6. Interfaces

Abstracción e independencia de clases

© **Manuel Lama** (manuel.lama@usc.es)

Concepto de interfaz

- Los interfaces son uno de los componentes clave en el **diseño de los programas**, ya que con ellos se establecen de forma muy precisa los requisitos que deben cumplir las clases del programa
  - Se **definen los tipos de datos** que se deben crear en el programa, de modo que pueden existir otros tipos de datos, pero **al menos** deben estar definidos los indicados en los interfaces
  - Se definen **exactamente los métodos** que deben tener las clases del programa, indicando exactamente sus nombres, los tipos de argumentos que reciben y el tipo de datos que devuelven
  - Cada clase puede tener más métodos públicos y privados, pero los **métodos de interés** son los indicados en los interfaces

Concepto de interfaz

- Un interfaz se entiende como un **compromiso o acuerdo** entre programadores aplicación para que las clases que crean unos se puedan usar sin errores ni mayores adaptaciones por otros

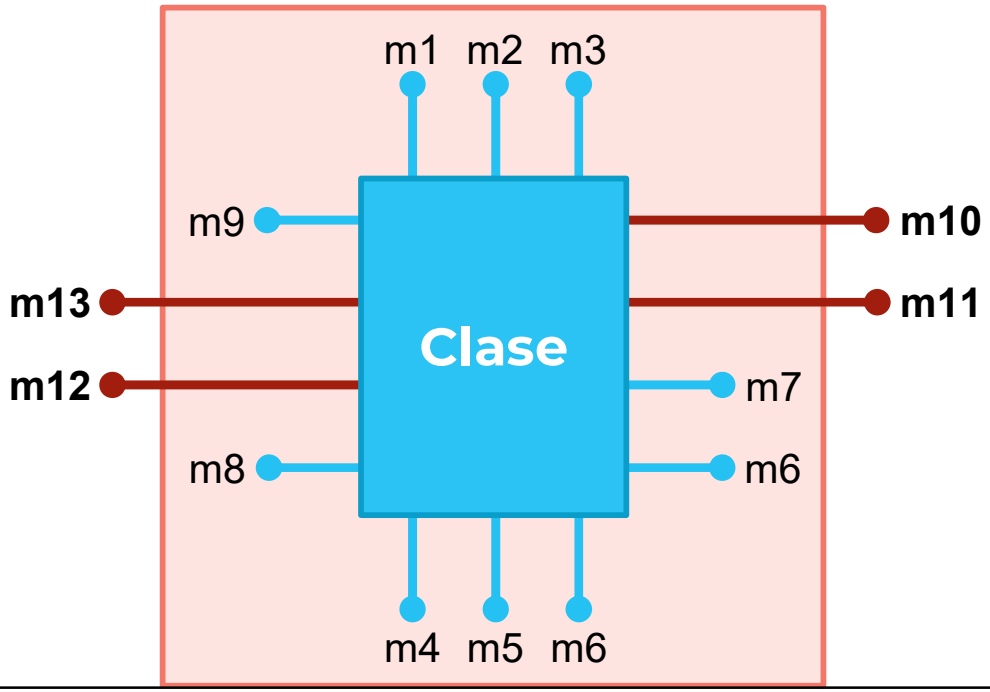
Programador 1: “En el interfaz se indica exactamente lo que necesito que hagan las clases que tienes que programar. No me interesa saber cómo vas a hacerlo, mientras lo hagas de la forma en la que se indica en el interfaz y que la implementación sea eficiente”

Programador 2: “¿Puedo crear otras clases de apoyo al interfaz que tengo que implementar? ¿Puedo crear otros métodos además de los que se indican en el interfaz?

Programador 1: “Puedes hacer lo que creas conveniente, mientras cumplas con el compromiso al que hemos llegado”

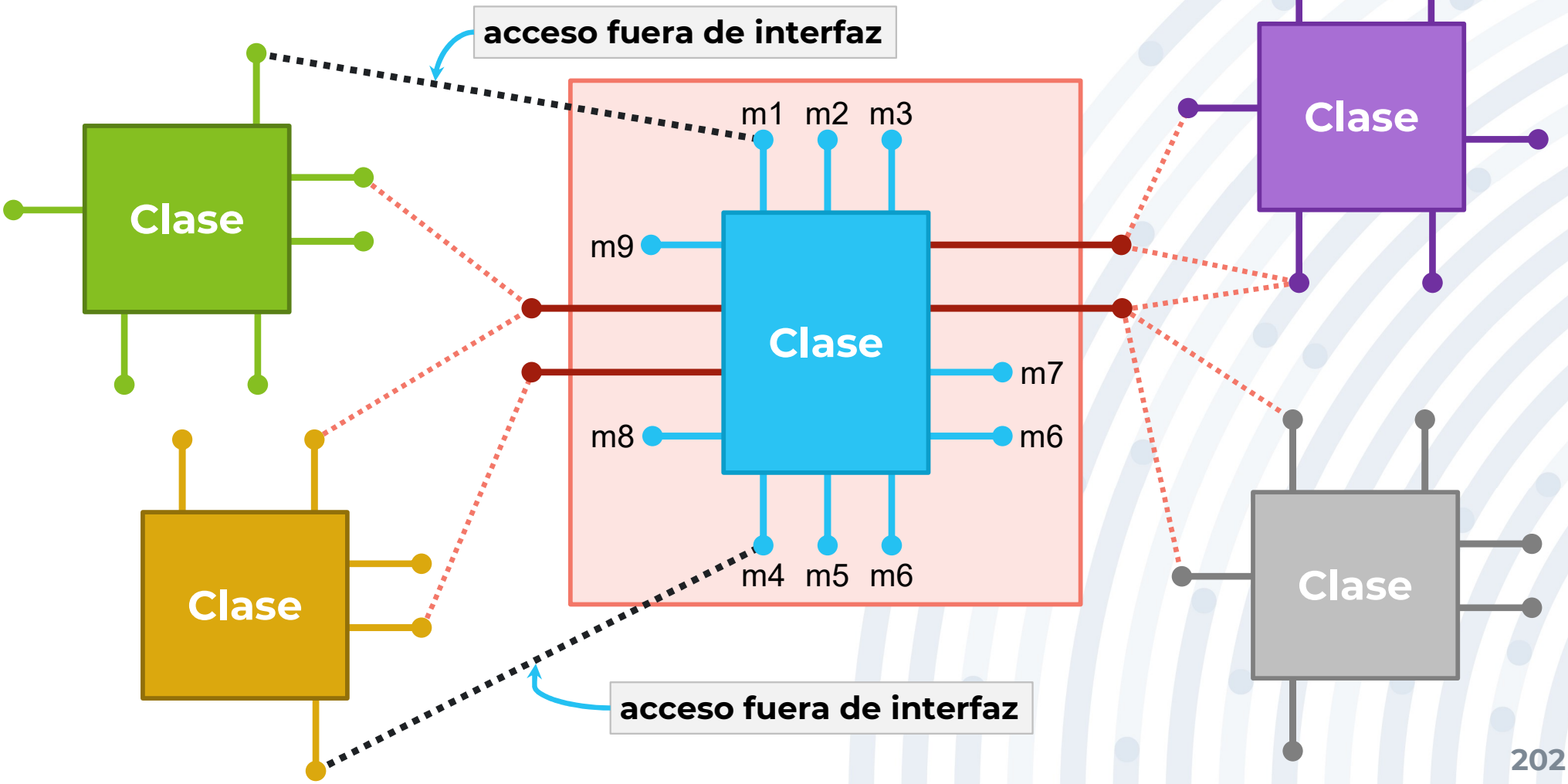
Concepto de interfaz

- El objetivo de un interfaz es establecer los métodos que una clase debe implementar y que serán los **métodos visibles y accesibles** por las otras clases del programa



- No solo se trata de los métodos públicos de las clases (**m1 al m13**)
- Se trata de **métodos públicos** que proveen la funcionalidad requerida por las otras clases del programa (**m10 al m13**)
- Los otros métodos **no son de interés** para las otras clases del programa (**m1 al m9**)

Concepto de interfaz



Concepto de interfaz

- El uso de interfaces facilita enormemente **el desarrollo y el mantenimiento de los programas**, puesto que con ellos se dividen las responsabilidades entre los programadores y permiten una implementación independiente de las clases
  - Un cambio en una clase en la que se implementan los métodos indicados en un interfaz **no afectará a las otras clases** que hace uso de dicho interfaz
  - Un cambio en el interfaz **afectará de forma significativa** tanto a las clases que implementan el interfaz como a las clases que lo usan
    - Los interfaces deberían ser invariantes durante el desarrollo de un programa; **una vez se definen, no deberían cambiar**

Interfaces en Java

- Un **interfaz** se entiende como una **plantilla de una clase** que contiene constantes, declaraciones de métodos abstractos, métodos por defecto y métodos estáticos
  - No se puede crear objetos de interfaces, con lo cual **no tienen constructores**
  - Las clases deben **implementar** los métodos abstractos del interfaz para que el interfaz pueda ser usado en otras clases

    public class <nombre_clase> ***implements*** *<nombre_interfaz>*
  - Una vez el interfaz haya sido implementado, se **comportará como una clase**, pudiendo invocar a los métodos abstractos, con la implementación de clase, los métodos estáticos y por defecto

Interfaces en Java

```
public interface <nombre_interfaz>
{
    // Constantes
    <tipo> <nombre_atrib> = <valor>;

    // Declaraciones de métodos (signaturas)
    <tipo> <nombre_método> (<tipo> <nombre> *);

    // Métodos por defecto
    default public <tipo> <nombre_método> (<tipo> <nombre> *) {
        <código>
    }

    // Métodos estáticos
    static public <tipo> <nombre_método> (<tipo> <nombre> *) {
        <código>
    }
}
```

interfaz clásico

añadidos en Java

Los atributos de un interfaz son implícitamente **públicos**, **estáticos** y **finales** (son constantes)

Los métodos de un interfaz son implícitamente **públicos** y **abstractos**

Los métodos **por defecto** solo se pueden definir en los interfaces y **son heredados** directamente por las clases que los implementan

Los métodos estáticos también se implementan en el interfaz y son accesibles **directamente** desde cualquier clase

Interfaces en Java

```java
public class EmpleadoImpl implements Empleado {

    @Override
    public float calcularSueldo() {
        float sueldo= this.base;
        float factor= (this.antiguedad > 15) ? 0.02f : 0.01f;
        for(int i=0;i<this.proyectosParticipado.size();i++) {
            Proyecto proyecto= this.proyectosParticipado.get(i);
            sueldo+= factor*proyecto.getPresupuesto();
        }
        return sueldo;
    }

    @Override
    public Proyecto getProyecto(String nombreProyecto) {
        for(int i=0;i<this.proyectosParticipado.size();i++) {
            Proyecto proyecto= this.proyectosParticipado.get(i);
            if(proyecto.getNombre().equals(nombreProyecto))
                return proyecto;
        }
        return null;
    }

    @Override
    public ArrayList<Proyecto> getProyectos(float minimo) {
        ArrayList<Proyecto> proyectosPMinimo= new ArrayList<>();
        for(int i=0;i<this.proyectosParticipado.size();i++) {
            Proyecto proyecto= this.proyectosParticipado.get(i);
            if(proyecto.getPresupuesto()>minimo)
                proyectosPMinimo.add(proyecto);
        }
        return proyectosPMinimo;
    }
```

implements

@Override

@Override

@Override

El método **getProyecto** obtiene un proyecto a partir de su nombre

```java
public interface Empleado {
    // Constantes
    float SUELDO_MAXIMO= 80000f;
    int MAXIMO_EMPLEADOS= 50;

    // Métodos abstractos
    float calcularSueldo();

    Proyecto getProyecto(String proyecto);

    ArrayList<Proyecto> getProyectos(float minimo);
}
```

El método **getProyectos** obtiene los proyectos con un presupuesto mayor de **minimo**

Interfaces en Java

- Si una clase implementa un interfaz, entonces esa clase tendrá los mismos métodos que el interfaz, **del mismo modo** que una clase derivada tiene los mismos métodos que una clase base
  - La clase que implementa el interfaz se puede entender que **es como una clase derivada del interfaz**, puesto que un interfaz es **equivalente** a una clase abstracta con las siguientes características
    - Tiene todos sus métodos abstractos
    - Tiene todos sus atributos constantes
    - No tiene constructores implementados
  - Se pueden aplicar **todos los conceptos** de herencia, jerarquías de clases, clases abstractas y polimorfismo

Clases abstractas vs Interfaces

Clases abstractas	Interfaces
Se definen cuando las clases derivadas tienen algunos métodos comunes	Se definen cuando las clases que los implementan tienen diferentes implementaciones para diferentes objetos
La herencia múltiple no es posible	Se puede entender que la herencia múltiple es posible, aunque solo puede haber una clase base
Pueden tener métodos abstractos y métodos concretos	Solamente tienen métodos abstractos, además de métodos estáticos y métodos por defecto
Los métodos abstractos pueden ser públicos y protegidos	Los métodos abstractos deben de ser públicos
Tienen constructores	No tienen constructores
Pueden tener cualquier tipo de atributo	Los únicos atributos que puede tener son de tipo estático y final (son constantes)

Interfaces en Java

```java
public class Empresa {
    private ArrayList<Empleado> empleados;

    public ArrayList<Empleado> getEmpleados() {
        return empleados;
    }

    public void setEmpleados(ArrayList<Empleado> empleados) {
        this.empleados = empleados;
    }

    public ArrayList<Empleado> empleadosEnProyecto(String proyecto) {
        ArrayList<Empleado> empleadosEnProyecto= new ArrayList<>();
        for(Empleado empleado : empleadosEnProyecto)
            if(empleado.getProyecto(proyecto)!=null)
                empleadosEnProyecto.add(empleado);
        return empleadosEnProyecto;
    }
```

En **ninguna parte** de Empresa se usa la clase **EmpleadoImpl**, sino que se usa el interfaz **Empleado**:

- **Lo importante** son los métodos, ya que las clases externas a una clase no deberían de poder acceder a los atributos
- Se podría cambiar incluso la clase con la que se implementa el interfaz **sin necesidad de cambiar** la clase Empresa

El método **getProyecto** que se invoca tiene la implementación realizada en la clase EmpleadoImpl. **Si se desea cambiar dicha implementación, no será necesario modificar la clase Empresa**

Métodos por defecto

- El principal problema de los interfaces es la incapacidad de las clases que los implementan para **adaptarse a los cambios** que tengan lugar en ellos
  - Al añadir nuevos métodos abstractos y/o actualizar los métodos que están declarados en un interfaz, las clases que implementan dicho interfaz **generarán errores de compilación**
    - Existirán casos en el programa en las cuales **no** se usarán los nuevos métodos del interfaz, mientras que en otras **sí** que será necesario utilizarlos
- Los **métodos por defecto** resuelven parcialmente el problema de adaptación de las clases a los cambios en los interfaces que implementan

Métodos por defecto

- Los métodos por defecto son métodos que se deben **declarar e implementar** en el interfaz, de modo que se **heredarán** por las clases que implementan dicho interfaz

```
// Métodos por defecto
default public <tipo> <nombre_método> (<tipo> <nombre> *) {
    <código>
}
```

  - La implementación debe operar únicamente con los **argumentos** del método por defecto, ya que en el interfaz no existen atributos que no sean constantes
  - Los métodos por defecto **pueden ser sobrescritos** en las clases que implementan el interfaz o en las clases derivadas de ellas

Métodos por defecto

```
public interface Empleado {
    // Constantes
    float SUELDO_MAXIMO= 80000f;
    int MAXIMO_EMPLEADOS= 50;

    // Métodos abstractos
    float calcularSueldo();
    Proyecto getProyecto(String proyecto);
    ArrayList<Proyecto> getProyectos(float minimo);

    // Método por defecto
    default ArrayList<Proyecto> getProyectos(ArrayList<Proyecto> proyectos,
                                             String tipo) {
        ArrayList<Proyecto> proyectosTipo= new ArrayList<>();
        for(Proyecto proyecto : proyectos)
            if(proyecto.getTipo().equals(tipo))
                proyectosTipo.add(proyecto);
        return proyectosTipo;
    }
}
```

Uno de los argumentos del método **getProyectos** es una lista de proyectos en los que ha participado un empleado, pero ese argumento en realidad es uno de los atributos de la clase Empleado

Como no se puede acceder a los atributos de empleado, ese atributo se deberá de pasar como argumento al método por defecto

Los métodos por defecto son implícitamente públicos. Además, desde estos métodos **se pueden invocar** a los métodos abstractos de interfaz, los cuales se implementarán en las clases que implementan al interfaz

Métodos por defecto

- Se debe de valorar la **conveniencia** de definir métodos por defecto, sobre todo si necesitan **argumentos asociados a atributos** de las clases que implementan los interfaces que contienen dichos métodos
  - El código se hace **más complejo de entender y mantener**, puesto que los métodos de las clases tienen argumentos que no serían necesarios al hacer referencia a los atributos de dichas clases

```
EmpleadoImpl emp= new EmpleadoImpl("ED2", 14);
ArrayList<Proyecto> proyectosEmp= emp.getProyectosParticipado();
ArrayList<Proyecto> proyectoTipo= emp.getProyectos(proyectosEmp, "IA");
```

proyectosEmp es una referencia al atributo **proyectosParticipado** de la clase EmpleadoImpl. Esa referencia se pasa como argumento al método **getProyectos** de la propia clase, lo cual **no tiene mucho sentido**

Métodos por defecto

- Los métodos por defecto **serán heredados** por las clases que implementan al interfaz que los contiene y por los interfaces derivados de dicho interfaz
  - Los métodos por defecto, al ser heredados, también **podrán ser sobrescritos** por las clases y los interfaces derivados
  - En la sobreescritura se puede **hacer uso de super** siguiendo el siguiente formato,

*<nombre_interfaz>**.super.***<método>*

La implementación de getProyectos no tiene mucho sentido, ya que **no hace uso del argumento proyectos**

```
public ArrayList<Proyecto> getProyectos(ArrayList<Proyecto> proyectos,
                                        String tipo) {
    return Empleado.super.getProyectos(this.proyectosParticipado, tipo);
}
```

Métodos estáticos

- Los métodos estáticos son métodos que pueden ser invocados desde el inicio del programa, de modo que **para invocarlos no es necesario** crear ningún objeto de la clase en la que se han definido
  - Se pueden definir en **cualquier clase**, sea abstracta, instanciable o final, o en **cualquier interfaz**
  - Se cargan **automáticamente en memoria** al arrancar el programa
  - En un método estático **únicamente se pueden invocar métodos** de la clase o del interfaz que sean estáticos, puesto que tienen que estar disponibles desde el arranque del programa
  - **No son heredados** por las clases e interfaces derivados de ellos

Métodos estáticos

```
public interface Empleado {
    // Constantes
    float SUELDO_MAXIMO= 80000f;
    int MAXIMO_EMPLEADOS= 50;

    // Métodos abstractos
    float calcularSueldo();
    Proyecto getProyecto(String proyecto);
    ArrayList<Proyecto> getProyectos(float minimo);

    // Método por defecto
    default ArrayList<Proyecto> getProyectos(ArrayList<Proyecto> proyectos,
                                             String tipo) {...7 lines }

    // Métodos estáticos
    public static float beneficios(ArrayList<Float> presupuestos) {
        float beneficios =0f;
        for(Float presupuesto : presupuestos)
            beneficios+= (presupuesto<100000) ? 0.001*presupuesto
                                              : 0.0001*presupuesto;
        return beneficios;
    }
}
```

beneficios es un método que calcula el beneficio para un empleado por haber participado en proyectos, recibiendo como argumento el conjunto de los presupuestos como un objeto de tipo ArrayList<Float>

Como es un método estático **beneficios** podrá ser invocado en cualquier momento y en cualquier clase siguiendo la forma,

Empleado.beneficios(...)

Métodos estáticos vs por defecto

Métodos por defecto	Métodos estáticos
Solamente pueden estar definidos en los interfaces	Se pueden definir en interfaces y en clases
Pueden invocar cualquier tipo de método, incluyendo métodos por defecto, estáticos y abstractos	Solamente pueden invocar métodos que, a su vez, sean métodos estáticos
Son heredados por las clases y por los interfaces derivados del interfaz que los contiene	No pueden ser heredados, siendo métodos que son específicos de las clases o de los interfaces en los que están implementados
No están disponibles en el arranque del programa, es necesario crear un objeto para invocarlos	Están disponibles al arrancar el programa

Herencia e interfaces

Una clase que implementa una interfaz es como una clase derivada de dicho interfaz

Una clase puede implementar más de un interfaz

Se puede entender que existe **herencia múltiple**

- Si una clase abstracta cumple las **mismas condiciones** que un interfaz “clásico”, es decir, solamente tiene métodos abstractos y no tiene ningún atributo que no sea constante, entonces se podría pensar que una **clase derivada tiene varias clase base**, o lo que es lo mismo, que existe herencia múltiple

Herencia e interfaces

- Una clase debe de **implementar todos los métodos abstractos** de todos los interfaces que implementa y **hereda todos los métodos por defecto** de todos esos interfaces
  - Si varios interfaces tienen en **común algún método abstracto**, la implementación de ese método en la clase **será válida** para todos los interfaces
  - Si varios interfaces tienen en **común algún método estático**, no existirá **ningún conflicto** en la clase, ya que ese tipo de métodos no serán heredan
  - Si varios interfaces tienen en **común algún método por defecto**, existirá una **colisión entre dichos métodos** en relación a cuál de ellos será el que heredará la clase

Herencia e interfaces

- ¿Cómo se **resuelve la herencia de los métodos por defecto** cuando existen varios interfaces base, es decir, cuando una clase implementa varios interfaces?
  - Para resolver el conflicto, la clase que implementa los interfaces **deberá de sobrescribir** los métodos por defecto que son comunes a dichos interfaces
    - **No existe** ningún mecanismo automático para seleccionar cuál es la implementación de los métodos por defecto que heredan las clases
    - Realmente **en la propia sobreescritura** de los métodos por defecto se puede realizar la selección de la implementación de los métodos por defecto, **utilizando para ello super**

Herencia e interfaces

```java
public interface EmpleadoBase {
    // Método por defecto
    default ArrayList<Proyecto> getProyectos(ArrayList<Proyecto> proyectos,
                                             String tipo) {
        ArrayList<Proyecto> proyectosTipo= new ArrayList<>();
        for(Proyecto proyecto : proyectos) {
            String tipoProyecto= proyecto.getTipo();
            if(tipoProyecto.equals(tipo) && !tipoProyecto.equals("IA")) {
                System.out.println("Proyecto -> " + proyecto.getNombre());
                proyectosTipo.add(proyecto);
            }
        }
        return proyectosTipo;
    }
```

```java
public interface Empleado {
    // Método por defecto
    default ArrayList<Proyecto> getProyectos(ArrayList<Proyecto> proyectos,
                                             String tipo) {
        ArrayList<Proyecto> proyectosTipo= new ArrayList<>();
        for(Proyecto proyecto : proyectos)
            if(proyecto.getTipo().equals(tipo))
                proyectosTipo.add(proyecto);
        return proyectosTipo;
    }
```

EmpleadoImpl debe implementar dos interfaces **EmpleadoBase** y **Empleado**

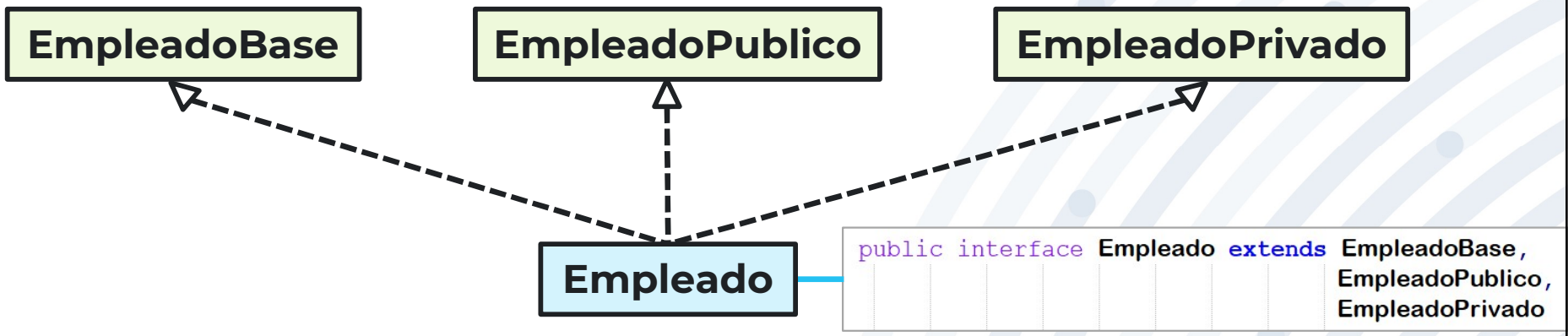
```java
public class EmpleadoImpl implements Empleado, EmpleadoBase {
    @Override
    public float beneficios(ArrayList<Float> presupuestos) {
        return EmpleadoBase.super.beneficios(presupuestos);
    }
```

La clase **EmpleadoImpl** tiene que implementar **getProyectos**, al ser un método común a los interfaces **EmpleadoBase** y **Empleado**. Con el uso de super, en realidad está eligiendo **la implementación de los interfaces que se prefiere**, la del interfaz **EmpleadoBase**

Herencia e interfaces

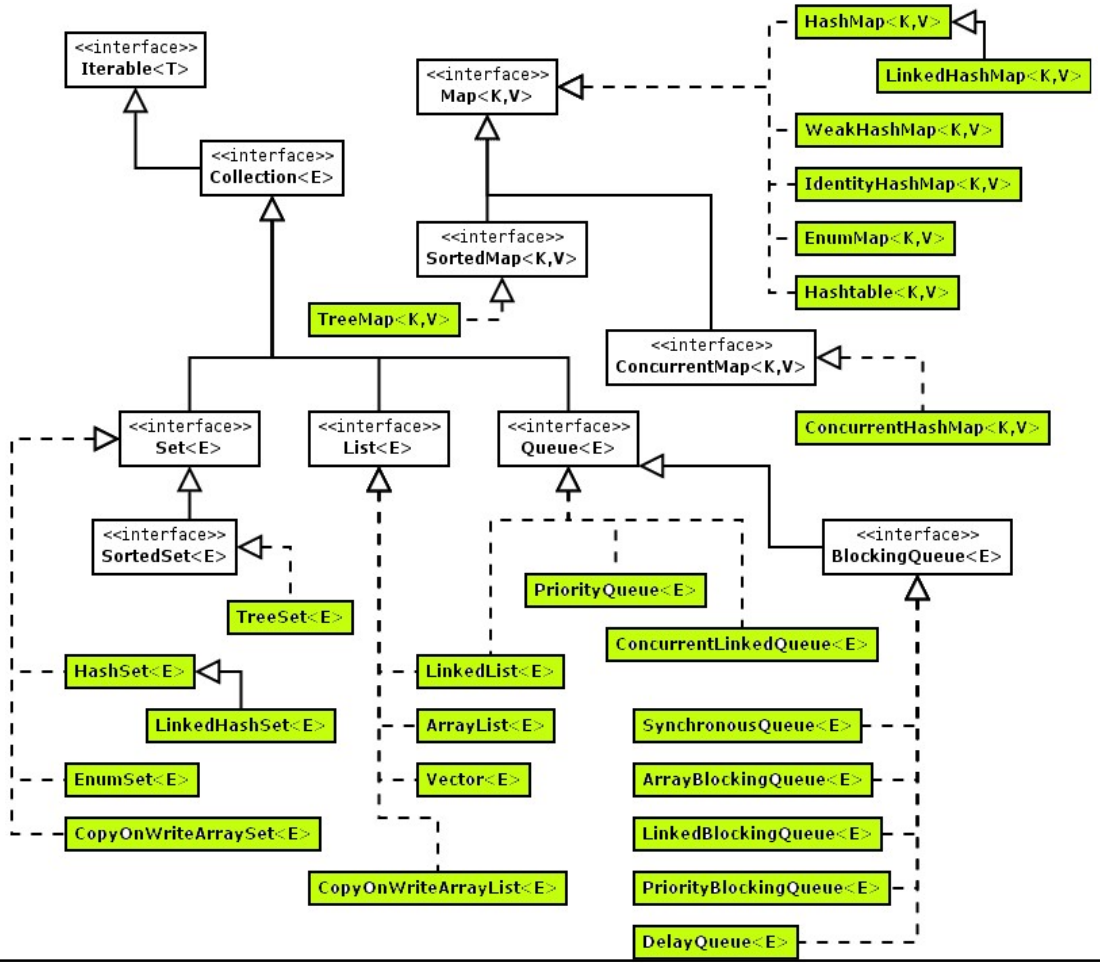
- Se pueden crear **jerarquías de interfaces** siguiendo las mismas reglas de herencia que en el caso de las jerarquías de clases
  - Un interfaz derivado puede tener varios interfaces base, es decir, **existe herencia múltiple entre interfaces**
  - Puesto que en un interfaz tiene todos sus métodos implícitamente públicos, los interfaces derivados **heredarán** todos los métodos de los interfaces base, **excepto los métodos estáticos**
  - Existe **sobreescritura de los métodos por defecto**, de manera que los interfaces derivados puede usar super para invocar la ejecución de los métodos por defecto de los interfaces base
  - No se puede establecer herencia entre clases e interfaces, es decir, **una clase no puede extender un interfaz ni viceversa**

Herencia e interfaces



- Si varios interfaces base tienen en **común algún método por defecto**, existirá una **colisión entre dichos métodos** en relación a cuál de ellos será el que heredará el interfaz derivado
  - La forma de resolver este conflicto **es la misma** que en el caso de las clases que implementan varios interfaces que tienen algún método por defecto en común

Herencia e interfaces



- Las jerarquías **combinan**
  1. La herencia de clases
  2. La herencia de interfaces
  3. La implementación de interfaces por parte de las clases
- Las jerarquías resultantes pueden ser muy complejas pero lo importante es que **sean extensibles** de forma relativamente sencilla

Interfaces

- **Buenas prácticas de programación** (**XXX**)

  Se deben definir interfaces cuando se quieren establecer de forma clara e inequívoca los métodos de que se van a usar en las restantes clases del programa

- **Buenas prácticas de programación** (**XXXI**)

  Se deben usar clases abstractas cuando se quiere hacer énfasis en la reutilización de código, incluyendo constructores que se invocan desde las clases base