2.5 Interfaces

Análisis y Diseño de Software
2º Ingeniería Informática
Universidad Autónoma de Madrid

Interfaces: ¿Qué son?

- Los métodos de una clase son su interfaz con el resto de clases
- Una interfaz es un tipo, definido por un grupo de métodos relacionados (sólo las cabeceras), que permiten modelar el comportamiento de un tipo de objetos
- En Java una interfaz es una colección de definiciones de métodos (sin implementación) y de constantes
- Interfaces vacías: Uso como etiquetas semánticas, ej: `Serializable, Cloneable`

¿Por qué son necesarias?

- Una interfaz define un tipo.
  - Similar a una clase con sólo métodos públicos abstractos, y constantes públicas (public static final)
  - Desacopla *especificación* de *implementación*
  - Un contrato entre la(s) clase(s) que implementan las interfaces y el código que usa la interfaz como un tipo
- Diferencia: Herencia múltiple
  - Java no admite herencia múltiple entre clases
  - Una interfaz puede heredar de una o varias interfaces, pero no puede heredar de una clase
- Una clase sólo puede heredar de una clase, pero puede además *implementar* cualquier número de interfaces

Ejemplo

```
public interface Trabajo{
  void ejecutarTrabajo();
}

public class RealizarBackup
       implements Trabajo
{
  private Database d;
  public void ejecutarTrabajo(){
    // Código de realizar backup
  }
}
```

```
public class ColaTrabajo
       implements Trabajo
{
  private List<Trabajo> pendientes;
  public void addTrabajo(Trabajo t){
    pendientes.add(t);
  }
  public void ejecutarTrabajo(){
    for (Trabajo t:pendientes)
      t.ejecutarTrabajo();

    pendientes.clear();
  }
}
```

Ejemplo

"implements" declara que la clase implementa una o varias interfaces

```
public interface Trabajo{
  void ejecutarTrabajo();
}

public class RealizarBackup
      implements Trabajo
{
  private Database d;
  public void ejecutarTrabajo(){
    // Código de realizar backup
  }
}
```

```
public class colaTrabajo
      implements Trabajo
{
  private List<Trabajo> pendientes;
  public void addTrabajo(Trabajo t){
    pendientes.add(t);
  }
  public void ejecutarTrabajo(){
    for (Trabajo t:pendientes)
      t.ejecutarTrabajo();

    pendientes.clear();
  }
}
```

Ejemplo

```
public interface Trabajo{
  void ejecutarTrabajo();
}

public class RealizarBackup
       implements Trabajo
{
  private Database d;
  public void ejecutarTrabajo(){
    // Código de realizar backup
  }
}
```

```
public class colaTrabajo
       implements Trabajo
{
  private List<Trabajo> pendientes;
  public void addTrabajo(Trabajo t){
    pendientes.add(t);
  }
  public void ejecutarTrabajo(){
    for (Trabajo t:pendientes)
      t.ejecutarTrabajo();

    pendientes.clear();
  }
}
```

Ambas clases deben implementar los métodos definidos en la interfaz

Ejemplo

```
public interface Trabajo{
  void ejecutarTrabajo();
}

public class RealizarBackup
       implements Trabajo
{
  private Database d;
  public void ejecutarTrabajo(){
    // Código de realizar backup
  }
}
```

```
public class colaTrabajo
       implements Trabajo
{
  private List<Trabajo> pendientes;
  public void addTrabajo(Trabajo t){
    pendientes.add(t);
  }
  public void ejecutarTrabajo(){
    for (Trabajo t:pendientes)
      t.ejecutarTrabajo();

    pendientes.clear();
  }
}
```

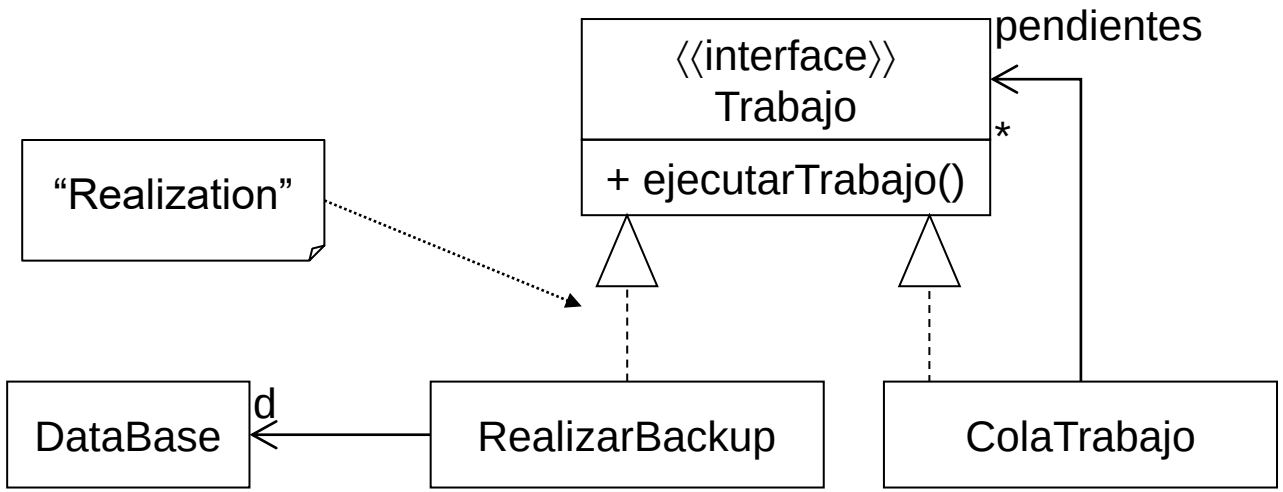
Trabajo es un tipo. Cualquier objeto de una clase que implementa esta interfaz se considera una instancia de dicho tipo.

Definición de interfaces: Sintaxis

```
[public | ...]? interface <interface-name>
      [extends <interface-name1>,
               <interface-name2> ... ]? {
     <interface-members>
}
```

- Una interfaz puede heredar de otras interfaces
- Variables de la interfaz
  - `[public static final] <tipo> <nombre-variable>=<valor>;`
  - Son variables “estáticas y finales” (constantes), implicitamente
- Los métodos se definen sin código (igual que los métodos abstractos)
  - Cualquier clase que implemente la interfaz los debe definir (o declararlos abstractos)
  - Se admite public y abstract, pero son opcionales, ya que son modificadores implícitos.

Definición de Interfaces en UML



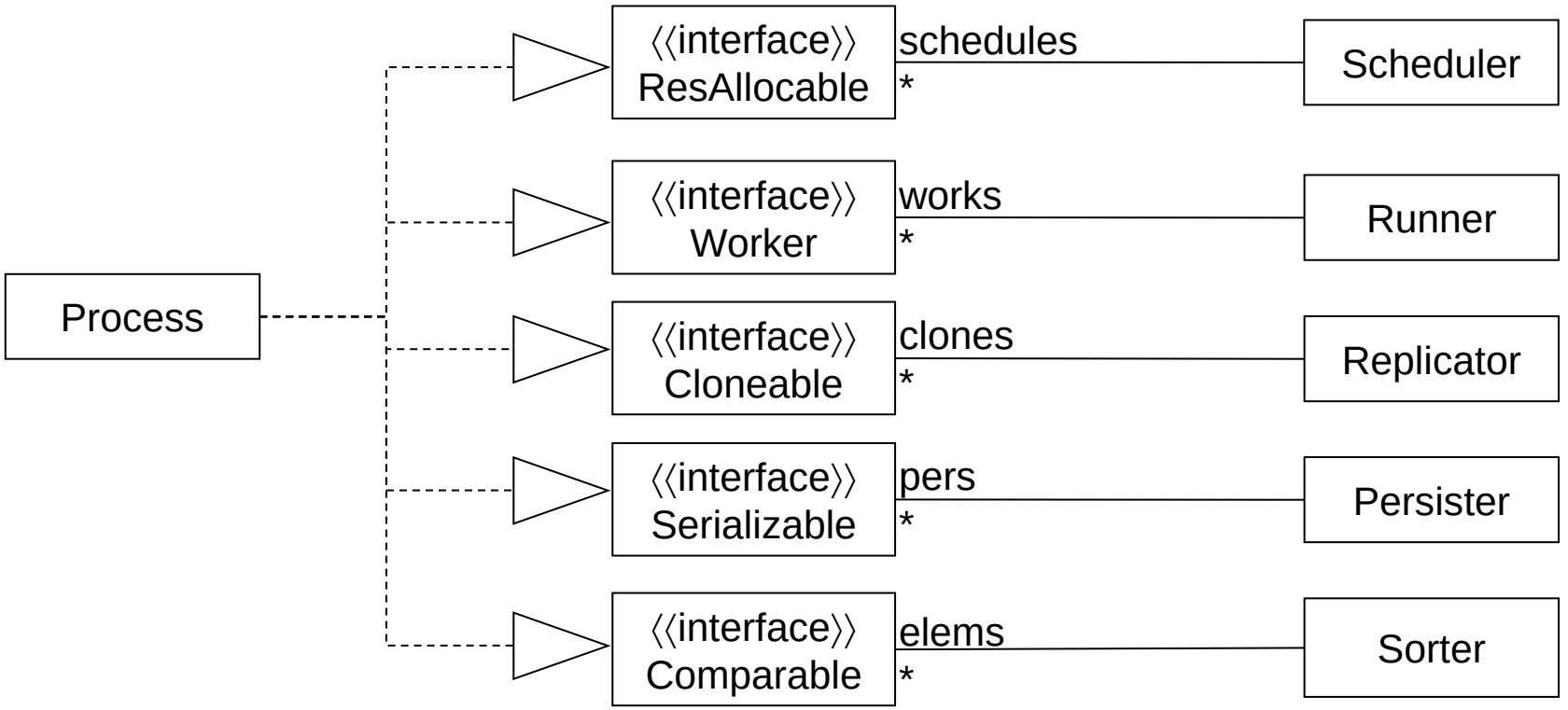
Ejercicio

- Diseña el API de una clase de utilidad con un método de ordenación `sort`, para ordenar listas de `Strings`

- El método `sort` debe ordenar los elementos de la lista por un criterio definido por el usuario, que se pasa como parámetro
  - A modo de ejemplo, define un criterio basado en la longitud del `String`

Utilidad

- Permite definir conjuntos de funcionalidad – interfaces – para poder acceder a clases heterogéneas de manera uniforme
- Permite ver una clase desde distintas perspectivas



Ejemplo

Interfaz Comparable

```
public interface Comparable{
  int compareTo(Object o)
}

public class Coche
implements Comparable
{
  int cilindrada;
  String marca;
  String modelo;
  //…
  int compareTo (Object o){
    Coche c= (Coche) o;
    return cilindrada-c.cilindrada;
  }
}
List listaDeCoches;
// …
Collections.sort(listaDeCoches.sort);
```

- Definida en `java.lang`.
- Da el “orden natural” de una clase.
- Las clases que la implementan se pueden ordenar, por ejemplo mediante: Collections.sort y Arrays.sort.

Problema: Dos clases que implementan esta interfaz pueden no ser comparables directamente

Ejemplo: Interfaces genéricas

Interfaz Comparable<T>

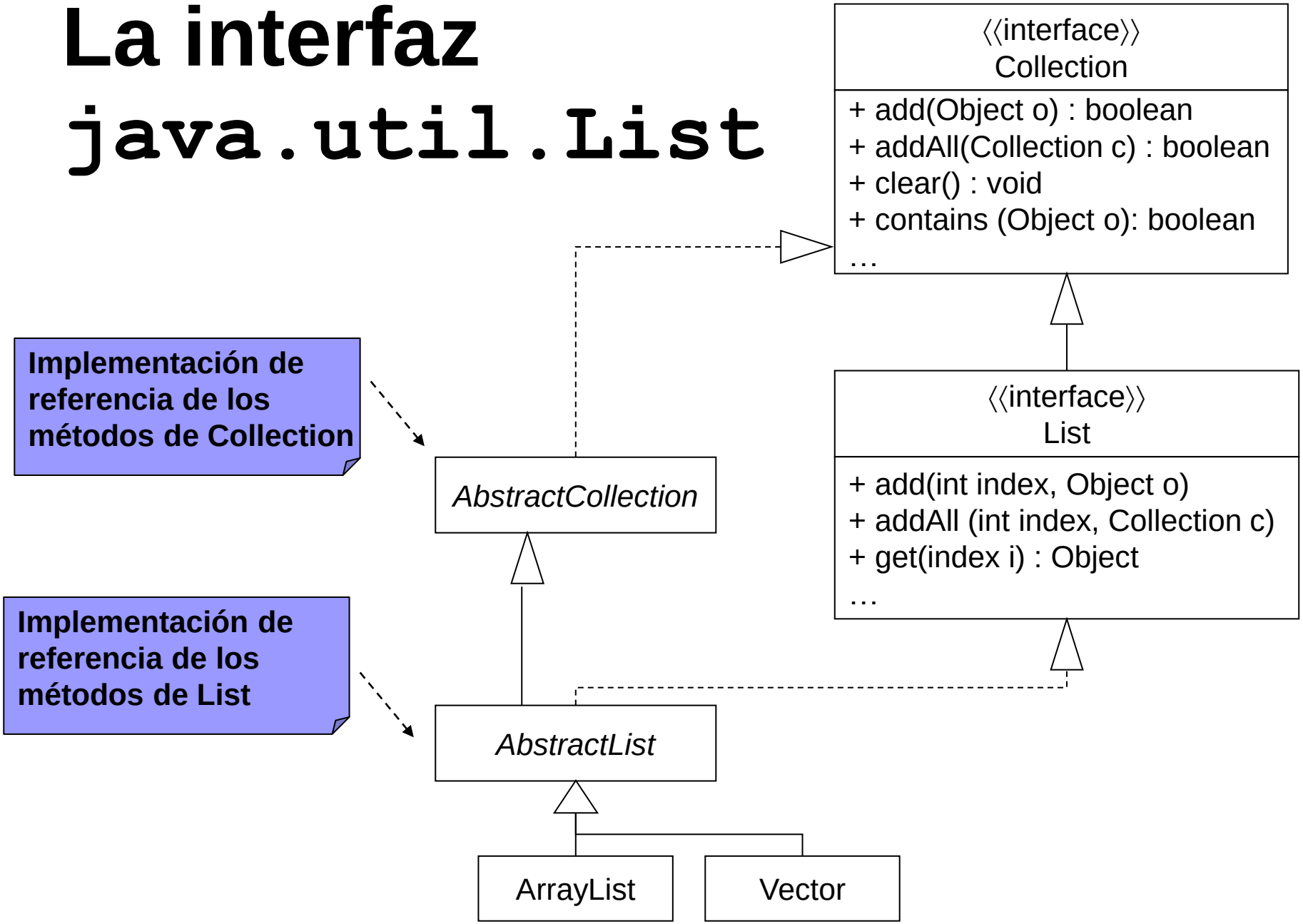
```
public interface Comparable<T>{
  int compareTo(T o)
}

public class Coche
implements Comparable<Coche>
{
  int cilindrada;
  String marca;
  String modelo;
  // …
  int compareTo (Coche c){
    return cilindrada-c.cilindrada;
  }
}
List<Coche> lista;
…
lista.sort();
```

- Interfaz Comparable genérica desde Java 5.0
- Evita los problemas de seguridad de tipos en la comparación.

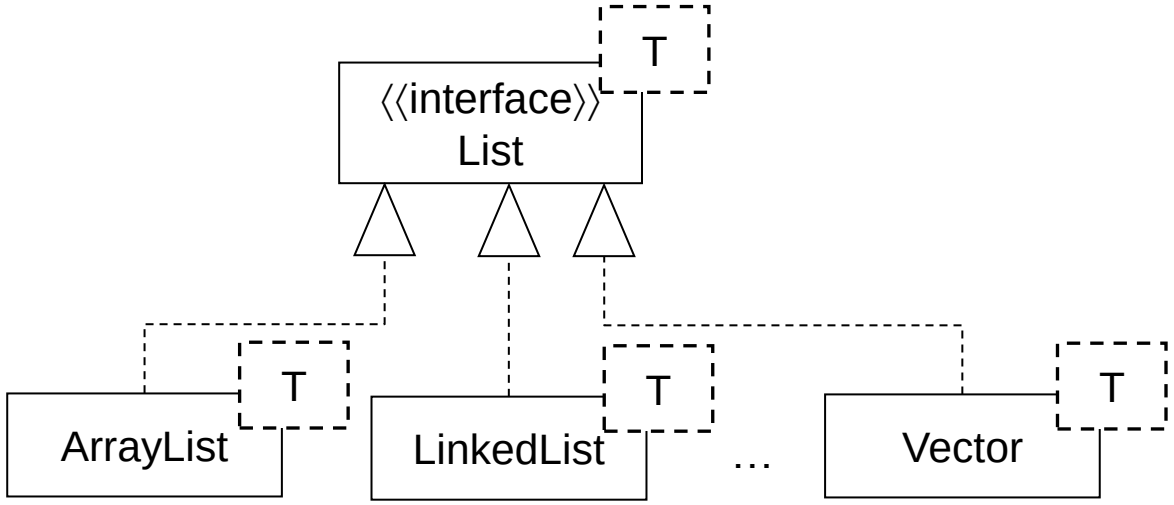
Ejemplos de Interfaces de la librería estandar

- *Cloneable*: Indica al método `Object.clone()` que se puede copiar campo a campo.
- *Comparable<T>*: Objetos comparables, orden natural
- *Comparator<T>*: Función de comparación
  - int compare(T o1, T o2))
- *Serializable*: El objeto se puede serializar (copia binaria a disco o enviado por red)
- TADs: Collection, Iterable, Queue, Deque, List, Map, Set, TreeModel
- DOM: Document, Node, Element, Attr

La interfaz `java.util.List`



Ejemplo: `List<>`



- Todos los objetos de tipo `List<>` se pueden usar de manera uniforme (no nos importa la clase concreta que estamos usando)

```
void method(List<String> l) {…}
```

- Podemos cambiar la clase implementación sin cambiar el código cliente

```
obj.method(new ArrayList<String>() )
obj.method(new LinkedList<String>() )
…
```

Modificación de interfaces

- Una vez creada una interfaz y las clases que la implementan, al añadir o modificar un método de una interfaz es necesario cambiar todas las clases que lo implementan.

- ¿Soluciones?
  - Crear una nueva interfaz, que hereda de la original y contiene los nuevos métodos
  - Dar una *implementación de referencia* en una clase (problema: en Java no tenemos herencia múltiple de clases)
  - Dar una implementación de referencia usando métodos default

Ejemplo de modificación

```
public interface Trabajo{
  void ejecutarTrabajo();
}

public interface TrabajoTransaccional extends Trabajo{
  void ejecutarEnTransaccion(Transaccion t);
}
```

- La nueva interfaz ofrece una mayor funcionalidad, y no requiere modificar las clases que implementan la versión reducida

- **Problema**: Las clases existentes no cumplen la nueva interfaz, aunque los nuevos métodos se puedan implementar usando los antiguos

Implementación de referencia

- Simplifican la implementación de interfaces que tienen muchos métodos
  - Algunos pueden ser implementados llamando a otros métodos de la interfaz
- Un ejemplo del JDK: `java.util.AbstractList`
  - Para una lista read-only, sólo necesitamos implementar dos métodos: *get(int)* and *size()*
  - Proporciona muchos métodos: *iterator, equals, indexOf(Object), lastIndexOf(Object), subList()*, etc.

Una implementación de referencia

```
public interface Arbol{
  Object getElemento();
  Arbol hijoIzq();
  Arbol hijoDer();
  boolean esHoja();
  boolean esVacio();
  Object search(Object o);
}
```

Subclases de ArbolAbstracto sobreescribirán algunos métodos, algunos también por razones de eficiencia.

```
public abstract class ArbolAbstracto
implements Arbol
{
  public boolean esHoja(){
    return hijoIzq().
           esVacio()&&
           hijoDer().
           esVacio(); }
  public Object search(Object o){
  //implementa búsqueda binaria…
  }
  // Podemos implementar los otros
  // métodos simplemente lanzando
  // una excepción.
}
```

Métodos default en interfaces

- Contienen código
  - Pueden llamar a otros métodos de la interfaz, incluso si estos no tienen código
- Pueden sobreescribirse (o no) en clases que implementen la interfaz

Métodos default

```
public interface Arbol<T>{
  T getElemento();
  Arbol<T> hijoIzq();
  Arbol<T> hijoDer();
  default boolean esHoja() {
                return  this.hijoIzq().esVacio() && this.hijoDer().esVacio();
  }
  boolean esVacio();
  default T search(T o) {
        if (this.getElemento().equals(o)) return this.getElemento();
        else {
                T result = null;
                if (! this.hijoIzq().esVacio() ) result = this.hijoIzq().search(o);
                if (result != null) return result;
                if (! this.hijoDer().esVacio() ) result = this.hijoDer().search(o);
                if (result != null) return result;
        }
        return null;
  }
}
```

Métodos default: motivación

- Poder añadir métodos a una interfaz sin romper código que ya funciona (los nuevos métodos de la interfaz serían métodos default)

- Especificar métodos que son opcionales
  - Dar una implementación que devuelve una excepción (por ejemplo `Collections.unmodifiableList(list);`)

- Facilitar la implementación de interfaces (similar a una clase abstracta con implementaciones de referencia)

Diferencias con clases abstractas

- Una interfaz no tiene estado interno
  - No puede declarar atributos (variables de instancia), sólo constantes
- Una clase sólo puede heredar de una clase, mientras que puede implementar varias interfaces
- El propósito de las interfaces sigue siendo especificar “qué” (firmas de métodos) y no “cómo” (código en los métodos)

Métodos estáticos en interfaces

- Es posible añadir métodos estáticos en una interfaz, de manera similar a como se hace en una clase.
- Útil para definir librerías.
  - Ejemplo: creación de algunos comparadores útiles en `Comparator<T>`

```
public interface Comparator<T> {
   int compare(T o1, T o2);
   static <T extends Comparable<? super T>> Comparator<T> naturalOrder() { … }

   static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator) { … }
   static <T> Comparator<T> nullsLast(Comparator<? super T> comparator) { … }
   //…
}
```

Emulando herencia múltiple de clases con interfaces

```
class C extends A, B{
}
```

- No se admite en Java
- Podemos usar interfaces:

```
interface A{}

interface B{}

interface C extends A, B{}

class AImpl implements A{ }

class BImpl implements B{ }
```

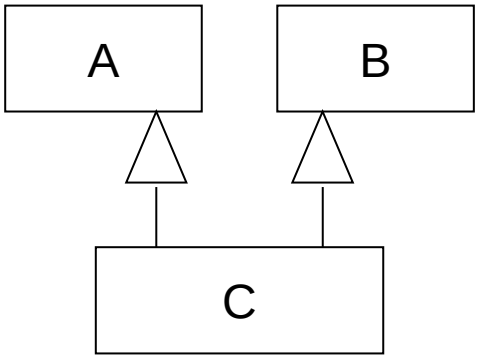
- Heredamos de la clase más compleja y usamos ***delegación*** en la otra:

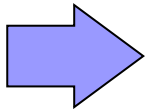
```
class CImpl extends AImpl
             implements C
{
  B b= new BImpl();
}
```

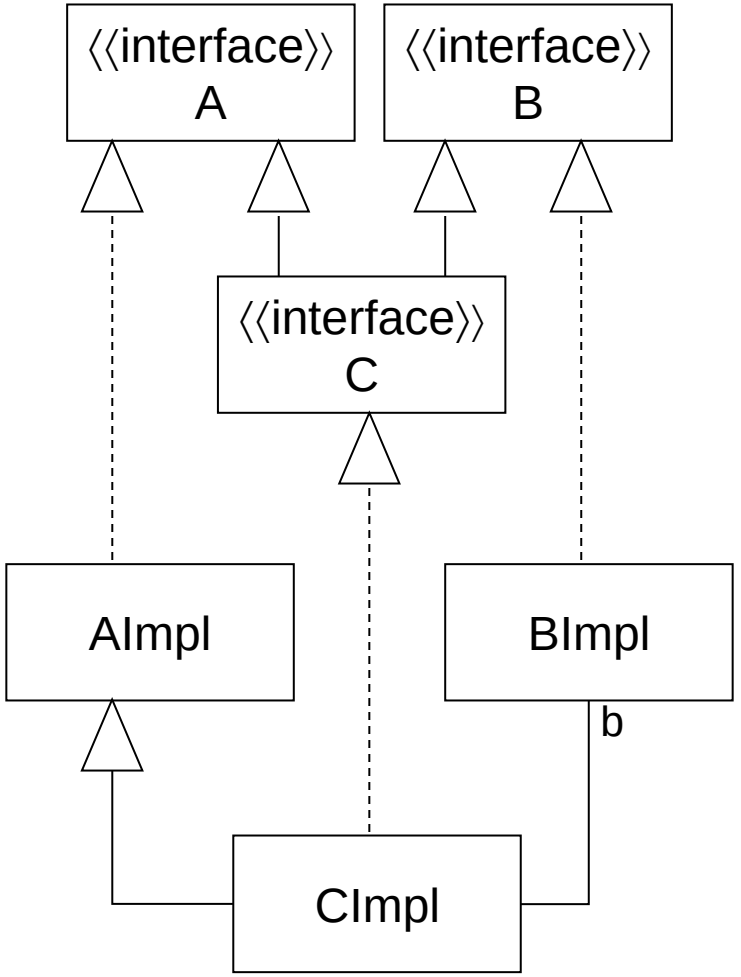
- Delegamos en «b» los métodos propios de B mientras que los de A se heredan, por ejemplo:

```
public int bIntMethod() {
    return b.bIntMethod();
 }
```

Herencia múltiple con interfaces



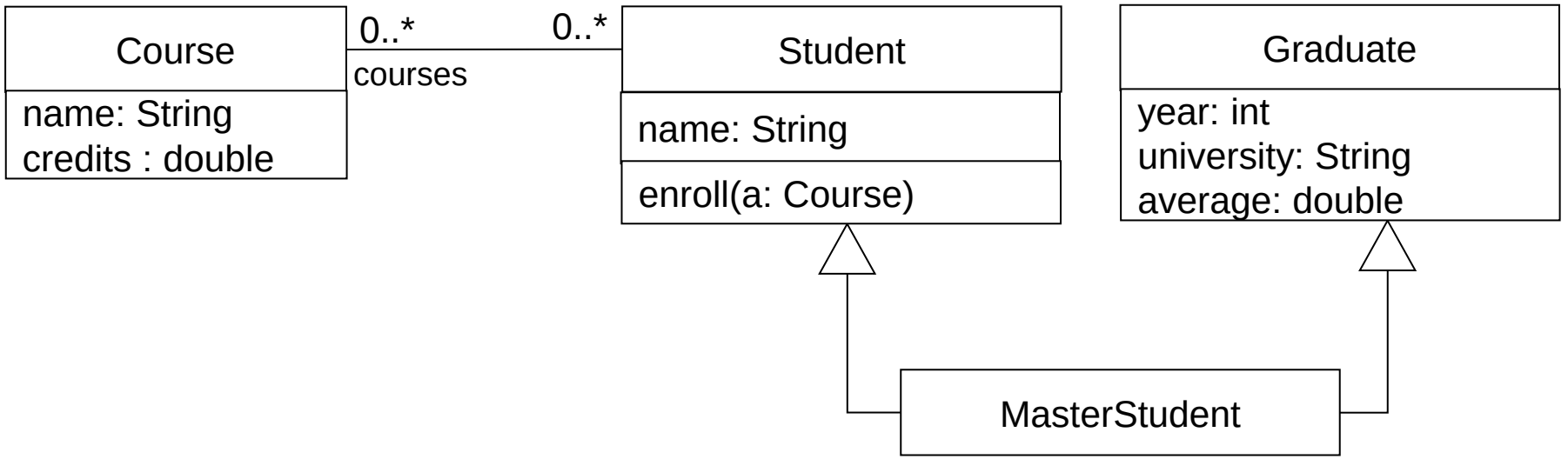




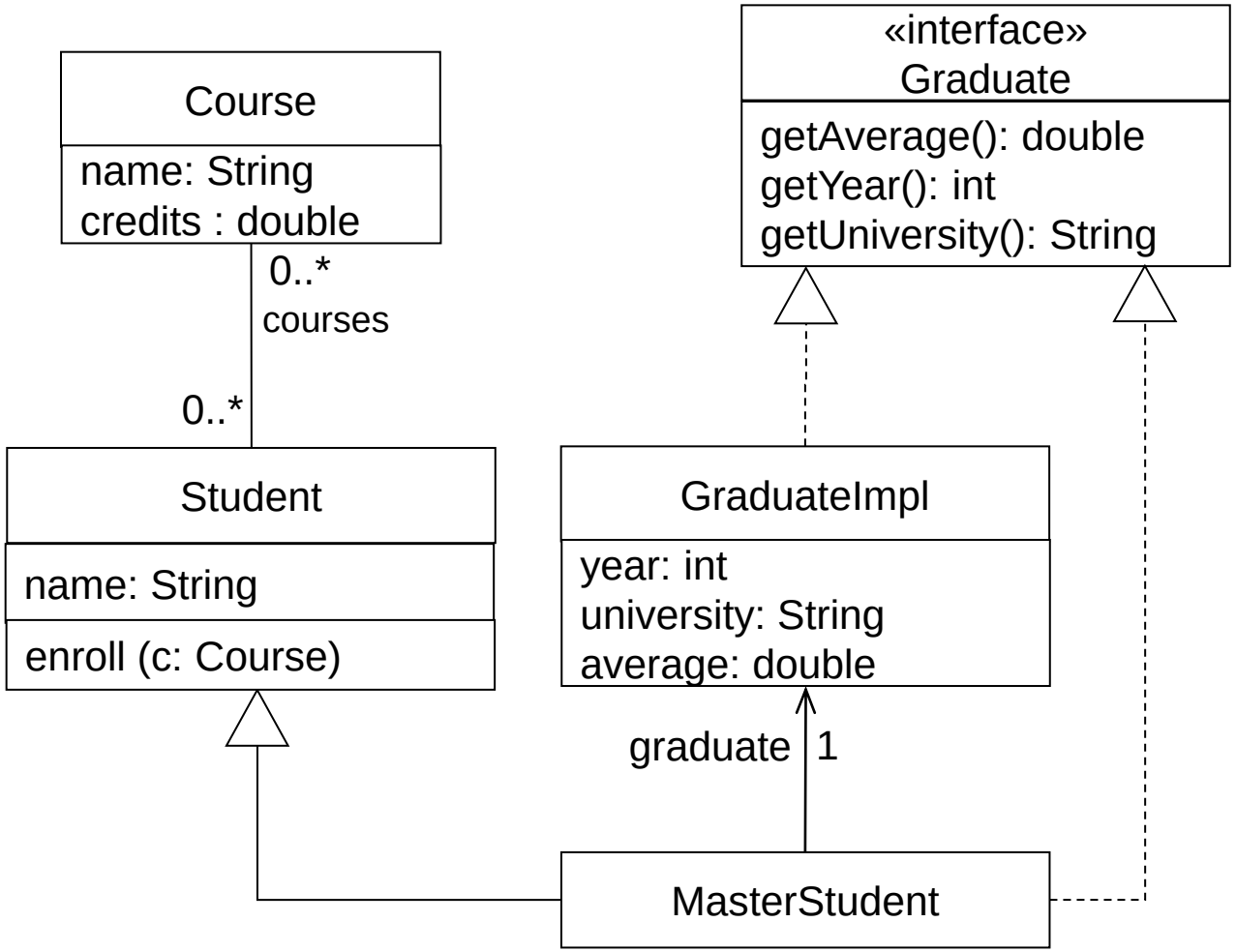
Válido en UML, y en lenguajes como C++, pero no en Java

Ejercicio (i)

- Modifica el diseño de más abajo para que sea implementable en Java



Ejerciccio (i, solución)



Ejercicio (ii)

- Crea una clase de utilidad `PrettyPrinter` para imprimir estructuras en forma de árbol usando indentación

- La clase de utilidad debe ser altamente reusable, p.ej., para poder usarse con la clase `Carpeta` del ejercicio anterior del sistema de ficheros

Resumen

- Una interfaz define un protocolo de comunicación entre objetos.
- La interfaz contiene declaraciones, sin implementación, de métodos y constantes.
- Una clase que implementa una interfaz ha de implementar todos los métodos de la interfaz (si no tendría que ser abstracta)
- Una interfaz define un tipo: su nombre se puede utilizar en cualquier sitio donde se pueda utilizar un tipo