abstractas

## □ Diagrama de clases



# Clases abstractas

---

## ❑ Caso de estudio:

- Declaramos abstracto el método que genera el llamamiento.

```
public abstract LinkedList<Candidato> getLlamamiento();
```

## ❑ Implementación del método en BolsaConservadora:

```
public LinkedList<Candidato> getLlamamiento() {  
    LinkedList<Candidato> llamamiento = new LinkedList<Candidato>();  
    int llamamientosRestantes = 2 * getOfertas().size();  
    LinkedList<Candidato> candidatos = getCandidatos();  
    for (int i = 0; llamamientosRestantes > 0 && i < candidatos.size(); i++) {  
        llamamiento.add(candidatos.get(i));  
        llamamientosRestantes--;  
    }  
    return llamamiento;  
}
```

# Clases abstractas

---

- En la clase `BolsaEstricta` la implementación es muy parecida, solo cambia la inicialización de la variable `llamamientosRestantes`.
  - Toma el valor de un atributo propio de la clase.

```
public LinkedList<Candidato> getLlamamiento() {  
    LinkedList<Candidato> llamamiento = new LinkedList<Candidato>();  
    int llamamientosRestantes = this.maximoLlamamiento;  
    LinkedList<Candidato> candidatos = getCandidatos();  
    for (int i = 0; llamamientosRestantes > 0 && i < candidatos.size(); i++) {  
        llamamiento.add(candidatos.get(i));  
        llamamientosRestantes--;  
    }  
    return llamamiento;  
}
```

# Clases abstractas

---

- La implementación del método que calcula el llamamiento pone de manifiesto un **problema** habitual en la implementación de métodos abstractos en los subtipos: **repetición de código**.
- **Solución:**
  - Un método abstracto puede ser utilizado por otros métodos ordinarios de una clase abstracta.
  - En tiempo de ejecución, la aplicación del método abstracto se ejecuta sobre un objeto (de un descendiente), que tendrá implementado el método.
  - De este modo se introduce el concepto de **método plantilla**: un método ordinario de una clase abstracta que hace uso de uno o más métodos abstractos.

# Clases abstractas

---

- En la clase `BolsaEmpleo` declaramos el **método plantilla**:

```
protected abstract int getMaximoConvocatoria();

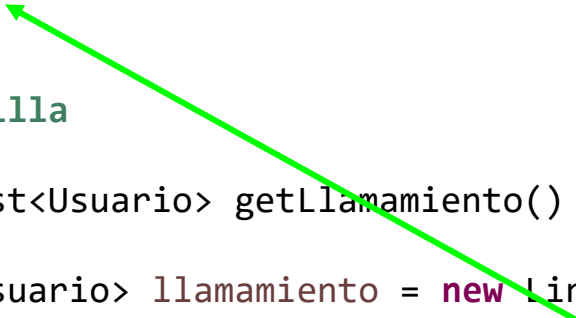
// Método plantilla

public LinkedList<Usuario> getLlamamiento() {

    LinkedList<Usuario> llamamiento = new LinkedList<Usuario>();
    int llamamientosRestantes = getMaximoConvocatoria();

    for (int i = 0; llamamientosRestantes > 0 && i < this.candidatos.size(); i++) {

        llamamiento.add(this.candidatos.get(i));
        llamamientosRestantes--;
    }
    return llamamiento;
}
```



# Clases abstractas

---

- ❑ Implementación del método abstracto en las clases descendientes.
- ❑ BolsaConservadora:

```
protected int getMaximoConvocatoria() {  
    return getOfertas().size() * 2;  
}
```

- ❑ BolsaEstricta:

```
protected int getMaximoConvocatoria() {  
    return maximoLlamamiento;  
}
```



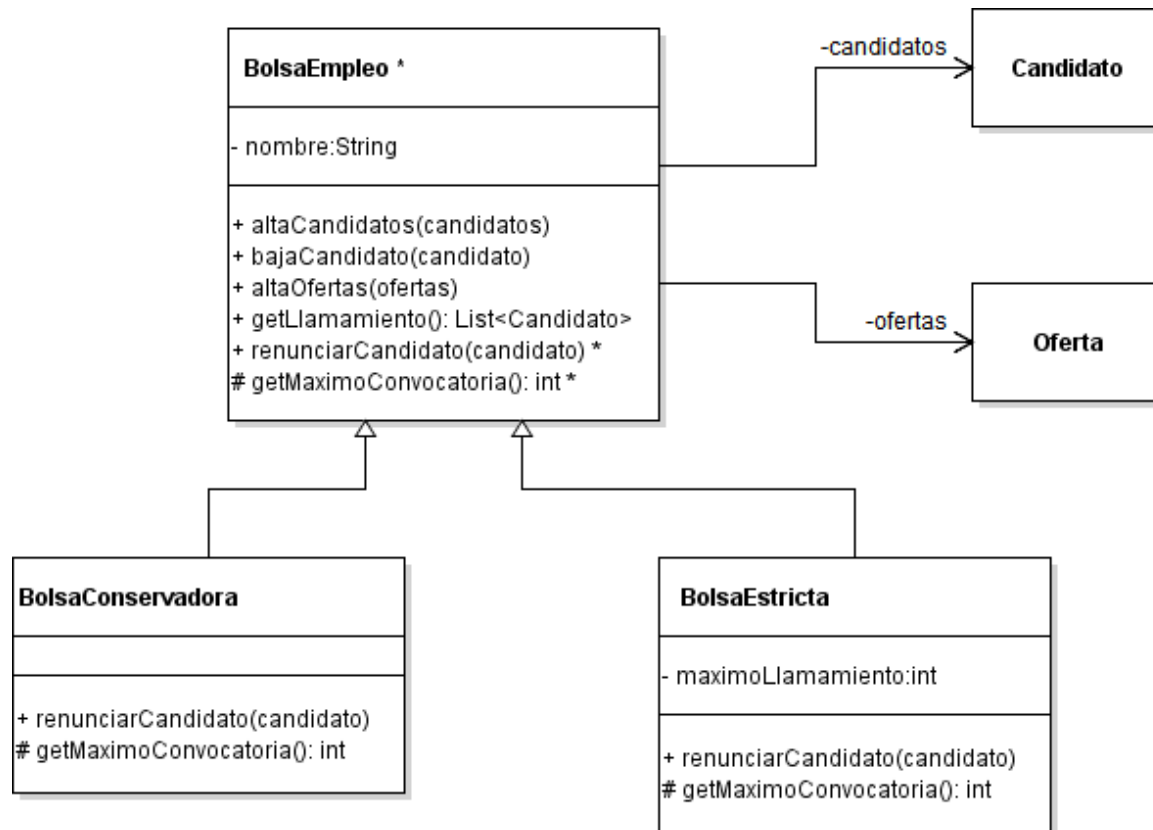
# Clases abstractas

---

- El **método plantilla** `getLlamamiento` define el algoritmo de cálculo de un llamamiento de la bolsa de empleo.
- La parte *variable* del algoritmo se declara como un método abstracto (`getMaximoConvocatoria`).
  - Las clases descendientes están obligadas a implementar este método, si no quieren ser abstractas.
  - El método es declarado con *visibilidad protegida* porque es un método de apoyo que solo debe ser visible por los descendientes, que deben implementarlo.
- El concepto de método plantilla es un mecanismo importante para definir **código reutilizable**.
- Define **comportamiento común** a todos los descendientes.

# Clases abstractas

## □ Diagrama de clases final



# Caso de estudio

---

- En el caso de estudio de las bolsas de empleo se ha aplicado **herencia por generalización** aplicando un proceso de *factorización*:
  - Se han identificado las características comunes en tiempo de diseño.
- En el siguiente caso de estudio se aplica de nuevo herencia por generalización. Sin embargo, en este caso se aplica a posteriori, esto es, una vez programado el código de las clases que van a ser factorizadas.
- A este proceso se denomina **refactorización**.

# Caso de estudio

---

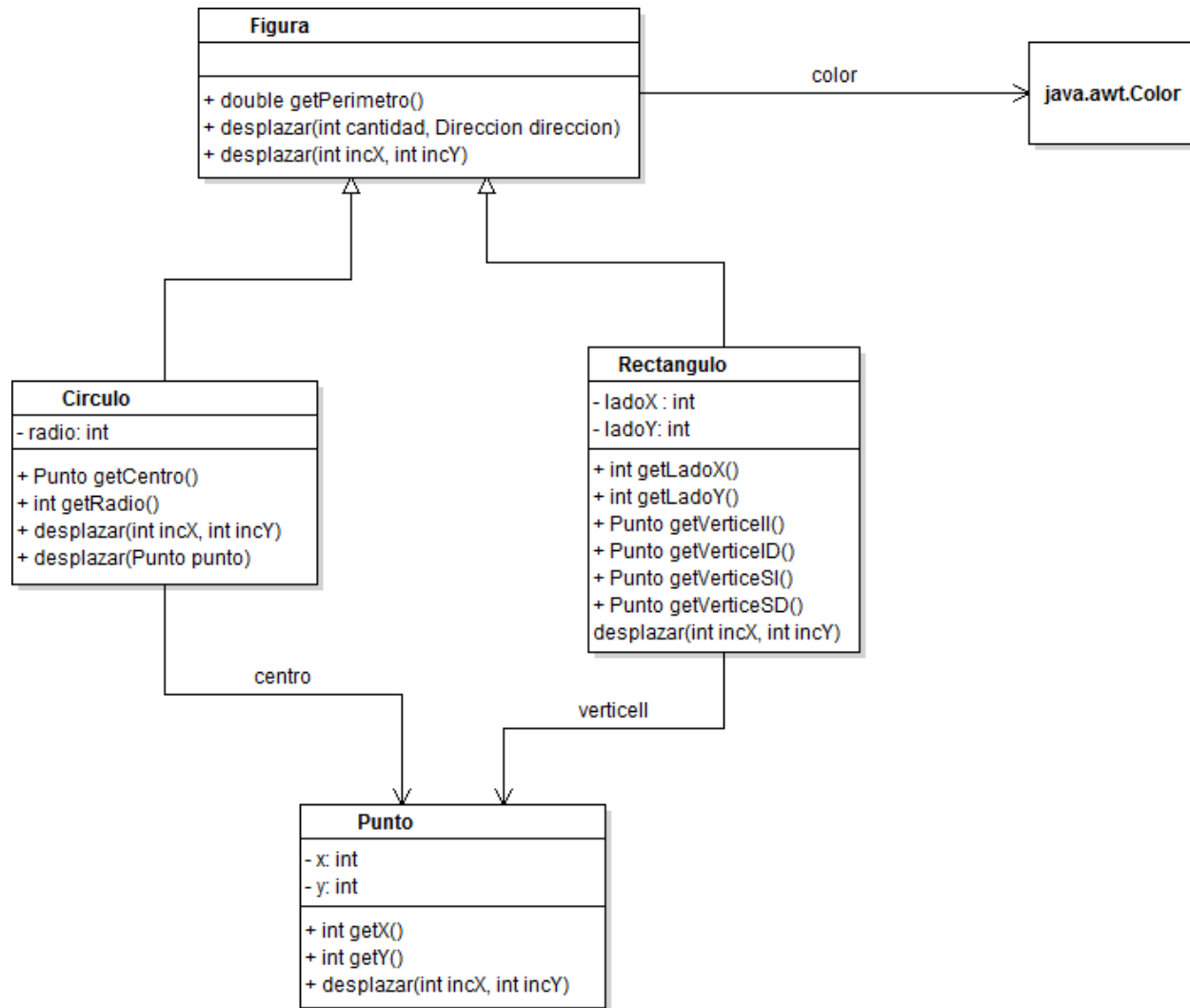
- **Motivación:**

- Tanto los círculos como los rectángulos tienen *perímetro*.
- Ambos se pueden *desplazar*.
- Pueden tener en común una nueva propiedad *color*.

- Identificamos el concepto **Figura** geométrica y definimos una clase que sea padre de **Circulo** y **Rectangulo** (**Generalización**).

- Se realiza la **refactorización** del código de las clases **Circulo** y **Rectangulo** para subir la funcionalidad común a la nueva clase.

# Caso de estudio



# Caso de estudio

---

- ❑ ¿Cómo se calcula el *perímetro* de una figura geométrica?
- ❑ ¿Tiene sentido incluir una implementación por defecto que retorne cero?
- ❑ El **método** `getPerimetro()` **no puede ser implementado** en la clase **Figura**.
- ❑ El **método** `getPerimetro()` es **abstracto**.
  - ➔ Es responsabilidad de las subclases implementarlo adecuadamente.
- ❑ La **clase** **Figura** es **abstracta**, ya que tiene un método abstracto.

# Caso de estudio

---

```
public abstract class Figura {  
    private Color color;  
  
    protected Figura(Color color){  
        this.color = color;  
    }  
  
    public Color getColor(){  
        return color;  
    }  
  
    public abstract double getPerimetro();  
    ...  
}
```

# Caso de estudio

---

- El método `desplazar(int incX, int incY)` es también **abstracto**:
  - No podemos ofrecer una implementación en la clase `Figura`.
  - Debe ser implementado en `Circulo` y `Rectangulo`.
  
- Sin embargo, el método `desplazar(int cantidad, Direccion dir)` sí puede ser programado haciendo uso del método abstracto `desplazar(int incX, int incY)`
  - Por tanto, el método es un **método plantilla**.



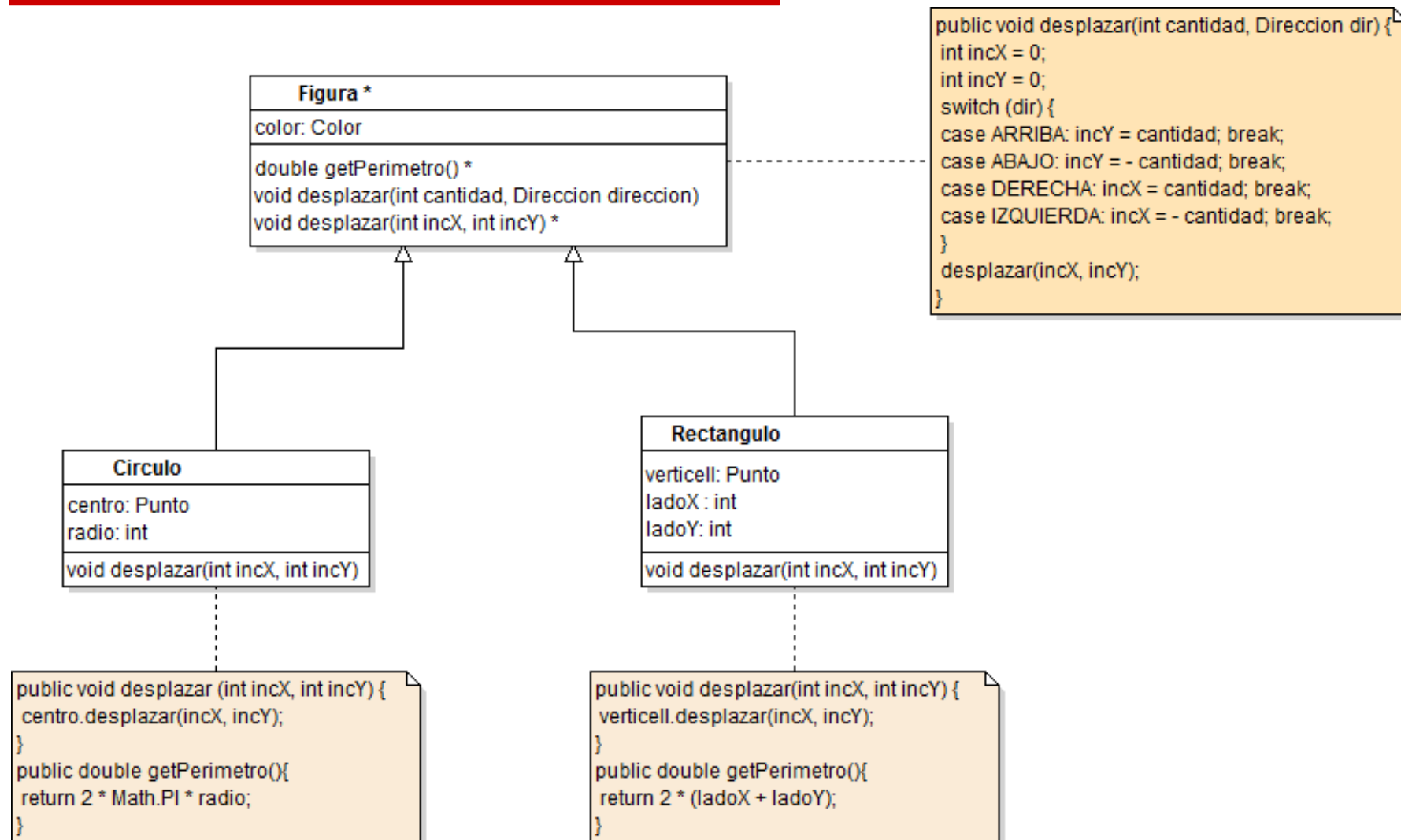
# Método plantilla en Figura

---

```
public void desplazar(int cantidad, Direccion dir) {  
    int incX = 0;  
    int incY = 0;  
    switch (dir) {  
        case ARRIBA: incY = cantidad; break;  
        case ABAJO: incY = - cantidad; break;  
        case DERECHA: incX = cantidad; break;  
        case IZQUIERDA: incX = - cantidad; break;  
    }  
    desplazar(incX, incY);  
}
```

```
public abstract void desplazar(int incX, int incY);
```

# Jerarquía de herencia



# Clases abstractas – Resumen

---

- ❑ **Una clase abstracta define un tipo**, como cualquier otra clase.
- ❑ Sin embargo, **no se pueden construir objetos** de una clase abstracta.
- ❑ Los **constructores** sólo tienen sentido para ser utilizados en las subclases.
- ❑ **Justificación de una clase abstracta:**
  - declara o hereda **métodos abstractos**
  - **y/o** representa un **concepto abstracto** para el que no tiene sentido crear objetos: publicación, figura geométrica, etc.

# Interfaces

---

- ❑ Construcción proporcionada por Java para la **definición de tipos** (sin implementación).
  - ❑ Una clase puede *heredar* de **cualquier número de interfaces**.
  - ❑ En este sentido, en lugar de “heredar”, se dice que una **clase implementa una interfaz**.
- ➔ Por tanto, el concepto de interfaz permite que una clase pueda ampliar su compatibilidad de tipos implementando múltiples interfaces, más allá de la compatibilidad con sus ancestros **limitada por la herencia simple**.

# Interfaz Atrapable

---

- ❑ Ejemplo: un elemento del juego es “atrapable” si podemos detenerlo y ponerlo en marcha cuando queramos.
- ❑ Se define este tipo de elementos como una interfaz.

```
public interface Atrapable {  
  
    void atrapar();  
    void liberar();  
}
```

- ❑ Los métodos de la interfaz son **abstractos**. No es necesario que se utilice el modificador abstract.

# Interfaz Atrapable

---

- ❑ Por defecto, en una interfaz **la visibilidad de las declaraciones es pública**. Además, sólo puede ser pública.
- ❑ Las interfaces pueden incorporar declaraciones (simplificadas) de **constantes**:

```
public interface Atrapable {  
  
    int LIMITE_ESPERA = 5000; // constante  
  
    void atrapar();  
    void liberar();  
}
```

# Implementación de una interfaz

---

- ❑ Ejemplo: sólo las burbujas limitadas y sensibles pueden ser *atrapables*.
- ❑ Las dos clases implementan la interfaz:

```
public class BurbujaLimitada extends Burbuja
    implements Atrapable {

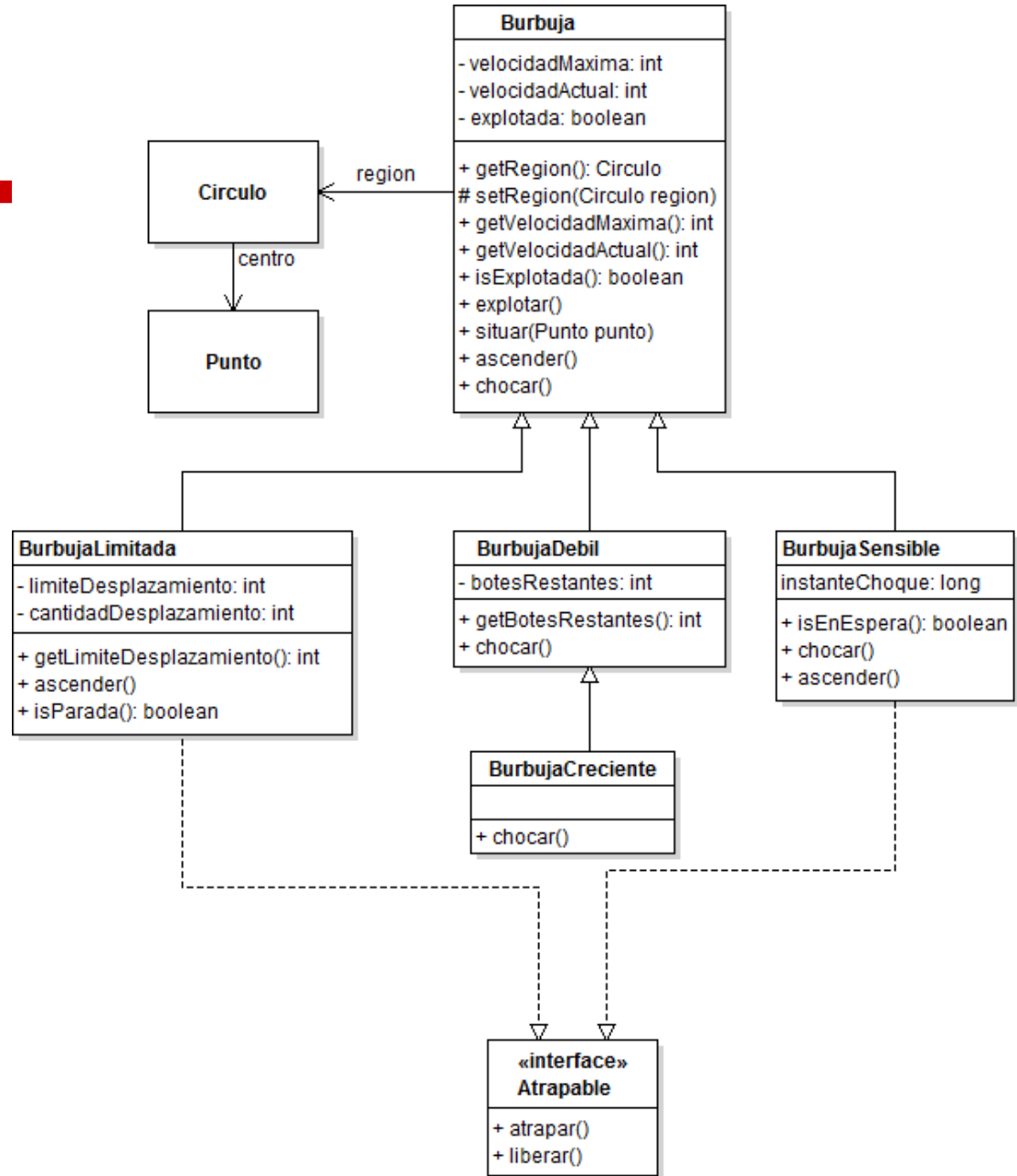
    ...

    @Override
    public void atrapar() {
        // ...
    }

    @Override
    public void liberar() {
        // ...
    }

}
```

# Interfaces





# Extensión de interfaces

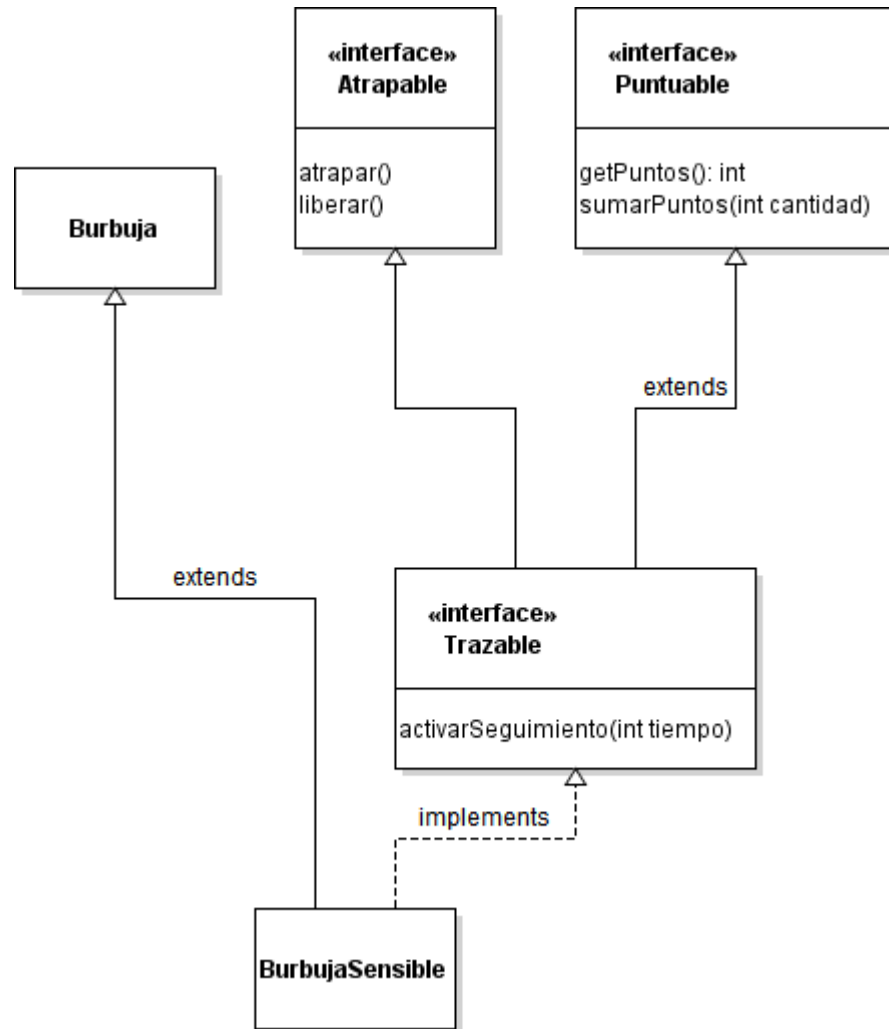
---

- Una interfaz puede **extender** otras interfaces:

```
public interface Trazable extends Atrapable, Puntuable {  
  
    void activarSeguimiento(int tiempo);  
  
}
```

- **Importante:** una clase solo puede heredar de una clase. En cambio, observa que una interfaz puede extender cualquier número de interfaces.

# Extensión de interfaces



# Aspectos clave de las interfaces

---

- Una **interfaz define un tipo** que puede ser utilizado para declarar variables.
- Sin embargo, **no se pueden construir objetos de una interfaz:**

```
Atrapable obj = new Atrapable(); // Error
```

- Los objetos asignables a una variable de tipo interfaz corresponden a clases efectivas (no abstractas) que implementan la interfaz:

```
Atrapable obj = new BurbujaLimitada(...);
```

# Aspectos clave de las interfaces

---

- Nótese que si una clase no implementa algún método de una interfaz debe declararse **abstracta**, ya que los métodos de las interfaces son abstractos.
- Las interfaces **resuelven la limitación de tipos impuesta por la herencia simple**: se puede implementar cualquier número de interfaces.

# Interfaces – métodos por defecto

---

- ❑ A partir de Java 8 es posible añadir **métodos implementados en una interfaz**. Estos métodos se denominan **método por defecto** o método de extensión
- ❑ Al igual que el resto de métodos de la interfaz se asume que un método por defecto es público.
- ❑ Una clase que implemente una interfaz con un método por defecto tiene dos opciones: 1) aceptar la implementación que ofrece la interfaz o 2) proporcionar otra implementación.
- ❑ Los métodos por defecto favorecen el mantenimiento de las interfaces.

# Interfaces – métodos por defecto

---

```
public interface Atrapable {  
    int TIEMPO_RETENCION = 1000;  
  
    void atrapar();  
    void liberar();  
  
    default void retener() {  
        atrapar();  
  
        Alarma.dormir(TIEMPO_RETENCION);  
  
        liberar();  
    }  
}
```

# Interfaces – métodos por defecto

---

- En el ejemplo anterior, el método `retener` sería equivalente a un *método plantilla* de una clase abstracta:
  - Un método implementado que se apoya en métodos abstractos.
  
- En general, un método por defecto puede contener cualquier código. No tiene la obligación de usar métodos de la interfaz.

# Interfaces – métodos *static*

---

- ❑ A partir de Java 8 es posible **implementar métodos *static*** en las interfaces.
- ❑ Se definen explícitamente con el modificador `static`.
- ❑ Al igual que el resto de métodos de la interfaz son `public`.
- ❑ Siempre se tienen que invocar utilizando el nombre de la interfaz.
- ❑ No existe colisión entre dos métodos `static` con la misma signatura implementados en dos interfaces diferentes.



# Interfaces – métodos *private*

---

- ❑ A partir de Java 9 es posible **implementar métodos `private`** en las interfaces, tanto de instancia como `static`.
- ❑ Los métodos privados mejorarán la reutilización de código.
- ❑ Los métodos sólo son accesibles dentro de la interfaz y no se puede acceder a ellos ni heredarlos de una interfaz a otra interfaz o clase.
- ❑ Lógicamente, el método de interfaz privado no puede ser abstracto.

# Interfaces- Ejemplo

---

```
interface Logging{
    default void logInfo(String message) {
        log(message, "INFO");
    }
    default void logError(String message) {
        log(message, "ERROR");
    }
    private void log(String message, String prefix) {
        System.out.println("Log message: " + prefix + " - " + message);
    }
}

final class OracleLogger implements Logging { }

public class PrivateInterfaceMethods {
    public static void main(String[] args) {
        OracleLogger logger = new OracleLogger();
        logger.logError("error en la conexión");
        logger.logInfo("Conexión establecida");
        logger.logError("consulta", "INFO"); //error
    }
}
```

# Interfaz Cloneable

---

- ❑ Es una interfaz de marca.
- ❑ No incluye la declaración de ningún método.
- ❑ Sirve para “marcar” una clase como copiable.
- ❑ La ejecución del método `clone` de la clase `Object` lanza una excepción si la clase del objeto que se quiere clonar no implementa la interfaz (`CloneNotSupportedException`).
- ❑ En una jerarquía de clases, sólo es necesario que la clase raíz implemente la interfaz.

# Interfaces vs. Clases abstractas

Interface Vs Abstract Class After Java 9		
	Interface	Abstract Class
Constructors	✗	✓
Static Fields	✓	✓
Non-static Fields	✗	✓
Static Final Fields	✓	✓
Non-final Fields	✗	✓
Private Fields	✗	✓
Protected Fields & Methods	✗	✓
Public Fields & Methods	✓	✓
Abstract methods	✓	✓
Static Methods	✓	✓
Non-static Methods	✓	✓
Final Methods	✗	✓
Non-final Methods	✓	✓
Default Methods	✓	✗
Private Methods	✓	✓
Private Static Methods	✓	✓

**Fuente:** <https://javaconceptoftheday.com/java-9-interface-private-methods/>