Constructores de las subclases (I)

- Los constructores no se heredan.

- Cuando se crea un objeto de una subclase, se invoca a su constructor, que a su vez invoca implícitamente al constructor sin parámetros de su superclase, y su superclase hará lo mismo, y así sucesivamente, es decir, se ejecutan primero los constructores de sus superclases y por último el propio constructor.

  Ejemplo:

  - Se define la clase *A* con un constructor sin parámetros que muestra el mensaje " Constructor de la clase A" cuando es invocado.
  - Se define la subclase *AB* que hereda de la clase *A* y que define un constructor sin parámetros que muestra el mensaje " Constructor de la clase AB" cuando es invocado.
  - Se define la subclase *ABC* que hereda de la clase *AB* y que define un constructor sin parámetros que muestra el mensaje " Constructor de la clase ABC" cuando es invocado.
  - Se crea un objeto de la clase *ABC* y se observan los mensajes.

Constructores de las subclases (II)

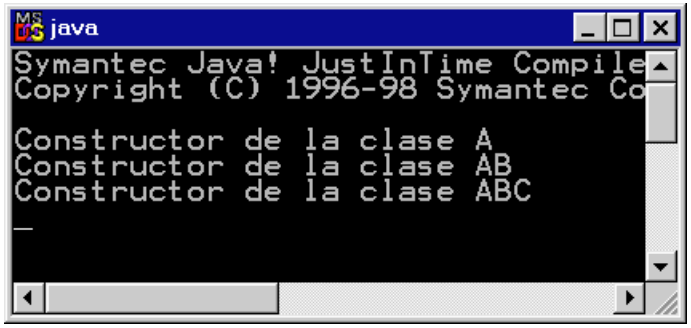

```
public class A
{
    public A () { System.out.println("Constructor de la clase A");}
}

public class AB extends A
{
    public AB () { System.out.println("Constructor de la clase AB");}
}

public class ABC extends AB
{
    public ABC () { System.out.println("Constructor de la clase ABC");}
}

public class MainClass
{
    public static void main(String [] args)
    {
        ABC objetoABC = new ABC();
    }
}
```

Definición de la superclase *A*

Definición de la subclase *AB*

Definición de la subclase *ABC*

Creación de un objeto de la subclase *ABC*



Constructores de las subclases (III)

- La **invocación implícita del constructor** de la superclase se realiza siempre **sobre el constructor sin parámetros**, y por lo tanto, es necesario que la superclase disponga de este tipo de constructor, y esto ocurre en dos casos:
  - la clase no define constructores, y por lo tanto el compilador le proporciona un constructor por defecto (sin parámetros).
  - La clase define explícitamente un constructor por defecto.

- También se puede **invocar de manera explícita** al constructor de la superclase mediante la palabra reservada **super**.
  La llamada al constructor de la superclase utilizando **super** debe ser la primera línea del código del constructor de la subclase.

```
identificadorSubclase (parámetrosConstructorSubclase)
{
        super(parametrosConstructorSuperclase);
        // cuerpo del constructor de la subclase
}
```

Destructores de las subclases (I)

- Los destructores (métodos **finalize**) de las superclases **no** son invocados automáticamente como ocurre con los constructores.
  La invocación se debe hacer de manera explícita, utilizando la palabra reservada **super**.
  El lugar más idóneo para invocar el destructor de una superclase es la última línea del destructor de la subclase.

- **Ejemplo:**
  - Se define la clase *X* y se implementa su destructor de manera que muestra el mensaje " Finalize clase X" cuando es invocado.
  - Se define la subclase *XY* que hereda de la clase *X* y se implementa su destructor para que muestre el mensaje " Finalize clase XY" cuando es invocado.
  - Se crea un objeto de la clase *XY* y después se asigna a su referencia el valor **null** para convertirlo en "basura" que recogerá el recoletor en la llamada forzada que se hace desde el main. Se observan los mensajes.

Destructores de las subclases (II)


```
class X{
    protected void finalize()  { System.out.println("Finalize de la clase X");}
}
```

→ Definición de la superclase *X*

```
class XY extends X{
    protected void finalize()
     {
        System.out.println("Finalize de la clase XY");
        super.finalize();
}}
```

→ Definición de la subclase *XY*


```
public class ClaseMain{
    public static void main(String [] args)
    {
        XY objetoXY = new XY();
        objetoXY = null;
          System.runFinalization();
          System.gc();
}}
```

→ Es posible que no sea necesario llamar a runFinalization, System.gc() puede ser suficiente

Concepto de interfaz (I)

- En **castellano**, una *interfaz es un dispositivo o sistema que utilizan entidades inconexas para interactuar*.

- En **Java**, también una interfaz es un mecanismo que permite interactuar a objetos no relacionados entre sí

- En Java, una interfaz es un *conjunto de declaraciones de métodos sin implementación y constantes, que son implementados posteriormente por las clases*.

- Por lo tanto, en Java, una interfaz constituye *un protocolo de comportamiento* (*declaraciones de métodos y constantes*) que *todas las clases que lo implementen deben definir* (código de los métodos) y *que comparten de cara al exterior*.

- Se consigue así **uniformizar** la comunicación entre ciertas clases.

Concepto de interfaz (I)

- Una **clase que implementa** una interfaz **hereda las declaraciones** de los métodos y **debe proporcionar una implementación** para todos ellos (**sólo** es posible la **omisión** de la implementación de uno, alguno o todos los métodos **si la clase que implementa** la interfaz **es abstracta**).

- Las constantes de la interfaz son accesibles directamente desde las clases que la implementan. Desde clases que no la implementan se puede acceder con la notación .

Concepto de interfaz (II)

- El **objetivo** de una interfaz es **definir un protocolo**.

- **Ejemplo:** Por ejemplo, se puede definir la interfaz "Dibujable" con los métodos "fijarPosición" y "dibujar", y todas las clases que implementen esta interfaz mostrarán al exterior la capacidad de fijar su posición y de dibujarse.

- Una **clase puede implementar una o varias interfaces**.
  - Para especificar que una clase implementa una o más interfaces se utiliza la palabra reservada implements seguida de los nombres de las interfaces separadas por comas.

```
class nombreClase implements interface1, interface2
```

- En la declaración de una clase, la cláusula implements es posterior a la cláusula extends.

Definición de interfaz (I)

- Sintaxis de declaración de una interfaz

[Modificadores] interface Identificador [SuperInterfaces] CuerpoInterface

- Modificadores: los únicos modificadores que puede tener una interfaz son: **public** y **abstract**.
  - **public**
    - una inteface declarada **public** es accesible desde fuera de su paquete. Si se omite este modificador, solamente es accesible desde el paquete donde está definida.
      Cada interfaz **public** debe ser definida en un fichero .java con el mismo nombre de la interfaz.
      No es posible definir en un mismo fichero .java una interfaz pública y una clase pública.
  - **abstract**
    - una interfaz es implícitamente abtracta. El modificador **abstract** está permitido pero no es obligatorio.

Definición de interfaz (II)

- Identificador
  - Es el nombre de la interfaz. Debe ser un identificador válido en Java.

- Superinterfaces
  - Opcional, especifica que la interfaz es una extensión de otras. Una interfaz que extiende a otra, hereda todos sus métodos y constantes.
  - Sintaxis: **extends IdentificadoresSuperInterfaces**
  - Una interfaz puede "extender" a varias interfaces, cuyos identificadores se especifican separados por comas a continuación de la palabra reservada extends.

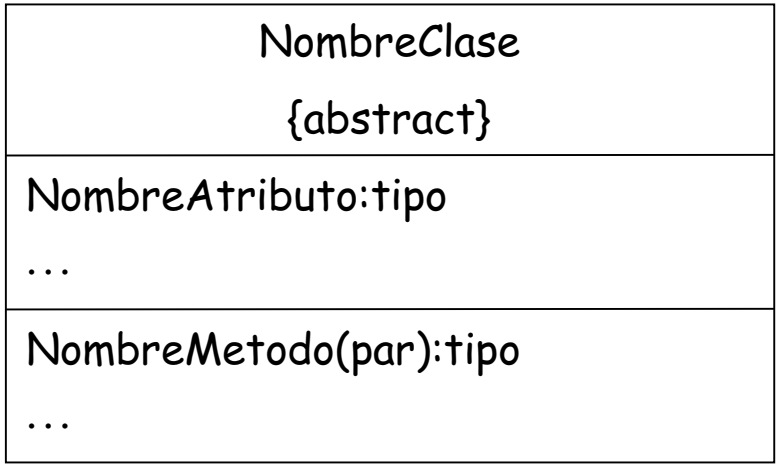
- Cuerpo de la interfaz
  - Formado por una lista de declaraciones de constantes y de métodos (sin implementación).
    - Todos los métodos son implícitamente public y abstract.
    - Todas las constantes son implícitamente public, final y static.
    - El uso de modificadores es opcional.

interfaz vs. clase abstracta (I)

- Una **interfaz** no **puede implementar ningún método**, y una clase abstracta sí puede.
- Una **clase puede implementar muchas interfaces** pero **sólo puede tener una superclase** directa.
- Una interfaz **no es parte de la jerarquía de clases**. Clases "inconexas" pueden implementar la misma interfaz.
- Una **interfaz puede extender múltiples interfaces**.

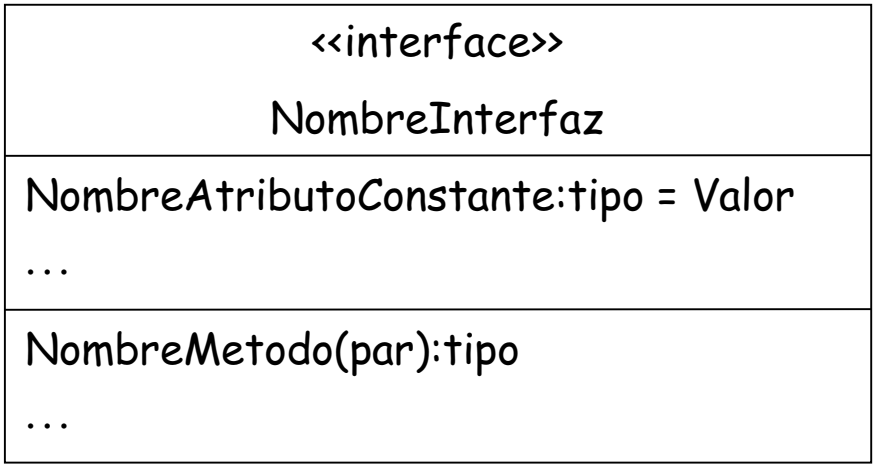
interfaz vs. clase abstracta (II)

- Notación UML para clases abstractas



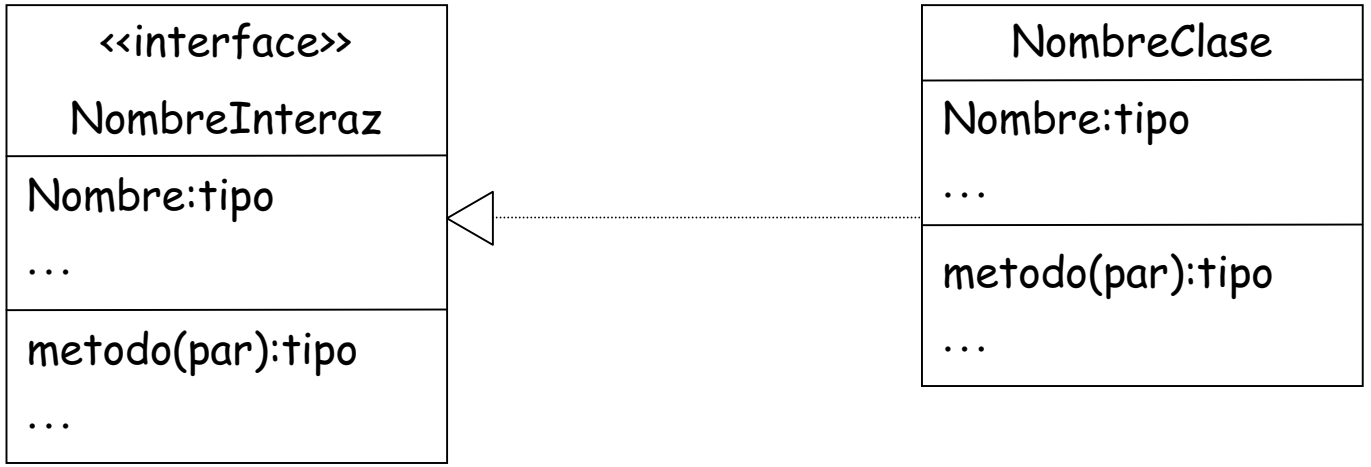
interfaz vs. clase abstracta (II)

- Notación UML para interfaces

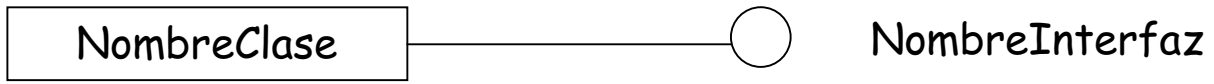


interfaz vs. clase abstracta (II)

- Notación UML para clases que implementan interfaces (relación de **realización** o **implementación**)
  - De manera "explícita"

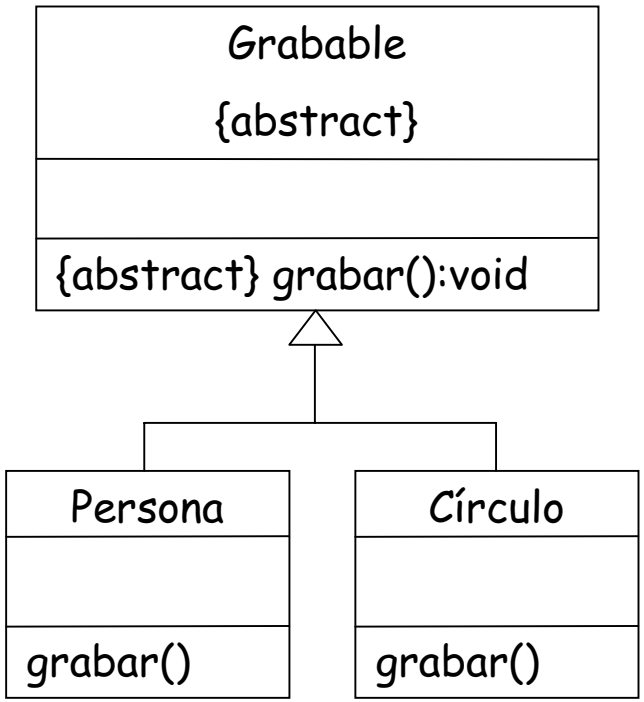


  - De manera "implícita"



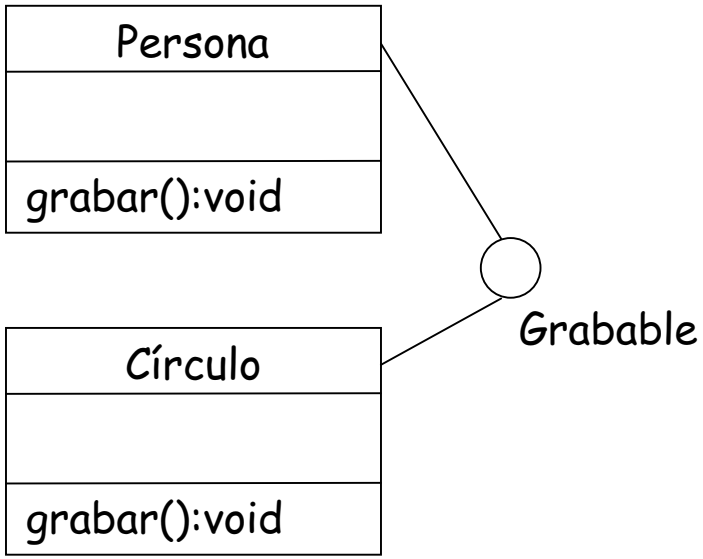
interfaz vs. clase abstracta (II)

- **Ejemplo:** Las clases "Persona" y "Circulo" semánticamente no pertenecen a la misma jerarquía. Para que ambas clases muestren el comportamiento de poder grabar su estado en un fichero se puede hacer de dos maneras:
  - forzar a las clases a que pertenezcan a la misma jerarquía creando la superclase común "Grabable"

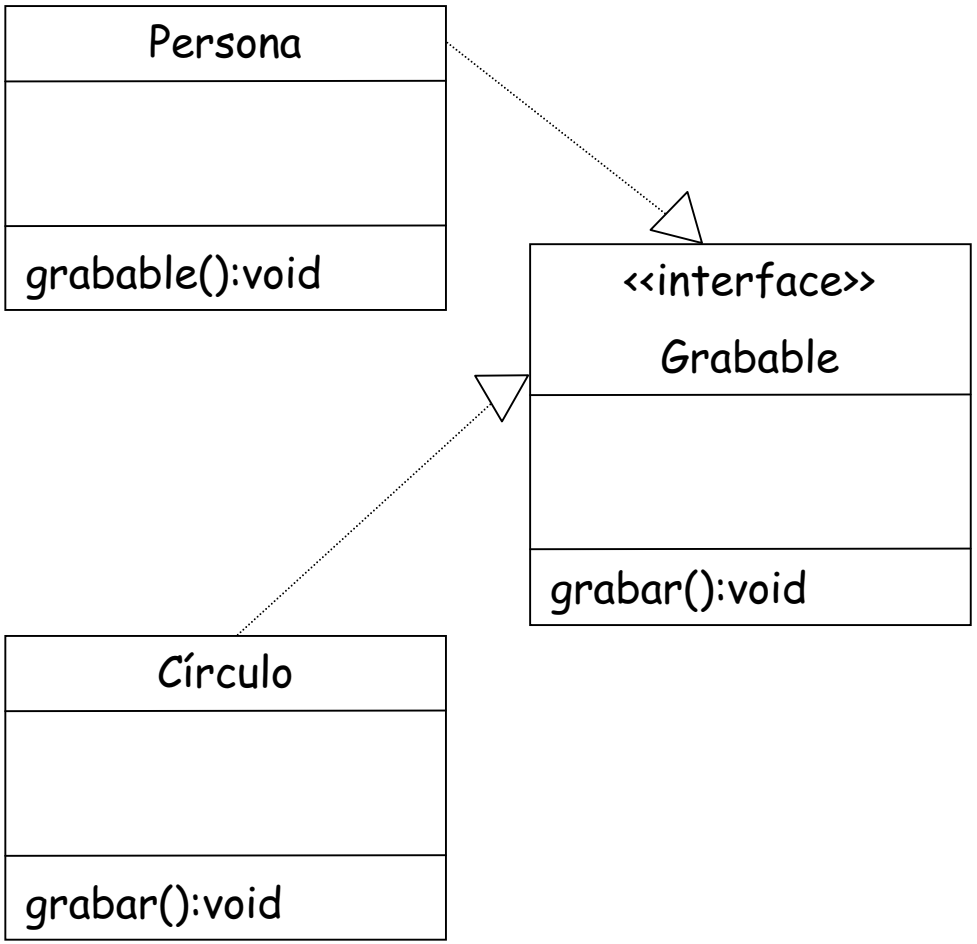


interfaz vs. clase abstracta (II)

- definir la interfaz "Grabable" y que ambas clases la implemenen.



De manera implícita



De manera implícita

Interfaces y polimorfismo (I)

- En cuanto al **polimorfismo**, las **referencias de un tipo interfaz** se comportan de la misma manera que las referencias a las clases abstractas. Una **referencia a una interfaz espera contener** una **referencia a cualquier clase que implemente dicha interfaz**, y en tiempo de ejecución el polimorfismo funciona de la misma manera que en las clases abstractas.

- **Ejemplo:**
  - Se define una interfaz *LaInterface* con dos métodos *metodo1* y *metodo2*.
  - Se definen tres clases inconexas *Clase1*, *Clase2* y *Clase3* que implementan *LaInterface*.
  - Se define un vector de referencias a la interfaz, y se asignan a los elementos del vector referencias a objetos de las tres clases (es posible porque implementan la misma interfaz) y se comprueba el polimorfismo.

Interfaces y polimorfismo (II)


```
public interface LaInterface
{
   public abstract String metodo1();
   public abstract String metodo2();
}
```

Definición de *LaInterface*

```
public class Clase1 implements LaInterface
{
     public String metodo1()  { return "metodo1 de Clase1"; }
     public String metodo2()  { return "metodo2 de Clase1"; }
}
```

Definición de *Clase1*

```
public class Clase2 implementes LaInterface
{
     public String metodo1()  { return "metodo1 de Clase2"; }
     public String metodo2()  { return "metodo2 de Clase2"; }
}
```

Definición de *Clase2*

```
public class Clase3 implementes LaInterface
{
     public String metodo1()  { return "metodo1 de Clase3"; }
     public String metodo2()  { return "metodo2 de Clase3"; }
}
```

Definición de *Clase3*

Interfaces y polimorfismo (III)


```
public class MainClass
{
    public static void main(String [] args)
    {
        LaInterface[] vector = new LaInterface[3];
        vector[0] = new Clase1();
        vector[1] = new Clase2();
        vector[2] = new Clase3();

        for (int i = 0; i<3; i++)
        {
            System.out.println(vector[i].metodo1());
            System.out.println(vector[i].metodo2());
        }
    }
}
```

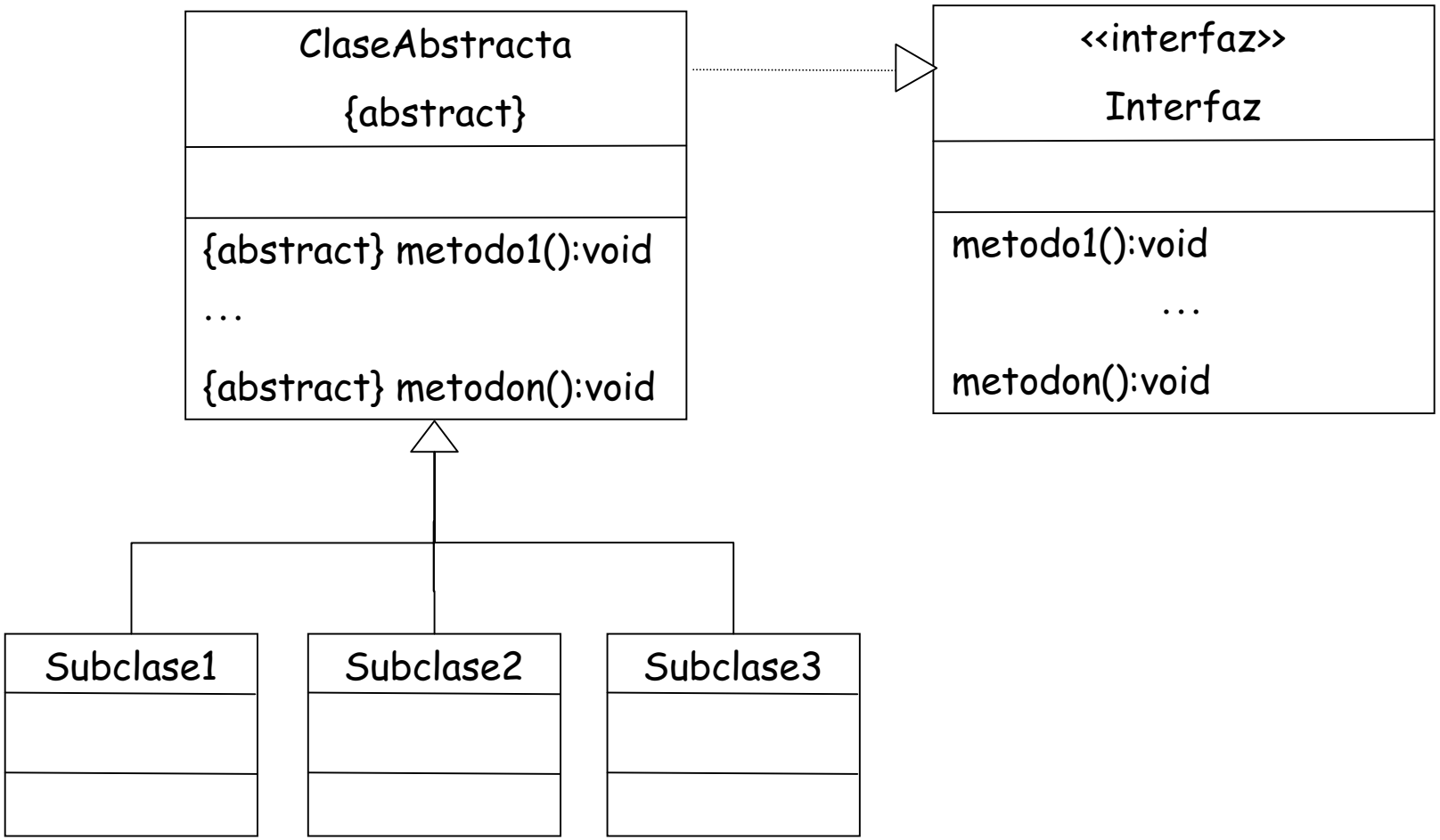
Declaración de un vector de referencias a la interfaz *LaInterface*

Asignación a los elementos del vector de referencias a las clases *Clase1*, *Clase2* y *Clase3*

Comprobación del polimorfismo

Interfaces y polimorfismo (IV)

- **Ejemplo 2:**
  - Piénsese en la codificación Java del siguiente diagrama de clases



Interfaces y polimorfismo (IV)

- Una **referencia a una interfaz espera** referenciar a un **objeto cuya clase implemente** dicha interfaz, pero lógicamente, **desde la referencia al objeto sólo** se tendrá **acceso** a los métodos y constantes **declarados en la interfaz**.

- **Ejemplo:**
  - Se define una interfaz con identificador *LaInterface* con dos métodos *metodo1* y *metodo2.*
  - Se define la clase *Clase1* que implementa *LaInterface* y además tiene un método propio de la clase.
  - Se define una variable de tipo *LaInterfaz*, se le asigna un objeto de la clase *Clase1* y se invoca al método propio de la clase para comprobar que en tiempo de compilación se genera un error ya que se intenta llamar a un método no definido en la interfaz.

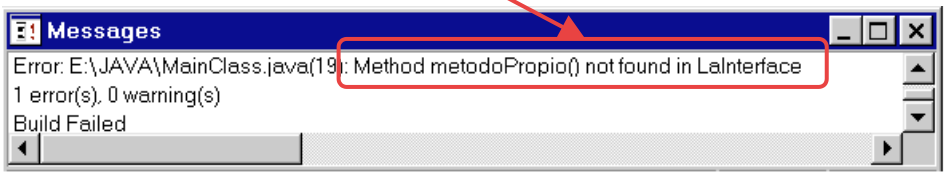
Interfaces y polimorfismo (V)


```
public interface LaInterface
{
   public abstracta String metodo1();
   public abstracta String metodo2();
}
public class Clase1 implementes LaInterface
{
     public String metodo1()  { return "metodo1 de Clase1"; }
     public String metodo2()  { return "metodo2 de Clase1"; }
     public String metodoPropio()  { return "metodo propio de Clase1"; }
}
public class MainClass
{
     public static void main(String [] args)
     {
               LaInterface objetoClase1 = new Clase1();
               System.out.println(objetoClase1.metodoPropio());
     }
}
```

Definición de *LaInterface*

Definición de *Clase1*

ERROR de compilación



La interfaces no pueden crecer ...

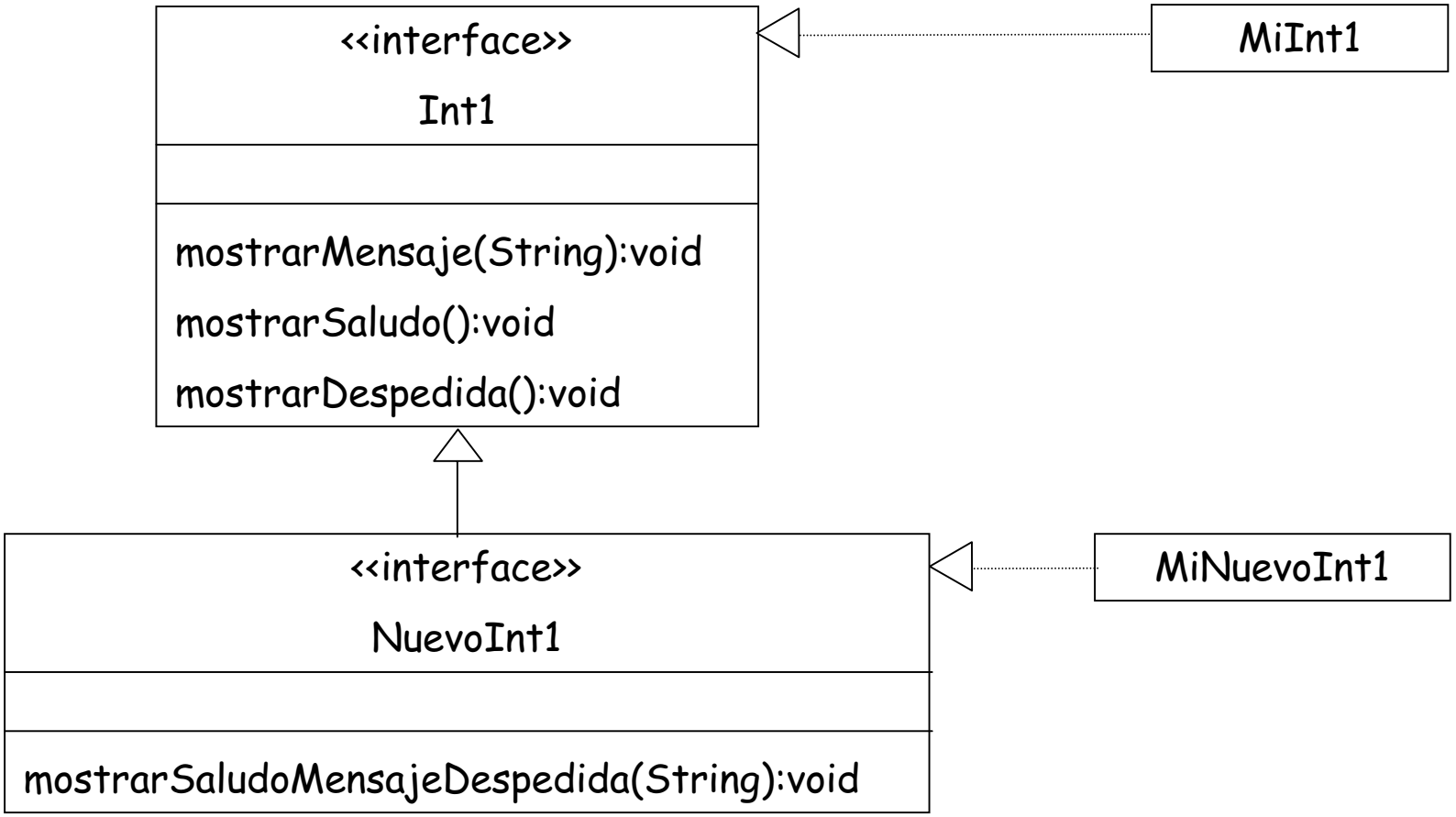
- Si se define una interfaz, y pasado el tiempo, cuando ya hay código utilizando dicha interfaz, se le quiere añadir una nueva funcionalidad, no hay que hacerlo en la propia interfaz porque entonces las clases que la implementaran dejarán de funcionar ya que la nueva funcionalidad (los nuevos métodos) no está implementada.
- Para evitar esto, si se quiere añadir más funcionalidades a una interfaz, es mejor extenderla.
- De esa manera se tendrá dos versiones del interfaz:
  - las clases ya codificadas seguirán implementando correctamente la versión antigua,
  - las clases que se codifiquen pueden elegir entre la versión anterior o la extendida.

La interfaces no pueden crecer ...(ejemplo I)

- **Ejemplo:**
  - Supóngase que se ha codificado una interfaz de nombre *Int1* con los métodos *void mostrarMensaje(String)* , *void mostrarSaludo()* , *void mostrarDespedida()*
  - Supóngase también que ya hay una clase *MiInt1* que implementa esa interfaz.
  - Supóngase que ahora se decide que nuevas clases (por ejemplo *MiNuevoInt1*) que podrían implementar *Int1* necesitarían un método más: *void mostrarSaludoMensajeDespedida(String)*.
  - Si se añade el método a *Int1* la clase *MiInt1* sería incorrecta hasta que se implementara en ella el nuevo método.
  - Una alternativa es definir una nueva interfaz (*NuevoInt1*) que añada el nuevo método. En este caso las clases anteriores siguen estando correctamente codificadas y las nuevas pueden decidir qué versión implementar. En nuestro caso *MiNuevoInt1* implementa *NuevoInt1*.

La interfaces no pueden crecer ...(ejemplo II)

- **Ejemplo:**
  - La semántica anterior puede representarse mediante el siguiente diagrama UML



La interfaces no pueden crecer ...(ejemplo III)


```
interface Int1
{
   final int a = 4;
   public final int b = 40;
   int c = 400;
   final int d = 4000;

   public void mostrarMensaje(String mensaje);
   public void mostrarSaludo();
   public void mostrarDespedida();
}
```

Interfaz Int1, métodos
mostrarMensaje
mostrarSaludo
mostrarDespedida

```
interface NuevoInt1 extends Int1
{
   public void mostrarSaludoMensajeDespedida(String mensaje);
}
```

Interfaz NuevoInt1, añade el método
mostrarSaludoMensajeDespedida

La interfaces no pueden crecer ...(ejemplo IV)


```
class MiInt1 implements Int1
{
   public void mostrarMensaje(String
     mensaje){System.out.println(mensaje);}
   public void mostrarSaludo(){System.out.println("hola");}
   public void mostrarDespedida(){System.out.println("adiós");}
}
```

La clase MiInt1 implementa Int1 y codifica todos sus métodos

```
class MiNuevoIntl implements NuevoInt1
{
   public void mostrarMensaje(String
     mensaje){System.out.println(mensaje);}
   public void mostrarSaludo(){System.out.println("hola");}
   public void mostrarDespedida(){System.out.println("adiós");}
   public void mostrarSaludoMensajeDespedida(String mensaje)
   {
      mostrarSaludo();
      mostrarMensaje(mensaje);
      mostrarDespedida();
   }
}
```

La clase MiNuevoInt1 implementa NuevoInt1 que tiene también el método nuevo

La interfaces no pueden crecer ...(ejemplo V)

main de prueba

```
public class PrInterfaz1
{
   public static void main (String [] args)
   {
      MiNuevoIntl intrfz = new MiNuevoIntl();
      intrfz.mostrarSaludoMensajeDespedida("je je je");
   }
}
```

Implementación de múltiples interfaces

- Si una clase implementa varios interfaces puede ocurrir que dos o más interfaces definan método con el mismo identificador. Los casos que pueden ocurrir son los siguientes:
  - Si los métodos tienen el mismo prototipo, se implementa un método con el mismo prototipo (sólo una vez).
  - Si los métodos difieren en el número o tipo de parámetro (sobrecarga) se implementan todas las sobrecargas.
  - Si los métodos sólo difieren en el tipo de retorno, el compilador produce un error (dos métodos de la misma clase no pueden diferir solamente en el tipo de retorno).

- Si una clase implementa dos interfaces que definen una constante del mismo nombre, en tiempo de compilación se genera un error.

Ejemplo de interfaces Java predefinidos (I)

- Java.lang contiene la interfaz Comparable con sólo un método

  public int compareTo(Object o)

  - Compara el argumento implícito con el explícito de forma que su resultado es el siguiente:
    - Un **entero negativo** si el argumento implícito es menor que el explícito (o)
    - **Cero**, si el argumento implícito es igual que el explícito
    - Un **entero positivo**, si el argumento implícito es menor que el explícito

- Java.util contiene la clase Collections que proporciona muchos métodos de clase para trabajar con diferentes tipos de colecciones.
- Entre ellos el método sort que ordena la colección siempre que todos sus elementos implementen la interfaz Comparable
- A continuación se muestra un ejemplo

Ejemplo de interfaces Java predefinidos (II)

- **Ejemplo:**
  - Las clases *Persona* y *Libro* (que no están relacionadas por herencia) implementan Comparable y codifican el método compareTo.
  - Se crea dos ArrayList, uno de personas (*lista_personas*) y otro de libros (*lista_libros*).
  - Se ordenan las dos listas proporcionándolas como argumento al método de clase sort de Collections.
  - Se muestra los resultados cuando se toman los datos de la línea de comandos.

Ejemplo de interfaces Java predefinidos (III)

- Fichero Libro.java

```
public class Libro implements Comparable
{
   public String titulo=null;

   public Libro(){}
   public Libro(String t)
   {
      titulo = t;
   }

   public int compareTo(Object o)
   {
      return this.toString().compareTo(o.toString() );
   }

   public String toString()
   {
      return this.titulo;
   }
}
```

La clase Libro implementa el interfaz Comparable

Codificación del método compareTo de Comparable

Ejemplo de interfaces Java predefinidos (IV)

- Fichero Persona.java

```
public class Persona implements Comparable
{
   public String nombre=null;

   public Persona(){}
   public Persona(String n)
   {
      nombre = n;
   }

   public int compareTo(Object o)
   {
      return this.toString().compareTo(o.toString() );
   }

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

La clase Persona implementa el interfaz Comparable

Codificación del método compareTo de Comparable

Ejemplo de interfaces Java predefinidos (V)

- Fichero Persona.java

```
import java.util.*;
public class PrInterfaces
{
   public static void main (String [] args)
   {
      ArrayList lista_personas = new ArrayList();
      ArrayList lista_libros = new ArrayList();
      ListIterator li_personas = null;
      ListIterator li_libros = null;
      Persona p = null;
      Libro l = null;

      for (int i = 0; i<args.length/2 ; i++) {
            p = new Persona(args[i]);
            lista_personas.add(p);
      }
      for (int i = args.length/2; i<args.length ; i++) {
            l = new Libro(args[i]);
            lista_libros.add(l);
      }

      Collections.sort( lista_libros );
```

Se importa java.util (donde está ArrayList y Collections)

Creación de las colecciones de libros y personas (lista_personas y lista_libros)

Se guardan la mitad de los argumentos de entrada en la colección de personas

Y la otra mitad en la de libros

Ejemplo de interfaces Java predefinidos (V)

- Fichero Persona.java

```
        Collections.sort( lista_libros );
        Collections.sort( lista_personas );


        System.out.println("LISTA DE LIBROS");
        li_libros = lista_libros.listIterator();
        while (li_libros.hasNext())
        {
           System.out.println(li_libros.next());
        }

        System.out.println("LISTA DE PERSONAS");
        li_personas = lista_personas.listIterator();
        while (li_personas.hasNext())
        {
           System.out.println(li_personas.next());
        }
    }
}
```

Se ordenan las dos colecciones

Se muestra por la salida estándar las dos listas