Universidad Complutense de Madrid
Facultad de Informática
Grado en Ingeniería Informática, Grupo B

Tecnología de la Programación

La Herencia en la Programación Orientada a Objetos

(Temas 4 y 5 de los apuntes)

Simon Pickin
Alberto Díaz, Puri Arenas, Marco Antonio Gómez

Herencia

- Mecanismo exclusivo y fundamental de la POO.
- Fomenta y facilita la reutilización de software
  - Es el principal mecanismo de la OO para ello
- Si se necesita una nueva clase de objetos
  - y se detectan suficientes similitudes con otra clase ya desarrollada,
  - se puede tomar la clase existente como punto de partida para desarrollar la nueva clase
  - de forma que se reutilizan características
    - ya implementadas, ahorrando tiempo y esfuerzo en la codificación
    - ya probadas, ahorrando tiempo y esfuerzo en la prueba y depuración.

Herencia, la relación *es-un*

- Una clase `B` es una subclase de `A` si posee o puede acceder a
  - todos los métodos y atributos de la clase `A`, y
  - métodos y/o atributos adicionales
- Esta definición es compatible con la herencia como especialización
  - una subclase posee un comportamiento y unos datos que son una extensión (un conjunto estrictamente mayor) de los de su superclase
- Los objetos de la clase `B` también pertenecen a la clase `A`
- Se suele decir que la herencia implementa la relación *es-un*
  - clase `Animal`, subclase `Perro` => un perro es un animal
  - clase `Persona`, subclase `Alumno` => un alumno es una persona

Ejemplo 1: la clase `Persona`

```
public class Persona {
   private long dni;
   private String nombre;

   public Persona() { dni = -1; nombre = ""; }
   public Persona(long unDni, String unNombre) {
       this.dni = unDni;
       this.nombre = unNombre;
   }
   .... // implementar métodos get y set para estos attributos

   // en el mundo real, utilizaríamos toString(), no mostrar()
   public void mostrar() {
       System.out.println("Nombre: " + this.nombre);
       System.out.println("DNI: " + this.dni);
   }
}
```

Ejemplo 1: definir la clase `Alumno`

- Después de haber definido y usado la clase `Persona`, suponemos que surge la necesidad de manejar un tipo concreto de personas †
  - Los alumnos.
- Un objeto alumno almacena la información de estado que almacena un objeto persona pero también necesita almacenar:
  - el número de matrícula
  - y el número de créditos aprobados.
- En vez de crear una clase `Alumno` desde cero, posiblemente copiando partes del código de `Persona`, y dado que un alumno es una persona, podemos utilizar el mecanismo de herencia
  - crear una clase `Alumno` como subclase de la clase `Persona`

† En realidad, el caso más común es darse cuenta de que dos clases que se están especificando, p.ej. `Alumno` y `Profesor`, tienen estado y comportamiento comunes que podrían factorizarse en otra clase, p.ej. `Persona`.

Ejemplo 1: la clase `Alumno`

```
public class Alumno extends Persona {
        private long numMatricula;
        private int creditosAprobados;

        public Alumno() { numMatricula = -1; creditosAprobados = 0; }

        // omitido constructor con parámetros

        public long getNumMatricula() { return numMatricula; }
        public long getCreditosAprobados() { return creditosAprobados; }
        public void setNumMatricula(int mat) { numMatricula = mat; }
        public void setCreditosAprobados(int cr) { creditosAprobados = cr; }
}
```

Ejemplo 1: `Alumno` hereda de `Persona`

- La clase `Alumno` hereda de la clase `Persona`
  - debido a la claúsula `extends Persona` utilizada en su declaración
- Las instancias de `Alumno` son también instancias de `Persona`,
  - un objeto de la clase `Alumno` también tiene disponibles los métodos públicos de la clase `Persona`:

```
Alumno alum = new Alumno();
alum.setNombre("Walterio Malatesta");
alum.setDNI(12312312);
alum.setCreditosAprobados(9);
alum.mostrar();
```

- El objeto `alum` incorpora tanto los atributos privados declarados en la clase `Alumno` como los heredados de la clase `Persona`
  - por lo que en memoria almacena dos `long`, un `int` y un `String`.
  - aunque los atributos de la clase `Persona` no están accesibles

Terminología

- Se dice que `Persona`
  - es la *clase padre*, *superclase* o *clase base* de la clase `Alumno`.
- De manera recíproca, `Alumno`
  - es la *clase hija*, *subclase* o *clase derivada* de la clase `Persona`.
- También se dice que
  - La clase `Alumno` especializa la clase `Persona`
  - La clase `Persona` generaliza la clase `Alumno`.

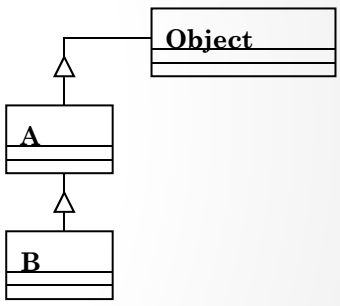
Subclases y superclases

- La clase derivada o subclase:
  - Hereda todas las características
    - atributos y métodos

    de la clase base o superclase.
  - Puede definir características adicionales
    - atributos y métodos
  - Puede redefinir características heredadas
    - métodos
- La herencia no afecta directamente a la superclase, pero véase
  - El uso de la palabra clave `virtual` en C++, C#, etc.
  - La dificultad intrínseca a la herencia conocida como *la fragilidad de la clase base* (véase el apéndice para más detalle)

Herencia y jerarquía de clases

- Java no permite la herencia múltiple
  - Una clase sólo puede ser subclase de una única clase
- La clase `java.lang.Object`
  - define e implementa el comportamiento común a todas las clases
- La relación de herencia induce una organización jerárquica cuya cima es la clase `Object`
  - Si una clase no se deriva explícitamente de alguna otra, implícitamente se deriva de la clase `Object`



Constructores

- Los constructores no se heredan

```
public class A {
    ...
    public A(int ix, int iy){ ... };
}

public class B extends A {
    ...
}

...
B b = new B(1,2); // error, ningún constructor encaja
```


Constructores

- El constructor de la superclase se puede invocar con `super`
  - debe ser la primera instrucción

```
public class A {
    private int x, y;
    public A(int ix, int iy) { x = ix; y = iy; }
}

public class B extends A {
    private int z;
    B(int ix, int iy) { super(ix, iy); z = 0; }
    B(int ix, int iy, int iz) {super(ix, iy); z=iz;}
    B(B b) {z = b.z; super(b.x, b.y); // error
}
```

Constructores, política por defecto en Java

1. Si una clase no contiene constructor alguno
   - el compilador inserta un constructor sin argumentos vacío, luego aplica la regla 2
2. Si la primera línea de cualquier constructor no es una llamada a otro
   - el compilador inserta una llamada al constructor sin argumentos de la superclase
   - Si la superclase tiene constructor pero no un constructor sin argumentos
     - la llamada fracasa y el código no compila
   - Si la superclase no tiene constructor alguno
     - la regla 1 se aplica (a la superclase) y el compilador inserta un constructor sin argumentos en la superclase y la llamada tiene éxito

- Y si no hemos especificado que la clase hereda de otra
  - hereda necesariamente de la clase del sistema llamada `Object`
  - Aplicación de la regla 2 resulta en una llamada al constructor sin argumentos de la clase `Object`

Implicaciones, constructores definidos por el programador

- Un constructor de una subclase puede contener una llamada a un constructor de su superclase:
  - `super(parámetros);` (donde el número de parámetros puede ser 0)
  - Si presente, **tiene que ser** la primera instrucción del código del constructor
- Si no es el caso, el compilador inserta una llamada al constructor sin parámetros de la superclase: `super();`
  - Como la primera instrucción del código del constructor
- Si se llama a `super` pero superclase no tiene constructor sin params
  - Error al compilar, si tiene al menos un constructor con parámetros
  - No hay error, si tampoco tiene ningún constructor con parámetros
    - Si no hay constructor, el compilador inserta un constructor sin parámetros
    - y luego inserta en este constructor la instrucción `super();`
    - etc. , recursivamente subiendo la jerarquía de herencia

Encadenamiento de constructores

- Asumiendo constructores sin parámetros, el procedimiento es:
  - Desde la clase, digamos X, del objeto que va a crearse
  - Ascender la jerarquía de herencia hasta la clase `Object`
  - Ejecutar las inicializaciones incluidas en las declaraciones de atributos
    - Ejecutando el constructor de cualquier objeto creado y asignado a un atributo
  - Ejecutar el código del constructor de la clase `Object`
  - Ejecutar las inicializaciones incluidas en las declaraciones de atributos de la clase hija
    - Ejecutando el constructor de cualquier objeto creado y asignado a un atributo
  - Ejecutar el código del constructor de la clase hija
  - Descender la jerarquía de herencia
    - ejecutando recursivamente las inicializaciones de atributos y de los constructores
    - hasta llegar a la ejecución del constructor de la clase X

Sobrescritura de métodos

```
public class A {
    private int x, y;

    // en el mundo real, utilizaríamos toString()
    public void print(){System.out.println(x + " " + y);}

}

public class B extends A {
    private int z;
}
```

- Un problema que tiene la clase `B` que hereda de `A` es que el método `print` sólo muestra las variables `x` e `y`

```
B b = new B(1, 2, 3);
b.print();  // 1 2     es decir, no hace lo debido
```

Sobrescritura de métodos

- Algunos de los métodos heredados pueden no resultar adecuados
  - siendo necesario volver a implementarlos en la subclase.
- S*obrescritura*: redefinición de la implementación de un método
  - dando una nueva implementación.
- Se crea un método en la subclase con la misma signatura
  - es decir, con el mismo nombre, número/tipo de parámetros y tipo devuelto
- Si se llama al método de un objeto de la subclase (resp. superclase)
  - Se ejecuta el método definido en la subclase (resp. superclase)
- Para llamar al método sobrescrito de la superclase desde la subclase
  - se utiliza la palabra clave `super`.

Ejemplo 2: sobrescritura de métodos

- Se puede sobrescribir el método `print` en la clase `B`

```
class B extends A {
    private int z;
    ...

    void print(){
        System.out.println(x + " " + y + " " + z);
    }
}

B b = new B(1, 2, 3);
A a = A(4,5);
b.print();      // 1 2 3   hace lo debido
a.print();      // 4 5     hace lo mismo que antes
```

Ejemplo 2, redefinición parcial

```
class B extends A {
    void print(){                  // redefinición parcial
        super.print();             // el print de la clase A
        System.out.println(" " + z);
    }
}
```


Sobrescritura de métodos, ejemplo `Persona` y `Alumno`

- Apliquemos la misma idea al ejemplo inicial de `Persona` y `Alumno`
  - Redefinimos el método `mostrar()` en `Alumno`
    - para que mostrara toda la información
    - Primero con una llamada `super.mostrar()`, luego imprimiendo el resto de la información
- ¿Qué mostraría este código?

```
public static void main(String args[]) {
        Alumno alumno=new Alumno();
        Persona persona=new Persona();

        persona.mostrar();  // imprime los atributos nombre y dni
        alumno.mostrar();   // ¿imprime qué?
}
```

Sobrescritura de métodos, ejemplo `Persona` y `Alumno`

- El método `mostrar()` del objeto almacenado en la variable `alumno` ejecuta
  - ¿el código del método `mostrar()` de la clase `Alumno`?
  - o ¿el código del método `mostrar()` de la clase `Persona`?
- Respuesta: se elige el método que está en la clase más especializada
  - de entre las clases del camino entre la clase del objeto cuyo método se ha llamado y la raíz de la jerarquía de herencia
  - En este caso, se ejecutaría el método `mostrar()` de la clase `Alumno`.
- La anotación de Java `@Override`
  - Indica explícitamente que el método en cuestión sobrescribe otro
    - Permite al compilador comprobar la igualdad de signatura con un método heredado
    - Java 6+: usado también cuando un método implementa un método de una interfaz
  - Algunos entornos de desarrollo pueden agregan automáticamente esta anotación

Ocultación de métodos estáticos

- El mecanismo de sobrescritura
  - permite redefinir métodos de instancia en una clase derivada
  - pero ¿qué pasa en el caso de los métodos estáticos?
- Los métodos estáticos
  - no pueden ser sobrescritos (redefinidos)
  - pero pueden ser *ocultados*
- Si una subclase define un método `static`
  - con la misma signatura que un método `static` de su superclase
  - el método de la clase derivada *oculta* al de la superclase
- Para llamar al método ocultado de la superclase desde la subclase
  - se utiliza `super`

Ejemplo 3: sobrescritura y ocultación

```
class Padre {
    public static void metodoClase() {
        System.out.println("metodoClase() en Padre");
    }
    public void metodoInstancia() {
        System.out.println("metodoInstancia() en Padre");
    }
}

class Hija extends Padre {
    public static void metodoClase() {
        System.out.println("metodoClase() en Hija");
    }
    public void metodoInstancia() {
        System.out.println("metodoInstancia() en Hija");
    }
}
```


Ejemplo 3: sobrescritura y ocultación

```
class Prueba {
    public static void main(String[] args) {
        Padre miObjeto = new Hija(); // Polimorfismo de subclase
        miObjeto.metodoInstancia();
        miObjeto.metodoClase();
    }
}
```

- Salida de este programa:

```
metodoInstancia() en Hija    // usa el tipo real
metodoClase() en Padre       // usa el tipo declarado
```

- En Java
  - Se pueden sobrescribir / redefinir métodos de instancia
  - Se pueden ocultar métodos de clase (métodos estáticos)
  - Es legal escribir `objeto.metodoClase` en vez de `Padre.metodoClase`

Ocultación de atributos

- Si una subclase declara un atributo
  - con el mismo nombre y tipo que un atributo de su superclase (*)
  - el atributo de la clase derivada oculta al de la superclase
- Para referenciar al atributo ocultado de la superclase desde la subclase
  - se utiliza la palabra clave `super`
  - Si la visibilidad no es `public` o `protected`: error
- La ocultación de atributos es desaconsejada

(*) Como veremos luego, en Java, basta con que el atributo de la subclase tengan el mismo nombre que el atributo de la superclase

Sobrescritura vs sobrecarga

- Sobrecarga de métodos:
  - Definición de múltiples métodos
    - Con el mismo nombre pero distinta lista de parámetros, esto es:
      - o bien el número de parámetro es distinto
      - o bien el tipo de al menos uno de los parámetros es distinto
  - El compilador de Java trata a métodos sobrecargados como métodos distintos
- Sobrescritura de métodos:
  - Definición de métodos en distintas clases de una jerarquía de herencia
    - con el mismo nombre, la misma lista de parámetros y el mismo tipo de retorno
  - El compilador de Java trata al método que sobrescribe como una versión distinta del método sobrescrito
    - En la subclase, el compilador substituye el código del método de la superclase por el del método de la subclase
- Si existen relaciones de subclase entre tipos de parámetros o de retorno
  - ¿Se trata de *sobrecarga de métodos o de sobrescritura de métodos*?

Relación entre tipos de atributos y métodos

- Suponemos que `B` es una subclase de `A` (`B` hereda de `A`)
- Para que un atributo `x` de `B` oculte un atributo `x` de `A` (*)
  - ¿cuál es la relación de subclase que se puede permitir entre
    - el tipo del atributo `x` de la clase `B`
    - y el tipo del atributo `x` de la clase `A`?
- Para que un método `m` de `B` sobrescriba un método `m` de `A`
  - ¿cuál es la relación de subclase que se puede permitir entre
    - el tipo de un parámetro del método `m` de la clase `B`
    - y el tipo del parámetro correspondiente del método `m` de la clase `A`?
  - ¿cuál es la relación de subclase que se puede permitir entre
    - el tipo de retorno del método `m` de la clase `B`
    - y el tipo de retorno del método `m` de la clase `A`?

(*) Se desaconseja ocultar atributos

Relación “correcta” entre tipos de atributos y métodos

- ¿Cuándo define una subclase un subtipo?
  - Tanto en el caso de sobrescritura como en el caso de sobrecarga
    - la subclase puede definir un subtipo
    - En el primer caso el subtipo tiene un método cuando en el segundo tiene dos
- ¿Cuándo debe haber sobrescritura y cuando debe haber sobrecarga
  - ¿Cuándo debe el subtipo tener un método y cuando debe tener dos?
  - ¿Qué es “lo correcto”?
- Para saber qué es “correcto”, veamos lo que dice la teoría de tipos
  - Para que podamos hacer definiciones que respetan la teoría de tipos
    - en lo que concierne a “tipos de función”

Subtipado de funciones en la teoría de tipos

- El tipo de función `S1→S2` es un subtipo del tipo de función `T1→T2` si
  - `T1` es un subtipo de `S1`
    - la relación entre los tipos de dominio es *contravariante*
  - `S2` es un subtipo de `T2`
    - la relación entre los tipos de codominio es *covariante*
  - Informalmente, el subtipo es "más liberal" en los tipos que acepta y "más conservador" en el tipo que devuelve
- Una manera más formal de decirlo: el constructor de tipos "→"
  - es contravariante en el tipo de dominio y covariante en el tipo de codominio
- Compare la idea de contravarianza y covarianza con la idea de
  - debilitar precondiciones y fortalecer postcondiciones (ver EDA)

Ejemplo 4, sobrescritura vs sobrecarga

```
public class AparatoAutomatizacion {... }

public class Maquina extends AparatoAutomatizacion {... }

public class SmartPhone extends Maquina {... }

public class Luddite {
       Maquina ultimoDestruido;
       public void destruir(Maquina maquina){... }
       public Maquina siguienteDestruir(){... }
}
```

Ejemplo 4: sobrescritura vs sobrecarga

```
public class LudditeModerno extends Luddite {
        SmartPhone ultimoDestruido;                          // covarianza
        public void destruir(Smartphone smartphone){... }    // covarianza
        public SmartPhone siguienteDestruir(){... }          // covarianza
}
public class LudditeNuevo extends Luddite {
        SmartPhone ultimoDestruido;                          // covarianza
        public void destruir(AparatoAutomatizacion aa){... } // contravarianza
        public SmartPhone siguienteDestruir(){... }          // covarianza
}
public class LudditeSiglo21 extends Luddite {
        SmartPhone ultimoDestruido;                          // covarianza
        public void destruir(Maquina maquina){... }          // invarianza
        public SmartPhone siguienteDestruir(){... }          // covarianza
```

- Diferencia: el tipo del argumento es covariante, contravariante o invariante
- Similitud: el tipo del atributo y el del retorno son covariantes

Ejemplo 4: sobrescritura vs sobrecarga

- En cada una de las tres clases derivadas de la clase `Luddite`
  - ¿hay dos métodos `destruir` y dos métodos `siguienteDestruir`, uno definido en la clase misma y otro heredado de la clase base (sobrecarga)?
  - ¿o hay un solo método `destruir` y un solo método `siguienteDestruir`, siendo el método definido en la clase misma una versión distinta del método definido en la clase base (sobrescritura)?
- ¿En cuál de las tres subclases hay solo dos métodos?
  - Eso es, ¿en cuál de ellas hay sobrescritura de los dos métodos?
  - 1. `LudditeModerno`, 2. `LudditeNuevo`, 3. `LudditeSiglo21`
- Respuesta : distintas lenguajes tienen distinta política al respecto
  - 2 y 3: Sather, C# (si se declara explícitamente); sigue la teoría de tipos
  - Solo 3: Java, C++, Scala; respeta, pero restringe, la teoría de tipos
  - 1 y 3: Eiffel; contradice la teoría de tipos; se pierde la seguridad de tipos

Relación entre tipos de atributos de sub- y super- clase

- Suponiendo que
  - la clase `B` hereda de la clase `A`
  - Tanto `A` como `B` tiene un atributo llamado `x`

  ¿es cierta la siguiente afirmación (*covarianza*)?
  - El atributo `x` de `B` ocultará el atributo `x` de `A` si y solo si
    - el tipo de `x` en `B` es el mismo que, o hereda de, el tipo de `x` en `A`
- En la mayoría de los lenguajes OO, no.
  - En Java, la ocultación de atributos ni siquiera toma en cuenta el tipo
    - Hay ocultación si los dos atributos tienen el mismo nombre
    - Pero, de todos modos, la ocultación de atributos está desaconsejada
- Ver el apéndice para un tratamiento un poco más formal

Relación entre signaturas de métodos de sub- y super- clase

- Suponiendo que
  - la clase `B` hereda de la clase `A`
  - tanto `A` como `B` tiene un método `m`

  ¿es cierta la siguiente afirmación?
  - El método `m` de `B` sobrescribirá el método `m` de `A` si y solo si
    - el tipo de retorno de `m` en `B` es el mismo que, o hereda de, el tipo de retorno de `m` en `A` (*covariante en el tipo de retorno*)
    - el tipo de cada uno de los parámetros de `m` en `A` es el mismo que, o hereda de, el tipo del parámetro correspondiente de `m` en `B` (*contravariante en el tipo de los parámetros*)
  - Y lo mismo para la ocultación de los métodos estáticos
- En la mayoría de los lenguajes OO, no (Java, ver la página siguiente)
- Ver el apéndice para un tratamiento un poco más formal

La covarianza en el tipo de retorno en Java

- Está implementado desde Java 5
- El compilador interpretará…
  - un método de la subclase con mismo nombre y tipo de retorno covariante
    - como un método que sobrescribe a otro, no como un método sobrecargado
  - un método de la subclase con mismo nombre y tipo de retorno no covariante
    - como un error, puesto que tampoco puede interpretarse como sobrecarga
- Recuerde que la sobrecarga se resuelve en tiempo de compilación…
  - y si la asignatura de dos métodos difiere sólo en el tipo de retorno…
  - y esta diferencia no es la covarianza de tipos en el contexto de herencia…
  - el compilador no tiene manera de saber cuál de los dos métodos se pretende usar

La contravarianza en el tipo del argumento en Java

- No está implementado
- El compilador interpretará…
  - un método de la subclase con mismo nombre y tipo de arg. contravariante…
  - como un método sobrecargado, no como un método que sobrescribe a otro
- Para ser interpretado como un método que sobrescribe a otro en Java…
  - los tipos de los argumentos del método de la subclase y la superclase…
  - tienen que ser idénticos
- Nótese que la invarianza es una restricción de la covarianza
  - por lo que exigir que el tipo de argumento sea invariante…
  - no contradice las definiciones de la teoría de tipos, solo las restringe (*)

* Por el contrario, el lenguaje Eifel usa la covarianza en los argumentos, lo que sí contradice las definiciones de la teoría de tipos y, en consecuencia, viola la seguridad de tipos pero Bertrand Meyer argumenta que es útil.

Arrays covariantes y contravariantes

- Suponiendo que `B` es una subclase de `A`, hay tres posibilidades:
  1. *Arrays covariantes*: el compilador permite asignar un array de objetos de la clase `B` a una variable o parámetro cuyo tipo se ha declarado como `A[]`
  2. *Arrays contravariantes*: el compilador permite asignar un array de objetos de la clase `A` a una variable o parámetro cuyo tipo se ha declarado como `B[]`
  3. *Arrays invariantes*: el compilador no permite ninguno de los anteriores
- Solo la tercera posibilidad preserva la seguridad de tipos
  - Pero es demasiado restrictivo: no se podría definir métodos genéricos como `void shuffle(Object[] A);`
- Los arrays de Java son covariantes
  - De este modo, métodos declarados para operar sobre arrays de `Object` pueden usarse como métodos "genéricos" (muy usado en Java antes de v5)
  - Pero pueden surgir errores que no se detectan en tiempo de compilación

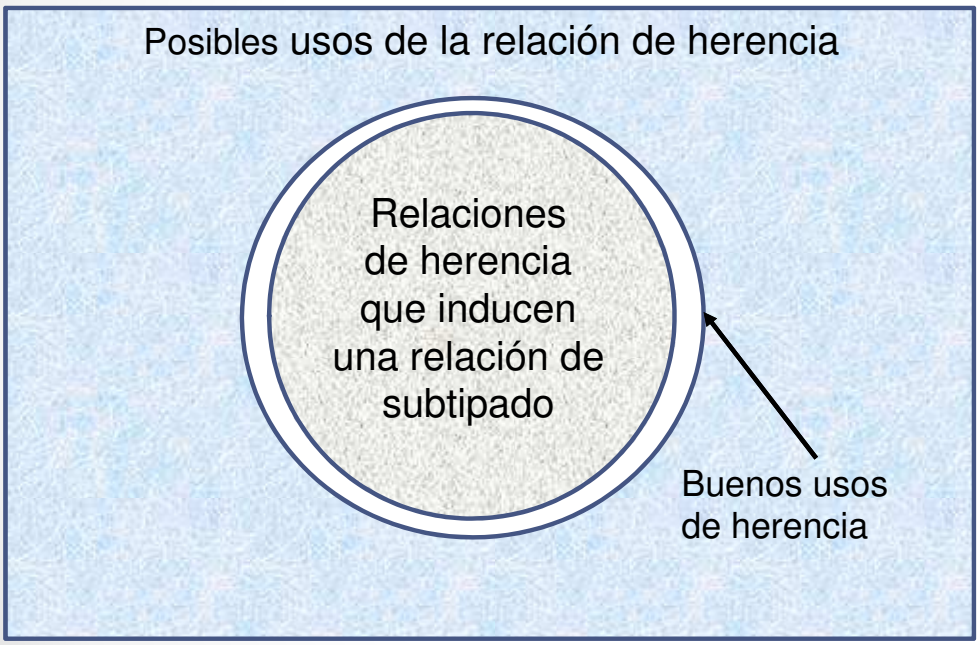
Ejemplo 6, colecciones covariantes

- Clases `Mamifero`, `Leon` y `Zebra` están relacionadas por herencia:
  - Un `Leon` es un `Mamifero` y una `Zebra` es un `Mamifero`
- Creamos un array de `Leon`: `Leon[] leones = new Leon[10];`
  - Lo almacenamos en la variable `leones`
- Puesto que los leones son mamíferos y los arrays son covariantes
  - Podemos hacer la asignación: `Mamifero[] mamiferos = leones`
- Luego insertamos una instancia de `Zebra` en el array `mamiferos`
  - sin que el compilador nos avise de un problema
- Hemos insertado una `Zebra` en un array de `Leon`
  - violando la seguridad de tipos (¡y la seguridad de la zebra!)
  - Puede provocar un error de tipo en tiempo de ejecución
    - P.ej. `for (Leon leon : leones) { // hacer algo leónico }`

Subtipos y subclases en la POO

- ¿Cuándo un tipo de datos es un subtipo de otro tipo de datos?
  - Informalmente, cuando *cualquier* operación sobre valores del supertipo
  - también operará correctamente sobre valores del subtipo
- Los lenguajes OO usan las clases como tipos
  - Se pueden declarar variables que tengan como tipo una clase
- Surge la pregunta: ¿una subclase define siempre un subtipo?
  - Es decir, ¿la relación de herencia entre clases induce una relación de subtipado (informalmente: "es_un") entre los tipos correspondientes?
    - Un mamífero es un vertebrado. Un alumno es una persona
  - Respuesta: en OO, todos los subtipos se inducen por la relación de herencia pero no todas las relaciones de herencia inducen subtipos
    - Herencia que no induce subtipado se llama *herencia de implementación*

Subtipos y subclases en la POO



Resumen

→ Hay muchos posibles malos usos del mecanismo de herencia

→ Si una relación de herencia induce una relación de subtipado, es un buen uso

Definición de subtipo compatible con la herencia

- El principio de sustitución de Liskov:
  - La clase `B` es un subtipo de la clase `A` si, en *cualquier* circunstancia, la sustitución de una instancia de la clase `A` por una instancia de la clase `B` no da lugar a ningún cambio en el comportamiento observable
- Formalmente, `B` es un subtipo de `A` ⇔
  - → Para cada método `m` de `B` que sobrescribe un método `m` de `A`
    - La precondición de `m` en `B` es igual o más débil que la de `m` en `A`
    - La poscondición de `m` en `B` es igual o más fuerte que la de `m` en `A`
  - → `B` preserva los mismo invariantes que `A` (puede añadir más)
  - → Todos los métodos de `B` preservan los invariantes de `B`
    - incluyendo los métodos que se heredan de `A`
- Subtipado comportamental fuerte (*strong behavioural subtyping*)
  - Desafortunadamente, no es decidible (el compilador no te puede ayudar)

Ejemplo 5: subclases y subtipos

- ¿Podemos derivar una clase `Cuadrado` de la clase `Rectángulo`?:

```
public class Rectangulo {

        private int base;
        private int altura;

        public Rectangulo(int b, int a){ base=b; altura=a; }

        public setBase(int b){ base=b; }

        public setAltura(int a){ altura=a; }

        public int area(){ return base * altura; }
}
```

Ejemplo 5: subclases y subtipos

```
public class Cuadrado extends Rectangulo {
    public Cuadrado(int lado){
        super(lado, lado);
    }
}
```

- ¿Bien? ¿Qué nos impide usar esta clase como sigue?

```
Cuadrado c = new Cuadrado(10);
c.setBase(20);
```

- Idem si `Rectangulo` tiene un método `estirar(int factor)` que multiplica la base (pero no la altura) por un `factor`

```
Cuadrado c = new Cuadrado(10);
c.estirar(2);
```


Ejemplo 5: subclases y subtipos

- ¿Cuál es el problema?
  - `Cuadrado` es una subclase pero no es un subtipo de `Rectangulo`
    - tal como están escritas las clases aquí
  - Concretamente, la subclase `Cuadrado` de la clase `Rectangulo` no respeta el principio de sustitución de Liskov
    - La subclase tiene una invariante adicional
- Más formalmente
  - Se viola el principio de sustitución de Liskov porque
    - la clase base contiene métodos que cambian el estado
    - y, al hacerlo, pueden violar un invariante de la clase derivada
  - Ej: si la clase `Rectangulo` no tiene *setters*, e incluye las coordenadas de la esquina inferior izquierda y un método `m()` de traslado en el espacio, aunque `m()` cambia el estado, no viola el invariante de `Cuadrado`

Herencia y visibilidad

- ¿Cómo afectan los modificadores de visibilidad de Java que vimos antes a los miembros heredados?
  - **`public`**: accesible desde los descendientes
  - El valor por defecto, *package private*: accesible desde los descendientes que están en el mismo *package*
  - **`private:`** no accesible desde los descendientes.
  - **`protected:`** accesible desde los descendientes
- Recuerde que miembros protegidos también son accesibles desde objetos de clases del mismo *package*
  - por razones históricas
  - en las primeras versiones de Java, no existía la visibilidad `private`

Clases abstractas

- En la parte más alta de una jerarquía de herencia
  - se encuentran las clases que representan los conceptos más abstractos
- Una clase abstracta
  - no sirve para crear objetos de esta clase
  - sirve para que se deriven nuevas clases de ella
    - Una clase abstracta no se puede instanciar
    - Solo se pueden instanciar clases no abstractas derivadas de ella
    - Las clases no abstractas derivadas representan conceptos concretos
- Una clase abstracta obliga a que todas las clases derivadas
  - tengan un cierto conjunto de métodos
  - sin especificar el comportamiento de todos ellos

Clases abstractas

- Los métodos de una clase abstracta:
  - o bien contienen implementaciones comunes a cada una de sus subclases
  - o bien han de ser implementados en cada una de sus subclases
    - Son *métodos abstractos* (o *métodos virtuales*), es decir, no tienen cuerpo
- Una clase es abstracta ⇔ al menos uno de sus métodos es abstracto
- En Java
  - Se utiliza la palabra reservada `abstract`
    - para declarar tanto clases abstractas como métodos abstractos
  - Una clase abstracta
    - puede tener atributos de instancia y atributos estáticos
    - puede tener métodos estáticos (abstractos o no)
    - puede tener constructores que se llamarán desde un constructor de subclase
    - no pueden tener métodos privados
      - No podrían llamarse nunca

Ejemplo 7: clases abstractas

- Creamos una clase abstracta `Trabajador`
  - que contiene el método abstracto `trabajar()` que se implementará (de manera distinta, se supone) en cada clase de trabajador derivada

```
abstract class Trabajador {
        abstract public boolean trabajar();
}
```

- Esto obliga a proporcionar un cuerpo para el método `trabajar()` en las clases derivadas
  - Si no, se hereda el método abstracto
  - y la clase derivada debe ser abstracta también

Ejemplo 7: Las clases abstractas no se pueden instanciar

- Una clase abstracta no se puede usar para crear un objeto

```
Trabajador trabajador = new Trabajador();   // error
```

- Sólo se pueden crear objetos de clases derivadas de la clase abstracta

```
public class TecnicoDeSuperficies extends Trabajador {
    private int horas;
    public TecnicoDeSuperficies(int tipo) { horas = 10*tipo; }
    public void toScreen() {System.out.println("T: " + horas);}
    public boolean trabajar() { … }
}

// ¡Ahora sí!
TecnicoDeSuperficies t1 = new TecnicoDeSuperficies(4);
```


Ejemplo 8: clases abstractas

```
abstract public class Figura {
        // centro de la figura
        protected int x;
        protected int y;

        public Figura(){
                x=0; y=0;
        }

        public Figura(int xx, int yy){
                x = xx; y = yy;
        }

        abstract public int area();
        abstract public void dibujarFigura();
}
```

Ejemplo 8: clases abstractas

```
public class Rectangulo extends Figura {

    private int base;
    private int altura;

    public Rectangulo(int xx, int yy, int b, int a){
        super(xx,yy);
        base=b; altura=a;
    }
    public int area(){
        return base * altura;
    }
    public void dibujarFigura(){…}

}
```


Ejemplo 8: clases abstractas

```
public class Cuadrado extends Figura {
     private int lado;

public Cuadrado(int xx, int yy, int side){
     super(xx,yy);
     lado = side;
}

public int area(){
     return lado*lado;
};

public void dibujarFigura(){…}
}
```

Interfaces

- Una interfaz es como una clase abstracta en la que
  - todos los métodos son abstractos
- Una interfaz consta de la declaración de
  - un conjunto de métodos (sin implementación)
  - unos valores constantes (*Constant Interface Antipattern*: Joshua Bloch)
  - unos atributos: sólo en algunos lenguajes (**no en Java**)
    - se interpreta como la declaración de un *getter* y, posiblemente, un *setter* (sólo *getter* si el atributo se declara como “readonly”)
- Las interfaces
  - sirven para capturar similitudes entre clases
    - sin forzar una relación de herencia entre ellas
  - se pueden usar como la base de un *protocolo de comportamiento*
    - El comportamiento en sí se añade en cada clase que la implementa

Interfaces

- Una interfaz es un tipo de datos de referencia
  - Puede utilizarse en cualquier lugar donde se espera un tipo
    - P.ej. en el argumento de un método o en la declaración de una variable
  - Cualquier objeto que implementa la interfaz cumplirá una tal declaración
- La herencia de interfaces induce una relación de subtipado
  - Una subinterfaz siempre define un subtipo
    - aunque a veces sobrecarga donde según teoría de tipos debe haber sobrescritura
    - *Desafortunadamente, ya no es cierto en Java 8 (ver más adelante)*
- Las interfaces se implementan en clases
  - Una clase que implementa una interfaz sin proporcionar una implementación para todos sus métodos tiene que ser abstracta

Interfaces en Java

- Se declaran como clases
  - pero con la palabra reservada `interface` en vez de `class`
- Todos sus métodos son públicos y sus atributos constantes
  - *Desde Java 8, se puede omitir el modificador de método* `public`
  - *y se pueden omitir los modificadores de atributo* `public static final`
- Sólo declaran signaturas de métodos (no definen cuerpos)
  - No declaran constructores
  - *Desde Java 8, se permiten cuerpos de ciertos métodos (ver más adelante)*
- Pueden heredar de una o varias otras interfaces
  - Sin implementaciones, la herencia múltiple no puede dar problemas
  - *Desde Java 8, las interfaces pueden contener implementaciones. ¡Cuidado!*

Interfaces en Java

- Normalmente van en un fichero `.java` independiente
  - igual que una clase
- Ejemplo de interfaz Java (Ejemplo 9):

```
public interface EjI1 {
        public void metodo1();
        public Integer metodo2();
        public Integer metodo3();
}

public interface EjI2 {
        public void metodo4();
        public Integer metodo5();
        public Integer metodo6();
}
```

Implementar una interfaz Java

- Se utiliza la palabra reservada `implements`
  - seguida por los nombres de una o más interfaces separadas por comas
  - para denotar que una clase implementa una interfaz
- La clase implementadora debe declarar todos los métodos
  - Si deja uno sin implementar, la clase debe declararse como abstracta
- Ejemplo 9:

```
class EjemploImpl implements EjI1, EjI2 {
        public void metodo1()   { ....}
        public Integer metodo2(){ ....}
        public Integer metodo3(){ ....}
        public void metodo4()   { ....}
        public Integer metodo5(){ ....}
        public Integer metodo6(){ ....}
}
```


Usar una interfaz Java como un tipo

- Se puede declarar una variable cuyo tipo es una interfaz
  - y luego asignarle cualquier objeto que implementa esta interfaz
  - Después se puede llamar a los métodos de la interfaz sobre esta variable
  - pero no a otros métodos que el mismo objeto implementa
- Ejemplo 9:

```
class EjemploImpl implements EjI1, EjI2 {
    public static void main(String args[]){
        EjemploImpl ejemplo = new EjemploImpl();
        EjI1 i1 = ejemplo; // tiene los métodos de EjI1
        EjI2 i2 = ejemplo; // tiene los métodos de EjI2
        i1.metodo3()       // OK
        i1.metodo4();      // error (EjI1 no tiene metodo4)
    }
}
```

Algunas interfaces de las bibliotecas Java

- Más tarde, veremos algunas interfaces de las bibliotecas Java, p.ej.
  - `Serializable` (interfaz "de marcado", es decir, sin métodos)
    - Sirve para indicar que una clase puede ser convertida en un flujo de bits (para poder escribirla en fichero o enviarla por la red)
  - `Runnable`
    - Se implementa si se quiere la funcionalidad de crear un nuevo hilo
    - Tiene un método `run()` (con una semántica parecida al `main()`)
  - `ActionListener`
    - Se implementa si se quiere la funcionalidad de ejecutar acciones particulares cuando se producen ciertos eventos
- La clase implementador sólo tiene que implementar los métodos
  - Tiene libertad para determinar estructuras de datos, métodos auxiliares

Resumen de Interfaces

- Una clase puede implementar varias interfaces
  - tiene que declarar todos los métodos de la interfaz
  - si deja alguno sin implementación, la clase tiene que ser abstracta
- Jerarquías de herencia de interfaces
  - Interfaces pueden heredar de (extender a) otras interfaces
  - Se permite la herencia múltiple (heredar de varias interfaces)
- Las interfaces son tipos
  - Se puede declarar una variable cuyo tipo es una interfaz
  - Se le puede asignar un objeto de una clase que implementa la interfaz
- Las interfaces no definen contratos
  - a pesar de lo que digan algunos autores
  - No restringen nada al comportamiento de los métodos declarados

Interfaces vs. clases abstractas: clases abstractas Java

- Una clase abstracta de Java sólo puede heredar de una clase
  - ¡como cualquier clase!
- La clase que la hereda no puede heredar de ninguna otra clase
- Sus métodos pueden ser `public`, `protected` o *package-private*
- Puede tener atributos (y, por tanto, estado)
- Incluye la implementación de algunos de sus métodos
- La clase que la hereda no está obligada a incluir todos sus métodos
  - Si la clase heredera no incluye alguno de los métodos, heredará el método sin cambios de la clase abstracta
  - Si el método en cuestión es abstracto, será abstracto en la clase heredera también y la clase heredera será abstracta

Interfaces vs. clases abstractas: interfaces Java

- Una interfaz de Java puede heredar de múltiples interfaces
- La clase que la implementa puede también implementar otras interfaces
- Los métodos son todos `public` (*declaración implicita desde Java 8*)
- No puede tener atributos que no sean constantes
- No incluye ninguna implementación de los métodos
  - *No es cierto para una interfaz de Java 8 (ver más adelante)*
- La clase que la implementa no está obligada a incluir todos sus métodos
  - Si la clase implementadora no incluye alguno de los métodos, está declarado implícitamente como abstracto pero la clase tiene que declararse abstracta
    - Si se extiende una interfaz Java7, la clase que la implementa no se compilará
  - *Java 8 permite métodos default in interfaces (ver más adelante)*
    - Si se añade un método default a una interfaz, la clase que la implementa compila
    - salvo si ya implementa una interfaz con método default con la misma signatura

¿Cuándo usar una interfaz o clase abstracta?

- Usar una clase abstracta cuando
  - se quiere compartir código entre clases muy relacionadas
  - se anticipa que las clases que extenderán a la clase abstracta
    - tendrán muchos atributos e implementaciones de métodos en común
    - o se quiere que hereden métodos que no son `public`
  - se quieren declarar atributos que no son constantes (estado de objeto)
- Usar una interfaz cuando
  - se anticipa que las clases que implementarán la interfaz
    - no tendrán mucha relación, p.ej. las que implementan `Comparable`
    - o serán de distintas partes de una jerarquía de herencia
  - se quiere especificar un tipo de datos
    - sin saber, o sin preocuparse por, cuáles son las clases que lo implementarán
  - se quiere aprovechar la herencia múltiple

¿Por qué preferir interfaces a clases abstractas?

- Las interfaces facilitan la adición de nueva funcionalidad
  - Es fácil adaptar clases existentes para que implementen nuevas interfaces
    - independientemente de si ya implementan otras interfaces
  - Se pueden usar para añadir funcionalidad mediante una clase envoltura
    - Una clase que implementa las interfaces requeridas, implementando algunos de los métodos y delegando llamadas a los demás en la clase envuelta
  - Se puede proporcionar una implementación parcial en una clase abstracta
    - Solo se puede añadir funcionalidad a una clase abstracta vía herencia
- Las interfaces son ideales para especificar comportamientos auxiliares
  - Llamados *mixin*, p.ej. la interfaz `Comparable`
- Permiten la construcción de estructuras de tipos no jerárquicas
  - Una jerarquía no siempre es la estructura más adecuada

¿Por qué preferir interfaces a clases abstractas?

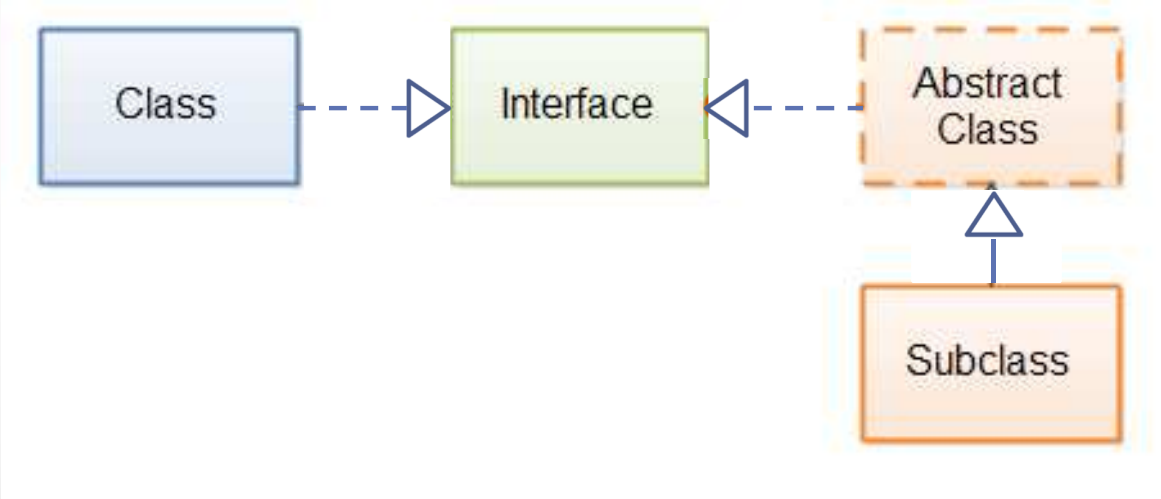
- Por otro lado, las interfaces son difíciles de "evolucionar"
  - P.ej. Si se añade un nuevo método a una interfaz, todas las clases que la implementaban ya no la implementarán y no compilarán
- Con los métodos `default` de Java 8
  - se busca paliar este problema
    - En particular, los diseñadores del lenguaje querían cambiar muchas de las interfaces de las bibliotecas estándar
    - para poder aprovechar los expresiones lambda de Java 8

¿Qué son las expresiones lambda de Java 8?

- ¿Qué es la programación funcional (base teórica: lambda cálculo)?
  - Excelente para la programación concurrente o dirigido por eventos
  - Funciones son "objetos de primera clase"
    - El sistema de tipos incluye tipos de función
    - Se puede pasar (limpiamente) una función como argumento a otra función
      - "limpiamente" no describe el pasar punteros de función sin tipo en C (i.e. `void*`)
      - "limpiamente" no describe el pasar punteros de función tipados en C/C++ (en particular, con `typedef`)
- Expresiones lambda: introducir la programación funcional en Java
  - Métodos que se pueden pasar como argumentos a otros métodos
  - Al añadir expresiones lambda a Java no se ha añadido tipos de función
    - El tipo de una expresión lambda es una interfaz funcional (a menudo inferida)
      - Inferencia de tipo: tipo de retorno inferido siempre, de parámetros inferido a veces
      - También se puede usar una interfaz funcional genérica

Uso conjunto de interfaces y clases abstractas

- Puede ser útil usar interfaces junto con clases abstractas
  - Implementaciones estándar heredan de la clase abstracta
    - *Java 8: solo se necesita si las implementaciones estándares incluyen estado*
  - Versiones más específicas implementan la interfaz



Interfaces en Java 8 (ver transparencias 54 y 55)

- Interfaces sólo declaran signaturas de métodos (no definen cuerpos)
  → **Desde Java 8, las interfaces pueden incluir implementaciones por defecto e implementaciones de métodos estáticos**
- La herencia entre interfaces induce una relación de subtipado
  - Una subinterfaz siempre define un subtipo
  → **En Java 8, con la introducción de métodos `default` y métodos estáticos con implementación, no siempre será cierto**
- Interfaces pueden heredar de una o varias interfaces
  - Sin implementaciones, la herencia múltiple no puede dar problemas
  → **En Java 8, con la introducción de métodos `default` y métodos estáticos con implementación, pueden surgir conflictos de nombre**
  → **Se desambiguan como en C++ (y con una sintaxis parecida)**

Ejemplo 10: métodos `default` en las interfaces Java 8

- Interfaz `java.lang.Iterable` en Java 7

```
public interface Iterable<T> {
    public Iterator<T> iterator();
}
```

- Interfaz `java.lang.Iterable` en Java 8 (con 2 nuevos métodos)

```
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> consumer) {
        for (T t : this) { consumer.accept(t); }
    }
    default Spliterator<T> spliterator {
        // … content of default method omitted here
    }
}
```


Clases finales

- Una clase `final` no puede usarse para derivar nuevas clases
  - El compilador señala un error si se intenta crear una subclase
- Una clase se puede hacer `final` por al menos dos razones
  - Para asegurar que no se puede crear subclases de una clase fundamental
    - con propiedades extrañas, incoherentes, indeseables o impredecibles
    - Por ejemplo, una clase inmutable como `java.lang.String`
  - Para evitar que se puedan introducir cambios en la superclase
    - que afecten a las subclases de manera imprevista
    - Ver problema de la "fragilidad de la clase base"
  - "*Diseña y documenta para la herencia o bien prohíbela*" Joshua Bloch

Clases finales

- Ejemplo

```
public final class B extends A {
   ...
}

public class C extends B {    // error
   ...
}
```


Métodos finales

- Un método `final` no puede ser redefinido en una subclase
  - El compilado señala un error si se intenta sobrescribirlo
- Una razón para hacer `final` a un método es
  - si su funcionamiento es crítico para mantener coherente el estado del objeto
  - Por ejemplo los métodos `wait()` y `notify()` de la clase `Object` usados en la programación concurrente
- Un método que se llama desde un constructor debería ser `final`
  - Si no, podría ser sobrescrito en una subclase con resultados inesperados
  - Por ejemplo, un método `inicializar` llamado desde el constructor
    - p.ej. para evitar duplicación de código entre constructor y método `reset()`

Métodos finales

- Ejemplo

```
public class A {
        final void mover(int dx, int dy) {
                ...
        }
}

public class B extends A {
        void mover(int dx, int dy) { // error
                ...
        }
}
```


Constructores y métodos finales

- Sobrescribir un método llamado desde un constructor puede ser peligroso

```
class Padre {
        public Padre() { metodoSobrescrito(); }
        public void metodoSobrescrito() {
                System.out.println("Inicializar Padre");
        }
}

class Hija extends Padre{
        public Hija() { metodoSobrescrito(); }
        public void metodoSobrescrito() {
                System.out.println("Inicializar Hija");
        }
        // imprime "inicializar Hija" dos veces
        public static void main(String [] args) {
                Hija h = new Hija();
        }
}
```

Sintaxis Java, modificadores

- Las palabras clave utilizadas para especificar la visibilidad
  - se llaman “modificadores de acceso”
- Las otras palabras clave que se pueden colocar antes de declaraciones
  - se llaman “modificadores no de acceso”
- Resumen de modificadores
  - Atributo: {`private` | `public` | `protected`} {`final`} {`static`}
  - Método: {`private` | `public` | `protected`} {`final` | `abstract`} {`static`}
  - Clase: {`public`} {`final` | `abstract`}

  Sólo se puede usar una palabra de cada par de llaves

Problemas de la herencia múltiple, Ejemplo 11: conflicto de nombres en C++

```
// método virtual de C++ ~ método que no es final de Java
#include <iostream >
using namespace std;

class Padre1 {
    public:
        void mostrar(){ cout << "Padre1"; };
};

class Padre2 {
    public:
        void mostrar(){ cout << "Padre2"; };
};

class Hija: public Padre1, public Padre2 { };
```

Problemas de la herencia múltiple, Ejemplo 11: conflicto de nombres en C++

```
int main(){
    Hija hija = Hija();
    hija.mostrar(); // error: acceso ambiguo de 'mostrar', puede
                    // ser el mostrar de la clase 'Padre1' o
                    // puede ser el 'mostrar' de la clase 'Padre2'
}
```

- Para solucionar el problema el lenguaje debe aportar
  - algún mecanismo (y posiblemente sintaxis) que permita desambiguar
  - En el caso de C++, la sintaxis es como sigue:

```
hija.Padre1::mostrar() // Llama a la implementación de Padre1
hija.Padre2::mostrar() // Llama a la implementación de Padre2
```

- Alternativamente, sobrescribir el método `mostrar` en `Hija`

Problemas de la herencia múltiple Ejemplo 12: conflicto de nombres en Java 8

```
interface Padre1 {
    default void mostrar() { System.out.println("Padre1"); }
}
interface Padre2 {
    default void mostrar() { System.out.println("Padre2"); }
}

class Hija implements Padre1, Padre2 { };

class Main{
    public static void main(String[] args){
        Hija hija = new Hija();
        hija.mostrar(); // error a la hora de compilar
    }
}
```

Para desambiguar: invocar `Padre1.super.mostrar()` o `Padre2.super.mostrar()` o sobrescribir el método `mostrar` en `Hija`

Problemas de la herencia múltiple, Ejemplo 13: herencia en diamante en C++

```
#include <iostream>
using namespace std;

class Abuelo {
    public:
        void mostrar1 (){
             cout << «Abuelo" << problematico << endl;
        };
        int problematico;
};

class Padre1: public Abuelo {
    public:
        Padre1 () { problematico = 1; }
        void mostrar2(){ cout << "Padre1" << endl; };
};
```


Problemas de la herencia múltiple, Ejemplo 13: herencia en diamante en C++

```
class Padre2: public Abuelo {
    public:
        Padre2() { problematico = 3; }
        void mostrar3(){ cout << "Padre2"; << endl; };
};

class Hija: public Padre1 , public Padre2 {
};

int main(){
    Hija hija=Hija();
    hija.mostrar2();
    hija.mostrar3();
    hija.mostrar1();  // error: ¿por cuál de las rutas llamamos?
    int newvar = hija.problematico; // ¿Cuál de los dos valores?
    cout << newvar << endl;
}
```

Problemas de la herencia múltiple
Ejemplo 13: herencia en diamante en C++

- Problema 1: duplicación de comportamiento heredado vía estructura en diamante
  - `Hija` contiene *dos métodos* `mostrar1`, uno heredado a través de `Padre1` y otro heredado a través de `Padre2`, por lo que el código *no compila*.
  - Solución 1: desambiguar el conflicto de nombres explícitamente como en ejemplo 11; sin embargo, lo más probable es que no hubiera intención de definir dos métodos.
  - Solución 2: especificar herencia *virtual* de `Abuelo`, para asegurar que existe una única copia de sus métodos & atributos.
- Problema 2: duplicación de estado heredado vía la estructura en diamante
  - `Hija` contiene *dos atributos* `problematico`, uno heredado a través de `Padre1` y otro heredado a través de `Padre2`, por lo que el código *no compila*.
  - Solución 1: desambiguar el conflicto de nombres explícitamente como en ejemplo 11; sin embargo, lo más probable es que no hubiera intención de definir dos atributos.
  - La herencia virtual no soluciona el problema de duplicación de estado heredado
    - `Hija` tiene una sola copia de `problematico` y `mostrar1` pero la ruta de herencia de `mostrar1` ya no puede especificarse explícitamente; resultado: ¡*comportamiento no definido*!

Problemas de la herencia múltiple
¿Hay problema de herencia en diamante en Java 8?

```
interface Abuelo {
    default void mostrar(){ System.out.println("hola") }
}
interface Padre1 extends Abuelo {}
interface Padre2 extends Abuelo {}
class Hija implements Padre1, Padre2 {}
```

- Problema 1: duplicación de comportamiento heredado vía estructura en diamante
  - Implementación similar a la herencia virtual de C++, es decir, `Hija` tiene acceso no ambiguo a la única implementación existente, por tanto no hay error de compilación.
  - No hay tal problema de diamante en Java 8.
- Problema 2: duplicación de estado heredado vía la estructura en diamante
  - Las interfaces no contienen atributos (salvo constantes)
  - No hay tal problema de diamante en Java 8.

Clase `Object`

- `Object` define un conjunto de métodos útiles, que pueden ser redefinidos en cada clase. En particular:
  - `public boolean equals(Object o)`
    - Permite definir una noción de igualdad para los objetos de la clase
  - `public hashCode()`
    - Devuelve un valor de hash para el objeto (debe ser coherente con `equals()`)
  - `public String toString()`
    - Permite definir la representación textual de los objetos de la clase
  - `public Object clone()`
    - Crea y devuelve una copia del objeto
  - `public Class getClass()`
    - Devuelve la clase a la que pertenece el objeto con el que se le invoca

Clase `Object`, método `equals()`

- Redefinición de `equals()` (por defecto, lo mismo que ==)

```
public class Otra{
        private int x;

        public boolean equals (Object obj) {
                if (this == obj) return true;
                if (obj == null) return false;
                if (this.getClass() != obj.getClass())
                        return false;

                Otra otra = (Otra) obj;
                return otra.x == this.x;
        }
}
```

Interfaces, métodos implícitos

- Consideremos el ejemplo 9 (ver transparencias 56-58)
- El tipo de la variable `i1` es la interfaz `EjI1` y, por tanto,
  - `i1` sólo puede usar los métodos de la interfaz `EjI1`, en particular
    - la llamada `i1.metodo4()` está señalada como un error por el compilador
- Pues, ¿el compilador aceptará la llamada `i1.toString()`?
  - En caso afirmativo, ¿por qué?
  - Ver *Java Language Specification*, Sección 9.2. "*Interface Members*"
    `http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.2`
  - Con Java 8 (casi) podría realizarse esta "herencia" dentro del lenguaje
    - Declarando los métodos de `Object` como métodos `default` en interface `Iobject`
    - ¿Y el "casi"?: ¿qué pasa con los métodos de `Object` que son `final`?

Polimorfismo

- Un lenguaje de programación tipado tiene polimorfismo si
  - hay construcciones que pueden tener múltiples tipos
- Tiene polimorfismo por inclusión (o polimorfismo de subtipos) si
  - utiliza subtipado, definido habitualmente vía una noción de substitución
  - las operaciones de un supertipo pueden operar sobre sus subtipos
- Una variable polimórfica es
  - una variable que puede contener valores de múltiples tipos
- Con el polimorfismo por inclusión, una variable se declara de un tipo
  - pero durante la ejecución puede contener valores de un subtipo
  - Nótese que se está identificando la noción de subclase con la de subtipo

Polimorfismo

- Supongamos que la clase `Perro` hereda de la clase `Animal`

```
public class Animal { ...  }
public class Perro extends Animal { ... }
```

- Una variable (polimórfica) declarada de tipo `Animal`
  - puede contener un objeto del subtipo `Perro`

```
Animal a = new Perro(); // OK, un Perro es un Animal
```

  - pero al revés no es cierto

```
Perro d = new Animal(); // error, un Animal no es un Perro
```

  - tampoco puede el valor de una variable de tipo declarado `Animal` asignarse a una variable de tipo declarado `Perro`, aunque su tipo real sea `Perro`

```
Perro d = a; // error, tipo declarado de a no es Perro…
             // … ni un subtipo de Perro
```


Polimorfismo, tipo estático y tipo dinámico

- El polimorfismo implica que toda variable tiene un
  - *tipo estático*: el tipo utilizado en la declaración de la variable
  - *tipo dinámico*: el tipo del objeto contenido en una variable.
- En el ejemplo de la página anterior
  - `Animal` es el tipo estático de la variable a
  - `Perro` es el tipo dinámico de la variable a.
- El tipo estático de una variable
  - se conoce en tiempo de compilación
- El tipo dinámico de una variable
  - puede no conocerse hasta tiempo de ejecución
  - puede cambiar durante una ejecución

Polimorfismo

- Java solo permite el acceso a los miembros (métodos y atributos) del tipo estático, p.ej.

```
public class A
{ ... public void incX() { x++; } ... }

public class B extends A
{ ... public void incZ() { z++; } ... }

A a = new A(); B b = new B(); A aa = new B();
a.incX();  // Ok
a.incZ();  // error, violación de tipo
b.incX();  // Ok
b.incZ();  // Ok
aa.incX(); // Ok
aa.incZ(); // error, violación de tipo
```


Vinculación dinámica

- ¿Cómo combina el polimorfismo con la rescritura?

```
public class A {
  private int x, y;
  public void print() { System.out.println(x + " " + y); }
}

class B extends A {
  private int z;
  void print() { super.print(); System.out.println(" " + z); }
}
```

- Cuál de los métodos `print` se invoca en el código siguiente?

```
A a = new B(1, 2, 3);
a.print();
```

Vinculación dinámica

- Se invoca el método del tipo dinámico de la variable `a`
  - Es decir, el método imprime: 1 2 3
  - Recuerde que, en el caso general, no se conoce el tipo dinámico en tiempo de compilación
- Vinculación dinámica
  - En presencia de métodos sobrescritos, el acto de asociar en tiempo de ejecución una llamada de un método a una implementación concreta del mismo
- La regla para elegir una implementación de método:
  - Utilizar la del tipo más específico posible
    - En la jerarquía de herencia, el primero que tenga el método en la secuencia entre el tipo dinámico y la raíz

Ejemplo 14: polimorfismo y vinculación dinámica

```
public abstract class Animal {
  public abstract comer();
}

public class Perro extends Animal {
  public void comer() { ... }
}

public class Gato extends Animal {
  public void comer() { ... }
}
```

Ejemplo 14: polimorfismo y vinculación dinámica

```
public class Test {

   public static void main (String args[]) {
      Animal granja[] = new Animal[2];
      granja[0] = new Perro();
      granja[1] = new Gato();
      for (int i = 0;i < 2; i++) {
         granja[i].comer();          // vinculación dinámica
       }
   }
}
```


Sobrecarga y sobrescritura (otra vez)

- Sobrecarga
  - Dos métodos tienen el mismo nombre pero difieren en el número y/o el tipo de los parámetros
  - Las signaturas no pueden tener como única diferencia el tipo de retorno
    - Si se llamara al método sin asignar el valor de retorno a una variable (tipada), el compilador no podría saber qué versión usar
  - *Se resuelve en tiempo de compilación usando tipos estáticos*
- Sobrescritura (o redefinición)
  - Una clase hija redefine un método de su superclase
    - En Java, el tipo de los parámetros tiene que ser igual y el del valor de retorno covariante
  - *Se resuelve en tiempo de ejecución usando tipos dinámicos*

Sobrecarga y polimorfismo en Java

- Elección entre métodos sobrecargados se hace en tiempo de compilación
  → Solo se puede usar el tipo estático de parámetros polimórficos

```
class A {
    public void hacerAlgo(Object a){
        System.out.println("El objeto NO es de tipo A");
    }
    public void hacerAlgo(A a){
        System.out.println("El objeto es de tipo A");
    }
    public static void main(String args []){
        A a = new A(); Object b = a; Object c = new A();
        A comprobador = new A();
        comprobador.hacerAlgo(a); // ¿Qué imprime?
        comprobador.hacerAlgo(b); // ¿Qué imprime?
        comprobador.hacerAlgo(c); // ¿Qué imprime?
    }
}
```


Vinculación dinámica, resumen

- Una jerarquía de herencia puede implicar sobrescritura de métodos
- Dadas la siguiente asignación y llamada de método:

```
A a = new B(….);   a.p(….);
```

la decisión sobre qué método aplicar en tiempo de ejecución se toma según las siguientes reglas:

1. Si el método `p(…)` no está definido en el tipo estático `A`, error; está definido si su declaración se encuentra en el camino por la jerarquía de herencia entre la clase `A` y la raíz (la clase `Object` o una interfaz).
2. Si está definido en el tipo estático, la implementación de `p(…)` que se utilizará es la primera que se encuentra cuando se recorre la jerarquía de herencia desde la clase `B` hasta la raíz.

Ejercicio: vinculación dinámica

```
class A {
    public void p(int x){ System.out.println("p:A"); }
}

class B extends A { }

class C extends B { }

class D extends C { }

B b = new C(); // tipo estático B, tipo dinámico C
b.p(2);        // ¿error? ¿qué es lo que se imprime?
```


Ejercicio: vinculación dinámica

```
class A {
    public void p(int x){ System.out.println("p:A"); }
}

class B extends A {
    public void p(int x){ System.out.println("p:B"); }
}

class C extends B { }

class D extends C { }

B b = new C(); // tipo estático B, tipo dinámico C
b.p(2);        // ¿error? ¿qué es lo que se imprime?
```

Ejercicio: vinculación dinámica

```
class A { }

class B extends A {
    public void p(int x){ System.out.println("p:B"); }
}

class C extends B {
    public void p(int x){ System.out.println("p:C"); }
}

class D extends C { }

B b = new C(); // tipo estático B, tipo dinámico C
b.p(2);        // ¿error? ¿qué es lo que se imprime?
```


Ejercicio: vinculación dinámica

```
class A { }

class B extends A { }

class C extends B {
    public void p(int x){ System.out.println("p:C"); }
}

class D extends C { }

B b = new C(); // tipo estático B, tipo dinámico C
b.p(2);        // ¿error? ¿qué es lo que se imprime?
```

Ejercicio: vinculación dinámica

```
class A { }

class B extends A { }

class C extends B { }

class D extends C {
    public void p(int x){ System.out.println("p:D"); }
}

B b = new C(); // tipo estático B, tipo dinámico C
b.p(2);        // ¿error? ¿qué es lo que se imprime?
```


Ejercicio: vinculación dinámica

```
class A { }

class B extends A { }

class C extends B { }

class D extends C {
    public void p(int x){ System.out.println("p:D"); }
}

D d = new D(); // tipo estático D, tipo dinámico D
d.p(2);        // ¿error? ¿qué es lo que se imprime?
```

Polimorfismo y *downcasting*

- Se puede cambiar explícitamente el tipo estático de una variable a su tipo dinámico mediante un *cast,* llamado, en este caso, un *downcast*

```
A a = new B();    // Clases definidas en el código de la P.89
B b = (B)a;       // La violación de tipo de la P.87 no ocurre
b.incZ();         // La violación de tipo de la P.89 no ocurre
( (B)a ).incZ(); // Alternativa a las dos últimas líneas
```

- Si el tipo estático de una variable es `A` y el tipo dinámico es `B`…
  - un downcast de la variable al tipo dinámico `B` permite…
  - la invocación subsiguiente de métodos de la clase `B` que no están declarados en la clase `A` (ni en ninguna de sus superclases)
  - Por ejemplo, en el código de la página 89, la invocación `b.incZ()`

Polimorfismo y *downcasting*

- No todo objeto de la clase `A` se puede convertir a la clase `B`

```
A a = new A();
B b = (B)a; // error a run-time ya que tipo dinámico no es B
```

- En Java se comprueban todas las conversiones explícitas en tiempo de ejecución
  - Si el valor de la variable, parámetro de retorno, etc. no pertenece a la clase a la cual se pretende convertir, ocurre una excepción
- La conversión de tipos explícita debería evitarse donde sea posible
  - Es como decir al compilador “no compruebes el tipo, confía en mí”
- La necesidad de usar `downcasting`
  - se considera un indicio de un mal diseño

Casting y la comparación de tipos

- Antes del cast, para evitar una excepción en tiempo de ejecución
  - es aconsejable comprobar el tipo dinámico del elemento que se pretende convertir
- En Java, se puede comprobar el tipo dinámico de dos maneras
  - Mediante el operador `instanceof`
    - Comprueba si tipo dinámico del objeto es igual que, o un subtipo de, el tipo dado (clase o interfaz)
    - Puede usarse en expresiones tales como: `obj instanceof ClaseDada`
  - Mediante el método `getClass()` heredado de la clase `Object`
    - Devuelve un objeto de la clase `Class` (ver el API de Reflection) que representa el tipo dinámico del objeto
    - Puede usarse en la expression `o.getClass() == ClaseDada.class`, donde `ClaseDada.class` devuelve el objeto de la clase `Class` que representa a la clase `ClaseDada` (la instancia de la metaclase `Class` que representa a `ClaseDada`)

Ejercicio: comparación de tipos

- Dada la jerarquía de clases siguiente:

```
public class A {…}; public interface D {…};
public class B extends A implements D {…}
public class C extends B {…}
```

y la asignación siguiente:

```
A a = new B();
```

¿Cuál de las siguientes expresiones tiene el valor `true`?

```
a instanceof A
a instanceof B
a instanceof C
a instanceof D
a.getClass() == A.class
a.getClass() == B.class
a.getClass() == D.class
```

Ejemplo 15: polimorfismo y downcasting

```
public class Persona {
    protected int NIF;
    public Persona(int unNIF){ NIF = unNIF; }
    public int getNIF(){ return NIF; }
}

public class Alumno extends Person {
    protected String estudios;

    public Alumno(int NIF, String titulacion) {
        super(NIF);
        estudios = titulacion;
    }
    public String getEstudios(){ return estudios; }
}
```


Ejemplo 15: polimorfismo y downcasting

```
// Un alumno de la educación a distancia
public class AlumnoDistante extends Alumno {

    private String codigoPais;

    public AlumnoDistante(int NIF, String titula, String pais){
        super(NIF, titula);
        codigoPais = pais;
    }
    public String getCodigoPais(){
        return codigoPais;
    }
}
```

Ejemplo 15: polimorfismo y downcasting

```
public class Main {

  public static void main(String[] args) {

    AlumnoDistante alumno =
         new AlumnoDistante(1, "Grado en Informática", "pt");

    int NIF = alumno.getNIF ();
    String estudios = alumno.getEstudios();
    String pais = alumno.getCodigoPais();
    System.out.println("NIF: " + NIF);
    System.out.println("Estudios: " + estudios);
    System.out.println("Código País: " + pais);
  }
}
```


Ejemplo 15: polimorfismo y downcasting

```
public class Main {

  public static void main(String[] args) {

    Alumno alumno =
         new AlumnoDistante(1, "Grado en Informática", "pt");

    int NIF = alumno.getNIF();
    String estudios = alumno.getEstudios();
    String pais = alumno.getCodigoPais(); // ¿correcto?
    System.out.println("NIF: " + NIF);
    System.out.println("Estudios: " + estudios);
    System.out.println("Código País: " + pais);
  }
}
```

Ejemplo 15: polimorfismo y downcasting

```
public class Main {

  public static void main(String[] args) {

    Alumno alumno =
         new AlumnoDistante(1, "Grado en Informática", "pt");

    int NIF = alumno.getNIF();
    String estudios = alumno.getEstudios();
    String pais = ((AlumnoDistante) alumno).getCodigoPais();
    System.out.println("NIF: " + NIF);
    System.out.println("Estudios: " + estudios);
    System.out.println("Código País: " + pais);
  }
}
```


Ejemplo 15: polimorfismo y downcasting

```
public class Main {

  public static void main(String[] args) {

    Persona alumno =
         new AlumnoDistante(1, "Grado en Informática", "pt");

    int NIF = alumno.getNIF ();
    String estudios = ((Alumno) alumno).getEstudios();
    String pais = ((AlumnoDistante) alumno).getCodigoPais();
    System.out.println("NIF: " + NIF);
    System.out.println("Estudios: " + estudios);
    System.out.println("Código País: " + pais);
  }
}
```

Ejemplo 15: polimorfismo y downcasting

```
public class Main {

  public static void main(String[] args) {

    Persona alumno =
         new AlumnoDistante(1, "Grado en Informática", "pt");
    AlumnoDistante alumnoDist = (AlumnoDistante) alumno
    int NIF = alumno.getNIF();
    String estudios = alumnoDist.getEstudios();
    String pais = alumnoDist.getCodigoPais();
    System.out.println("NIF: " + NIF);
    System.out.println("Estudios: " + estudios);
    System.out.println("Código País: " + pais);
  }
}
```


Ejemplo 16: Polimorfismo e interfaces en Java

```
interface ConductaAnimal {
    public void comer();
}

interface ConductaPerro extends ConductaAnimal {
    public void grunhir();
}

interface ConductaGato extends ConductaAnimal {
    public void ronronear();
}
```

Ejemplo 16: Polimorfismo e interfaces en Java

```
public class Perro implements ConductaPerro {
    public void comer() { ... }
    public void grunhir() { ... }
}

public class Gato implements ConductaGato {
    public void comer() { ... }
    public void grunhir() { ... }
}

public class Extraterrestre implements ConductaGato,
                                       ConductaPerro {
    public void comer() { ... }
    public void grunhir() { ... }
    public void ronronear() { ... }
}
```


Ejemplo 16: Polimorfismo e interfaces en Java

```
public class Test{
    public static void main (String args[]) {
        // ConductaAnimal es el tipo estático de los 4 objetos en granja
        ConductaAnimal granja[] = new ConductaAnimal[4];
        granja[0] = new Perro();
        granja[1] = new Gato();
        granja[2] = new Extraterrestre();
        granja[3] = new Extraterrestre();
        for (int i = 0; i<4; i++) {
            granja[i].comer();                    // vinculación dinámica
        }
        granja[2].grunhir();                      // violación de tipo
        ( (Perro) granja[0] ).grunhir();          // OK, cast al tipo clase
        ( (ConductaGato) farm[1] ).ronronear();   // OK, cast al tipo interfaz
        ( (ConductaPerro) farm[2] ).grunhir();    // OK, cast al tipo interfaz
        ( (Extraterrestre) farm[3] ).grunhir();   // OK, cast al tipo clase
    }
}
```

Polimorfismo en C++

- En C++, no todas las variables son polimórficas y no todos los métodos pueden ser sobrescritos (razón: la vinculación dinámica es costosa)

```
class A {
    public:
        virtual int f() { return 1; }       // puede ser rescrito
        int g() { return 2;}                // no puede ser rescrito
};
class B : public A {
    public:
        int f() { return 4; } // usado si tipo estático o dinámico es B
        int g() { return 8; } // usado solo si tipo estático es B
};
int main () {
    A a; B b;  // variables no polimórficas
    A *p = &b; // variable polimórfica (tipos: estático A, dinámico B)
    int c = a.f() + b.f() + p->g() + p->f(); // ¿cuál es el valor?
    return 0;
}
```


Downcasting en C++

```
class A {
    public:
        virtual void foo (); // Asegura que el compilador genera
                             // RTTI. RTTI solo es disponible para
        ...                  // clases polimórficas, es decir, clases
                             // clases con al menos un método virtual
};

class B : public A {
    public:
        void metodoEspecificoAB ();
        ...
};
```

Downcasting en C++

```
B *b = new B();  // Tipo estático y dinámico B
A *a1 = b;       // Tipo estático A, tipo dinámico B.
A *a2 = new A(); // Tipo estático y dinámico A.
B *ptr;

ptr = b;       // OK
ptr = a1;      // Error en ejecución (necesita un downcast)
ptr = dynamic_cast <B*>(a1); // OK (chequeo en ejecución)
ptr = dynamic_cast <B*>(a2); // No OK (chequeo en ejecución)
                             // ptr cogerá el valor NULL
                             // Programador puede hacer: if (!ptr)
ptr = (B*)a1; // Sin ningún chequeo de tipos, funcionará aquí
ptr = (B*)a2; // Sin ningún chequeo de tipos, error de ejecución
              // El cast de C está deprecado
```


Destructores y polimorfismo en C++

- No incluir un destructor virtual en una clase base puede ser peligroso
  - Destructores C++ deberían ser `public` y `virtual`, o `protected` y no `virtual`

```
class CEntity { };

class CBadGuy : public CEntity {
        public:
                CBadGuy () { _weapon = new CWeapon; } // constructor
                ~CBadGuy() { delete _weapon; }        // destructor
        private:
                CWeapon* _weapon;
};

int main () {
        CEntity* e = new CBadGuy;
        delete e; // fuga de memoria
        return 0;
}
```

Universidad Complutense de Madrid
Facultad de Informática
Grado en Ingeniería Informática, Grupo B

Tecnología de la programación

Apéndice

Ejemplo1: fragilidad de la clase base

```
public class CountingSet extends Set {
  private int count;

  public void add(Object o) {
    super.add(o);
    count++;
  }
  public void addAll(Collection c) {
    super.addAll(c);
    count += c.size();
  }
  public int size() {
    return count;
  }
}
```

- ¿Qué pasa con `count` si cambiamos el método `addAll()` de la clase `Set` para que se implemente mediante `c.size()` llamadas a `add()`?

Ejemplo2: fragilidad de la clase base

```
class Super {
    private int counter = 0;
    public void inc1(){ counter++; }
    public void inc2(){ counter++; }
}

class Sub extends Super {
    @Override
    public void inc2(){ inc1(); }
}
```

¿Qué pasará si cambiamos el método `inc1()` de la clase `Super` como sigue?

```
    void inc1() { inc2(); }
```

Piense en una llamada al método `inc2()` de una instancia de la clase `Sub`

"*Diseña y documenta para la herencia o bien prohíbela*" Joshua Bloch

Notación, Covariante y Contravariante

- Denotemos `Cov((U, V),(W,X))`
  - cuando la relación entre los tipos de `U` y `V` es covariante con respecto a la relación entre los tipos `W` y `X`
  - Esto es, `U` será subtipo de `V` siempre que `W` sea subtipo de `X`
- Denotemos `Contrav((U, V),(W,X))`
  - cuando la relación entre los tipos de `U` y `V` es contravariante con respecto a la relación entre los tipos `W` y `X`
  - Esto es, `V` será subtipo de `U` siempre que `W` sea subtipo de `X`

Relación entre Signaturas en Tipos y Subtipos

- Supongamos que
  - La clase `B` tiene un atributo `x` de tipo Tx_B
  - La clase `A` tiene un atributo `x` de tipo Tx_A

  ¿es cierta la afirmación siguiente (coherente con la teoría de tipos)?

  `B.x` oculta `A.x` $\Leftrightarrow$ `Cov((`Tx_B`, `Tx_A`),(B,A))`
- Supongamos que
  - La clase `B` tiene un método `m:` $P_{B1} \times \ldots \times P_{Bn} \rightarrow R_B$
  - La clase `A` tiene un método `m:` $P_{A1} \times \ldots \times P_{An} \rightarrow R_A$

  ¿es cierta la afirmación siguiente (coherente con la teoría de tipos)?

  `B.m` sobrescribe `A.m` $\Leftrightarrow$ `Cov((`R_B`, `R_A`),(B,A))`
  y $\forall i$ `Contrav((`P_{Bi}`,`P_{Ai}`),(B,A))`
- En la mayoría de los lenguajes OO, no (ver transparencias anteriores)