v 1.9

Tema 3
Herencia y polimorfismo con Java

Programación Orientada a Objetos

José Francisco Vélez Serrano

Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Índice

- La herencia de clases
- Polimorfismo dinámico
- Clases abstractas e interfaces

La herencia de clases

Java permite la herencia simple de clases, que básicamente permite que una clase copie de otra su interfaz, su comportamiento y que añada nuevo comportamiento en forma de código.

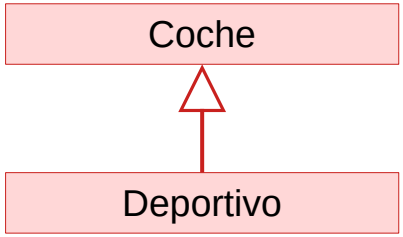
Definición de herencia en Java

La herencia de clases posibilita continuar desarrollando una clase base (padre o super clase) en una nueva clase derivada (clase hija o sub-clase).

En Java se usa la palabra reservada extends para indicar que una clase deriva de otra.

```
[Modificador de clase] class <identificador> [parámetros] [herencia] [excepciones] {
    [método|propiedad|clase|bloque inicialización]*
}

[herencia] ::= [extends <super_clase>]
```

```
class Coche {
}

class Deportivo extends Coche {
}
```



¿Qué permite la herencia?

En general, en una clase derivada se puede:

- Definir nuevos miembros que se añaden a los ya existentes en las clases base.
- Invocar miembros de las clases base.

Estas posibilidades están limitadas por algunos modificadores usados en las clases base. La principal limitación es que miembros privados en la clase base no son visibles en las clases derivadas.

La herencia se usa para añadir funcionalidad

La herencia suele usarse para añadir funcionalidad a una clase de la que se hereda.

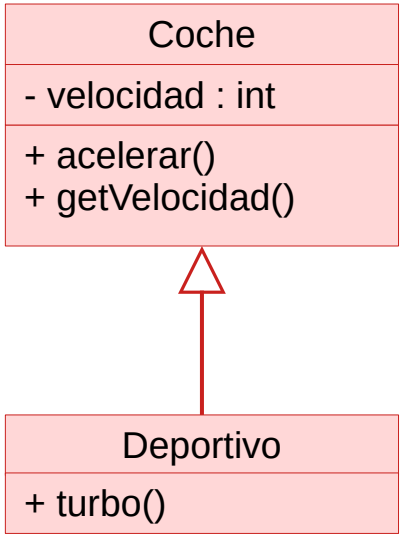
Por ejemplo: se hereda de una clase Pila y se añade un método toArray que devuelve los elementos de la pila en un array.

En este caso la herencia evita duplicar código (copiar y pegar) para crear una clase muy similar a otra que ya tenemos.

También evita modificar clases que funcionan correctamente, y que ya son usadas por otros clientes, cuando se les quiere añadir nuevas funcionalidades.

La herencia de clases

"Herencia" vs "copiar y pegar código"




```java
//Clase base, padre o super clase
public class Coche {
  private int velocidad;

  public void getVelocidad() {
     return velocidad;
  }

  public void acelerar() {
     velocidad++;
  }
}
```

```java
//Clase derivada o clase hija
public class Deportivo extends Coche {

  public void turbo() {
    for (int c = 0; c < 10; c++)
      acelerar();
  }
}
```

≈

```java
public class Deportivo {
  private int velocidad;

  public void getVelocidad() {
     return velocidad;
  }

  public void acelerar() {
     velocidad++;
  }

  public void turbo() {
    for (int c = 0; c < 10; c++)
      acelerar();
  }
}
```

Se puede invocar en la clase hija

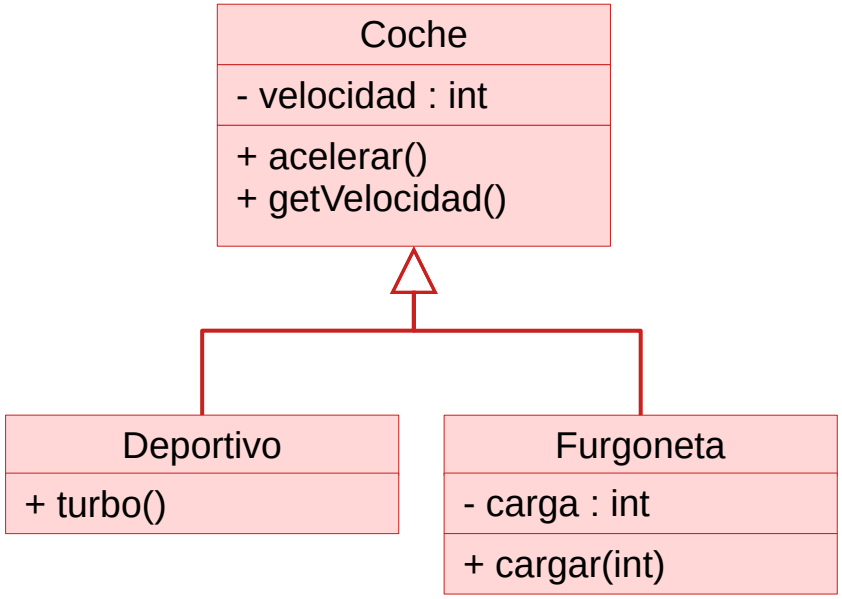
No se puede invocar en la hija

Ejemplo de herencia

```java
//Clase base, padre o super clase
public class Coche {
  private int velocidad;

  public void getVelocidad() {
     return velocidad;
  }

  public void acelerar() {
     velocidad++;
  }
}
```

```java
//Clase derivada o clase hija
public class Deportivo extends Coche {

  public void turbo() {
    for (int c = 0; c < 10; c++)
      acelerar();
  }
}
```

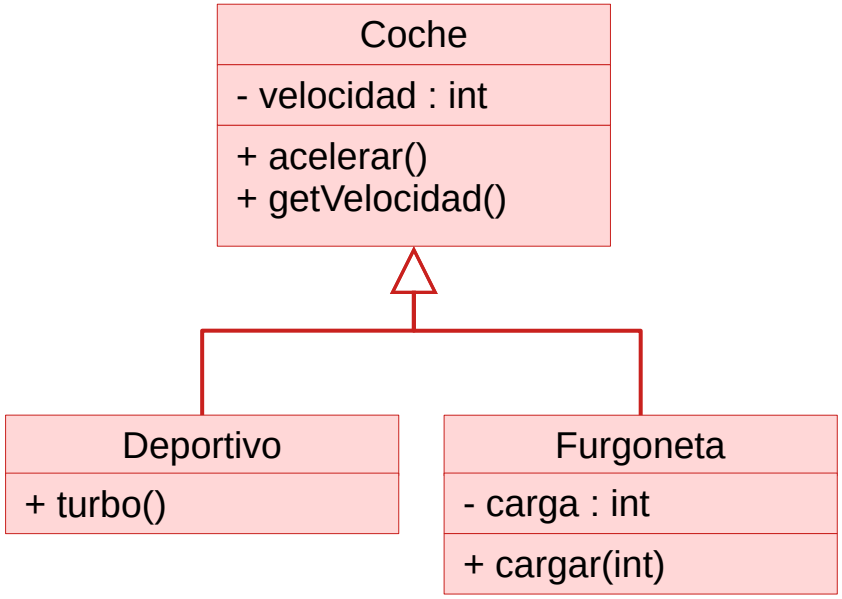
```java
//Clase derivada o clase hija
public class Furgoneta extends Coche {
  private int carga;

  public void cargar(int carga) {
    this.carga = carga;
  }
}
```



Coche define los métodos acelerar y getVelocidad, por tanto Deportivo y Furgoneta los tienen. Deportivo añade turbo, mientras que Furgoneta añade cargar.

Cada objeto tiene sus propias propiedades

```
public static void main() {
  Deportivo d = new Deportivo();
  Furgoneta f = new Furgoneta();

  d.acelerar();
  f.acelerar()

  System.out.println(d.getVelocidad());
  System.out.println(f.getVelocidad());
}
```



Cada objeto de la clase Coche tendrá su propia propiedad velocidad con un valor independiente.

Java tiene una única jerarquía de clases

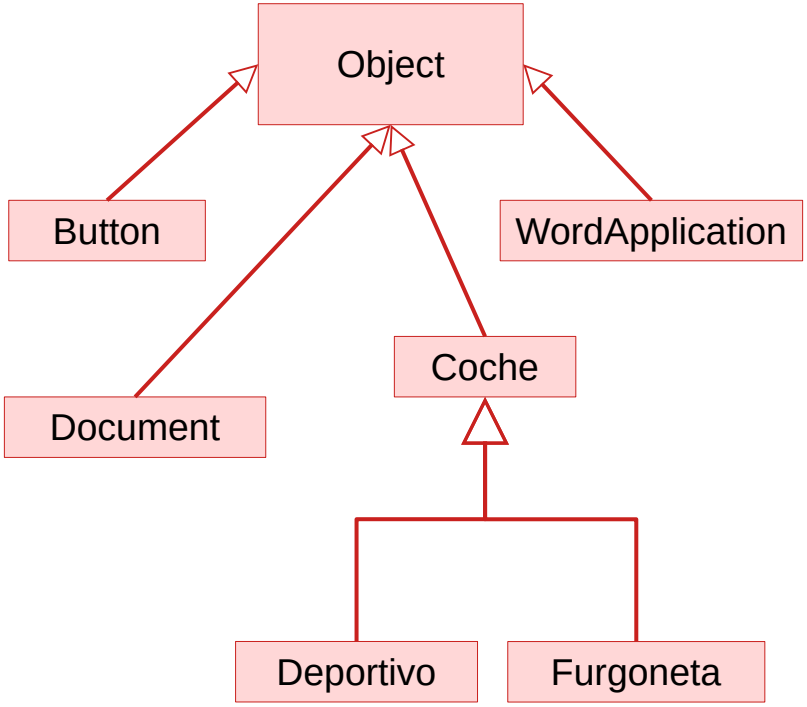
En Java cada clase deriva de otra clase.

En Java solo hay un árbol de herencia.

La clase Object está en la raíz del árbol de herencia de Java.

Cuando no se declara de qué clase deriva una clase, Java la hace derivar de la clase Object.

En Java una clase no puede derivar de dos o más clases.

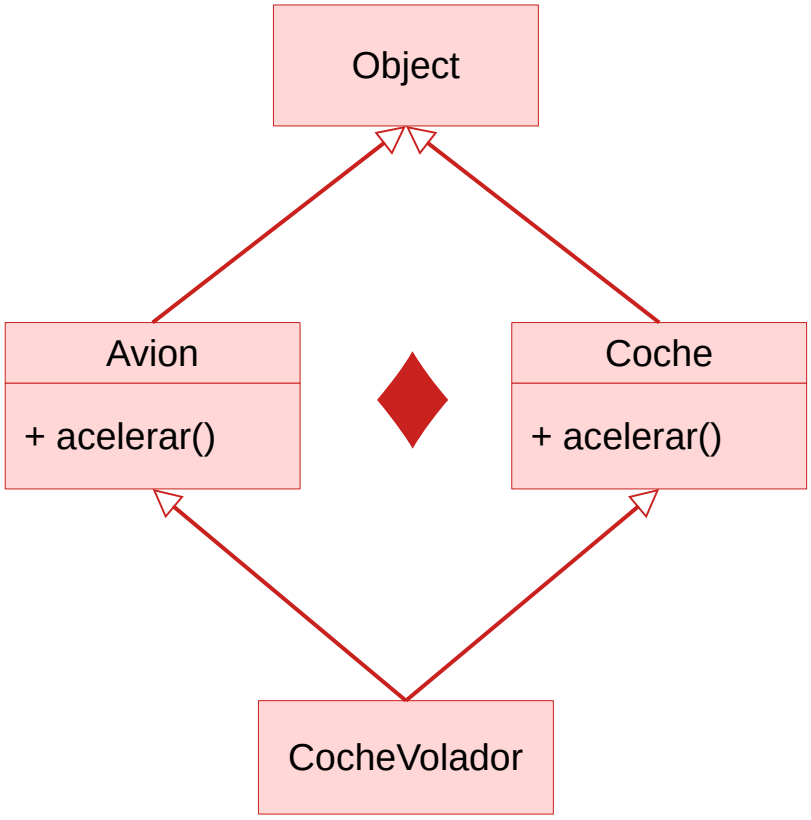


Herencia múltiple y el problema del diamante

Si una clase hereda de dos o más clases aparece al problema del diamante.

Dicho problema se deriva de la ambigüedad resultante al invocar un método definido en dos puntos de una jerarquía. ¿Qué método se ejecutaría cuando se le pide acelerar a un CocheVolador? ¿Quién invoca antes al constructor de Object?

Java no permite la herencia múltiple de clases, evitando este problema. Otros lenguajes (C++, Python...) sí la permiten, y definen complejas políticas para resolver la ambigüedad.



Modificador de control de acceso protected

Un miembro protected (# en UML) es accesible desde:

- la propia clase
- clases que hereden de ella
- clases definidas en el mismo paquete.

```
public class Coche {
  protected int velocidad;

  public void acelerar() {
    velocidad++;
  }
}

public class Deportivo extends Coche
{

  public void turbo() {
    for (int c = 0; c < 10; c++)
      acelerar();
  }

  public void frenar() {
    velocidad = 0;
  }
}
```

Resumen de control de acceso

	Visibilidad			
Modificador	Clase	Paquete	Subclase	Cualquiera
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
sin modificador	Sí	Sí	No	No
private	Sí	No	No	No

Sobrecarga dinámica de métodos y super

Cuando en una clase derivada se redefine un método ya definido en una clase base se dice que se está sobreescribiendo el método (o sobrecargandolo).

La declaración del método que sobrecarga debe ser igual a la del método sobrecargado añadiendo el tag @override.

Desde una clase derivada siempre se puede invocar al método de la clase base usando la propiedad privada super.

Los clientes de una clase derivada no ven los miembros sobrescritos de la clase base, solo ven la versión proporcionada por la clase derivada.

```
public class Coche {
  private int velocidad;

  public void acelerar() {
     velocidad++;
  }

  public int dameVelocidad() {
     return velocidad;
  }
}

public class Deportivo extends Coche {

  @override
  public void acelerar() {
    super.acelerar();
    super.acelerar();
    turbo = true;
  }
}
...
metodoExterno() {
  Deportivo f = new Deportivo();
  f.acelerar();
  System.out.println(f.dameVelocidad());
}
```

2

Métodos final

Los métodos definidos como final no pueden cambiar su implementación en clases derivadas. No pueden ser sobrecargados.

El uso de final tiene dos objetivos:

- Fijar el diseño, pues fijar razones de diseño al impedir cambios.
- Aumentar la eficiencia, pues permiten ligadura estática de las funciones lo que redunda en mayor velocidad.

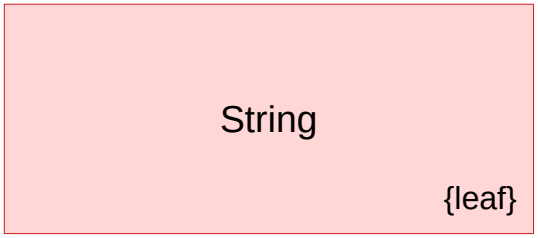
```
public class Coche {
  private int velocidad;

  public void acelerar() {
     velocidad++;
  }

  final public int dameVelocidad() {
     return velocidad;
  }
}

public class Deportivo extends Coche {
  boolean turbo = false;

  @override
  public void acelerar() {
    super.acelerar();
    super.acelerar();
    turbo = true;
  }
}
```

Clases finales

De una clase final no se puede heredar para crear una nueva clase.

Se suele usar para reforzar decisiones de diseño, como por ejemplo impedir que se derive de una clase inmutable para hacerla mutable.

```
final class String {
     char      charAt(int index){...
     int       compareTo(String anotherString){...
     String    concat(String str){...
     boolean   contains(CharSequence s){...
     boolean   equals(Object anObject){...
     byte[]    getBytes(){...
     int       indexOf(int ch){...
     int       lastIndexOf(int ch){...
     int       length(){...
     boolean   matches(String regex){...
     String    replace(char oldChar, char newChar){...
     String    substring(int beginIndex, int endIndex){...
     String    toLowerCase(){...
     String    trim(){...
     static    String    valueOf(double d){...
     ...
}
```



La herencia y los constructores

En Java, una clase siempre debe declarar explictamente los constructores que desea.

Los constructores de una clase padre no son visibles para los clientes de una clase hija.

Desde el constructor de una clase derivada se puede llamar al constructor de la superclase utilizando super.

Si no se llama al constructor de la clase padre, al definir un constructor en una clase derivada, Java añade una llamada al constructor por defecto de la superclase (si la superclase no tuviese constructor por defecto no compilaría).

```
public class Coche {
  private int velocidad;

  public Coche(int velocidad) {
    this.velocidad = velocidad;
  }

  public void acelerar() {
     velocidad++;
  }

  final public int dameVelocidad() {
     return velocidad;
  }
}

public class Deportivo extends Coche {
  private boolean turbo;

  public Deportivo() {
    super(120);
    turbo = true;
  }
}
```

Ejemplo de herencia

```
class Coche {
  public int velocidad; //Mal estilo
  private int ruedas = 4;

  //El método private no se hereda
  private void quitarRueda() {
    ruedas--;
  }

  //El método protected se hereda
  protected void frenar() {
    velocidad = 0;
  }

  //Un método friend se hereda si
  //el derivado está en el paquete
  void acelerar() {
    velocidad++;
  }

  //El método public se hereda
  public void abrirPuerta() {
    personas++;
  }
}
```

```
//La clase Deportivo deriva de Coche
class Deportivo extends Coche {
  private int inyectores = 12;

  public void turbo() {
    ruedas++;  //Error, no visible
    quitarRueda(); //Error, no visible
    velocidad--;  //Mal estilo
    frenar();
    abrirPuerta();
    inyectores++;
    acelerar();
  }
}

class Prueba {
  public static void main(String [] arg){
    Deportivo d = new Deportivo();
    d.inyectores++; //Error, no visible
    d.quitarRueda(); //Error, no visible
    d.velocidad = 5; //Mal estilo
    d.abrirPuerta();
    d.turbo();
  }
}
```

Polimorfismo dinámico

El polimorfismo es una potente herramienta que permite utilizar jerarquías de clases sin preocuparse de los detalles específicos de cada clase.

Referencias de un tipo, objeto de otro

En Java un objeto de un tipo T puede estar referenciado por una referencia del tipo T o por una referencia de cualquier tipo padre de T.

```
class A {
}

class B extends A {
}

...

metodoExterno() {
  B b = new B();
  A a;
  a = b;
}
```

```
class Coche {
}

class Deportivo extends Coche {
}

...

metodoExterno() {
  Deportivo d = new Deportivo();
  Coche c;
  c = d;
}
```

El casting de clases

Una referencia a un tipo base solo puede asignar a un tipo derivado si se hace casting y el tipo coincide.

Se produce un error en el programa si un casting no tiene éxito.

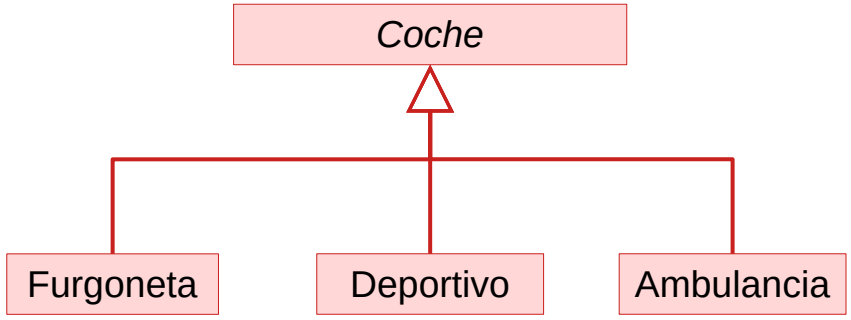
Se puede comprobar el tipo de un objeto mediante la palabra reservada instanceof. Su uso evita los errores por casting.

```
class Coche {
}

class Deportivo extends Coche {
}

...

metodoExterno() {
  Deportivo f = new Deportivo();
  Coche c = f;
  if (c instanceof Deportivo)
    Deportivo f2 = (Deportivo) c;
}
```

El nombre de polimorfismo

Una cosa es polimorfa si tiene múltiples formas.

En el ejemplo de la izquierda, la referencia c a Coche es polimorfa porque en ejecución (dinámicamente) puede tomar la forma de un Furgoneta, una Ambulacia, un Deportivo...




```
class Coche {
}

class Ambulancia extends Coche {
}

class Furgoneta extends Coche {
}

class Deportivo extends Coche {
}

public class EjemploPolimorfismo{

  public static Coche dameCoche() {
    final double azar = Math.random();

    if (azar > 0.75)
      return new Ambulacia();
    else if (azar > 0.5)
      return new Furgoneta();
    else
      return new Deportivo();
  }

  public static void main(String [] args) {
    Coche c = dameCoche();
    //¿De qué tipo es lo que apunta c?
  }
}
```

Polimorfismo dinámico

¿Qué es el polimorfismo?

El polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.

Gracias a la posibilidad de usar una referencia de un tipo base para apuntar a tipos derivados se consigue el polimorfismo dinámico en Java.

```
class A {
  public void pintaNombre() {
    System.out.println("A")
  }
}

class B extends A {
  public void pintaNombre() {
    System.out.println("B")
  }
}

class C extends A {
  public void pintaNombre() {
    System.out.println("C")
  }
}
```

```
A a;

if (Math.random() > 0.5)
     a = new B();
else
     a = new C();

a.pintaNombre(); //¿Qué método se invocará?
```

Polimorfismo dinámico

Ejemplo de uso de polimorfísmo dinámico

```
public class Coche {
  protected int velocidad;

  public void acelerar() {
    System.out.println("Puf, puf...");
    velocidad += 1;
  }
}
```

```
public class Ambulacia extends Coche {
  public void acelerar() {
    System.out.println("Ninoninonino...");
    velocidad += 10;
  }
}
```

```
public class Deportivo extends Coche {
  public void acelerar() {
    System.out.println("Fiuuuu");
    velocidad += 20;
  }
}
```

```
public class Furgoneta extends Coche {
  private boolean estaCargado = false;

  public Furgoneta(boolean cargado) {
    this.cargado = cargado;
  }

  public void acelerar() {
    if (estaCargado)
      System.out.println("Bruuumm");
    super.acelerar();
  }
}
```

```
public class EjemploPolimorfismo{

  public static Coche dameCoche() {
    final double azar = Math.random();

    if (azar > 0.75)
      return new Ambulacia();
    else if (azar > 0.5)
      return new Furgoneta(true);
    else
      return new Deportivo();
  }

  public static void main(String [] args) {
    Coche c = dameCoche();
    c.acelerar();
  }
}
```

La llamada a c.acelerar() es idéntica para todas las clases derivadas de Coche.

Solo en ejecución se puede saber cuál de las formas se ejecutará.

Polimorfismo dinámico

Otro ejemplo de uso de polimorfismo

```
class Coche {
  private int velocidad;
  public void acelerar() {
    velocidad++;
  }
  public void frenar() {
    velocidad--;
  }
  public int dameVelocidad() {
    return velocidad;
  }
}

class Ambulacia extends Coche {
  public void acelerar() {
    System.out.println("Ninoninonino...");
    velocidad += 10;
  }
}

class Deportivo extends Coche {
  public void acelerar() {
    for (int c = 0; c < 10; c++)
      super.acelear();
  }
}
```

```
class ZonaDeFrenado {

  public detener(Coche c) {

    while (c.dameVelocidad() > 0)
      c.frenar();
  }
}
```

Puedo llamar al método detener de ZonaDeFrenado pasándole cualquier tipo de Coche.

Polimorfismo dinámico

La herencia se usa para particularizar

La herencia suele usarse para particularizar la funcionalidad de una clase de la que se hereda.

Por ejemplo: Se hereda de una clase Text y se crear la clase CorrectText que sobreescribe el método insertWord para evitar que se inserten palabras incorrectas. Esto lo hace llamando al insertWord del padre solo cuando la palabra insertada está en un diccionario.

En este caso , además de evitar copiar y pegar código al crear clases similares y evitar tocar una clase que funciona, gracias al polimorfismo se posibilita que un cliente que use objetos de tipo Text use objetos de la clase CorrectText sin saberlo.

Polimorfismo dinámico

La herencia crea familias de clases

La herencia se usa para crear familias de clases que hacen cosas parecidas.

Por ejemplo: se hereda de AlgoritmoDeOrdenación para crear las clases QuickSort, MergeSort...

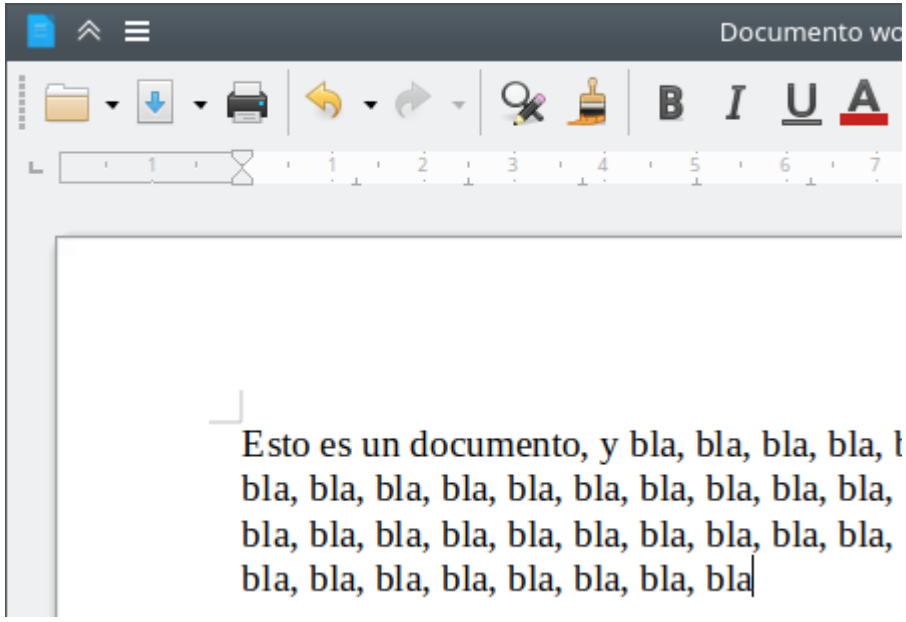
La herencia permitiría que se compartan métodos comunes entre las diferentes clases que heredan de la clase padre sin duplicar código. Así, en el ejemplo, la clase padre podría tener métodos comunes para insertar elementos, intercambiarlos...

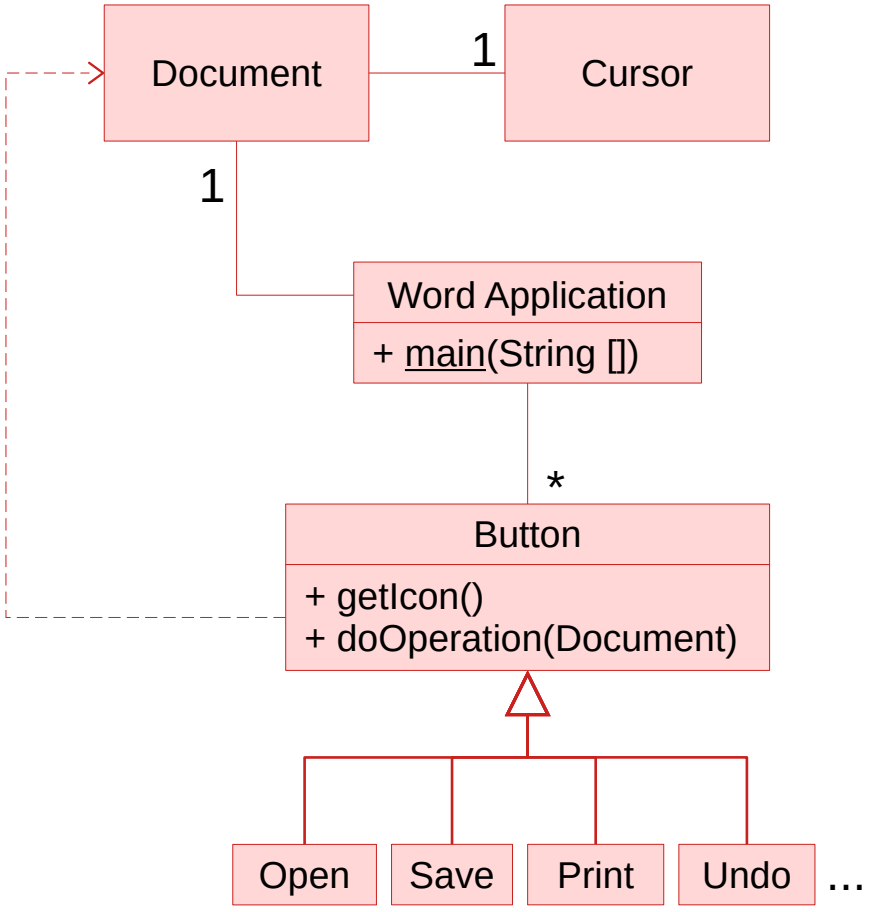
Además, de nuevo gracias al polimorfismo, facilita a los usuarios aprender a usar los diferentes algoritmos al dotarlos de la misma interfaz.

Polimorfismo dinámico

Uso de polimorfismo en un proyecto real

Gracias al polimorfismo, la aplicación no necesita conocer qué hace cada botón. Solo necesita invocar a su método doOperation cuando alguien lo pulse.





Consecuencias de la jerarquía única en Java

Se ha visto que:

- En Java un objeto de un tipo T puede estar referenciado por una referencia del tipo T o por una referencia de cualquier tipo padre de T.
- Todas las clases de Java derivan, en última instancia, de la clase Object.

Por tanto, cualquier objeto de Java puede ser apuntado por una referencia de tipo Object.

Entre otras cosas, esto permite crear estructuras de datos que admiten cualquier clase de objeto.

```
public class Coche {
  public void acelerar() {
    ...
  }
}

public class Pila{

  public void push(Object o) {
    ...
  }

  public Object pop() {
    ...
  }

  public boolean isEmpty() {
    ...
  }
}

metodoExterno() {
  Pila p = new Pila();
  Coche c = new Coche();
  p.push(c);
  ...
  Object o = p.pop();
  //o.acelerar() no compilaría

  Coche c2 = (Coche) o;
}
```

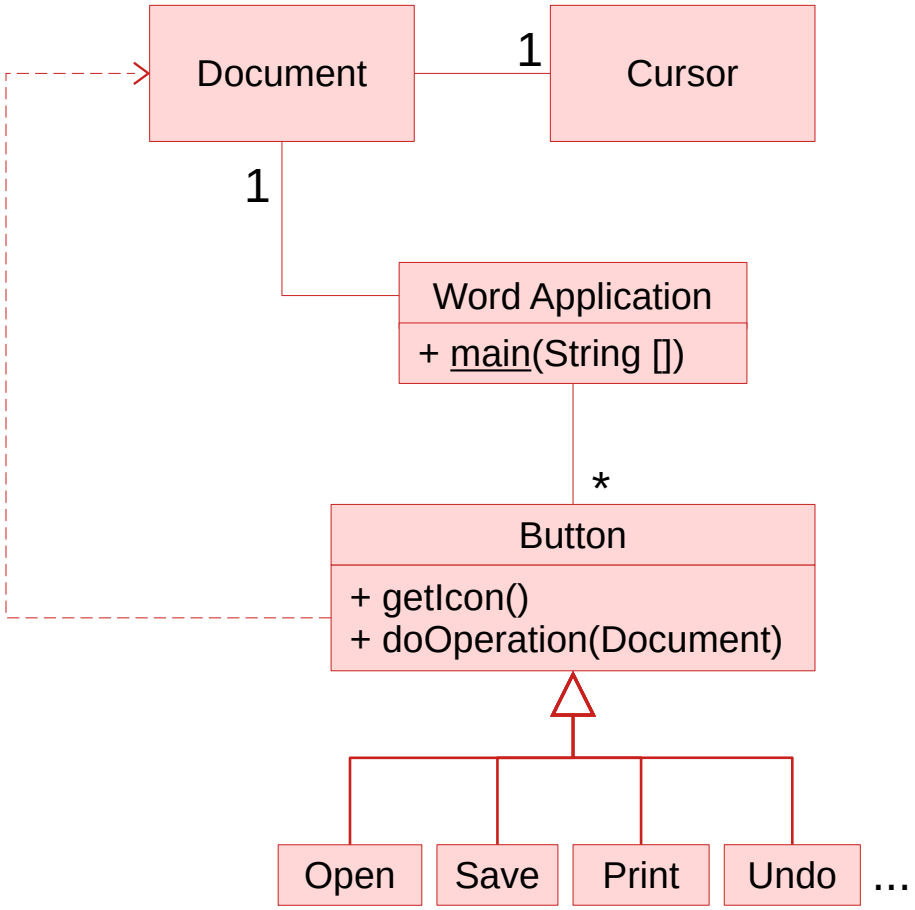
Clases abstractas e interfaces

En esta sección se analiza la separación entre implementación e interfaz que subyace al lenguaje Java.

Necesidad de métodos sin implementación

En ocasiones se desea que algunos métodos de las clases base de una jerarquía solo fijen las operaciones que se pueden realizar, sin implementar ningún comportamiento concreto.

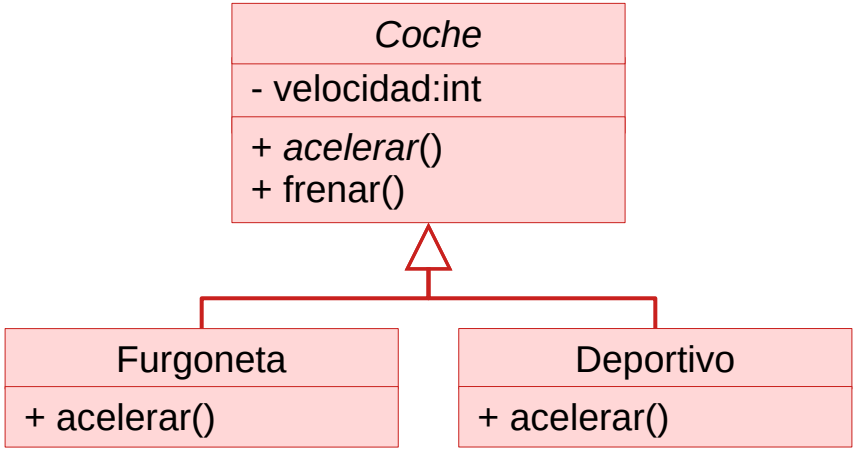
En el diagrama adjunto, puede que no tenga sentido que el método doOperation de la clase Button defina ningún comportamiento.



Las clases abstractas

- Las clases abstractas se usan para agrupar comportamiento común de las clases derivadas.
- En UML se representa lo abstracto poniéndolo en cursivas.
- El modificador abstract indica los métodos que no tienen implementación.
- Una clase con métodos abstractos se dice que es una clase abstracta.

```
abstract class Coche {
  private int velocidad;

  public abstract void
acelerar();

  public final void frenar() {
    velocidad = 0;
  }
}
```



Métodos y clases abstractas en Java

- No pueden crearse objetos de clases abstractas, porque tendría métodos sin implementación y su ejecución daría error.
- Si una clase deriva de clase abstracta debe implementar todos los métodos o indicar que, a su vez, es abstracta.
- Si todos los métodos de una clase abstracta son abstractos (no tienen implementación) se dice que la clase es abstracta pura.

Clases abstractas e interfaces

Ejemplo de uso de métodos abstractos

```java
public abstract class Coche {
  protected int velocidad;

  public abstract void acelerar();
}
```

```java
public class Ambulacia extends Coche {
  public void acelerar() {
    System.out.println("Ninoninonino...");
    velocidad += 10;
  }
}
```

```java
public class Deportivo extends Coche {
  public void acelerar() {
    System.out.println("Fiuuuu");
    velocidad += 20;
  }
}
```

```java
public class Furgoneta extends Coche {
  private boolean estaCargado = false;

  public Furgoneta(boolean cargado) {
    this.cargado = cargado;
  }

  public void acelerar() {
    if (estaCargado)
      System.out.println("Bruuumm");
    velocidad += 5;
  }
}
```

```java
public class EjemploPolimorfismo{

  public static Coche dameCoche() {
    final double azar = Math.random();

    if (azar > 0.75)
      return new Ambulacia();
    else if (azar > 0.5)
      return new Furgoneta(true);
    else
      return new Deportivo();
  }

  public static void main(String [] args) {
    Coche c = dameCoche();
    c.acelerar();
  }
}
```

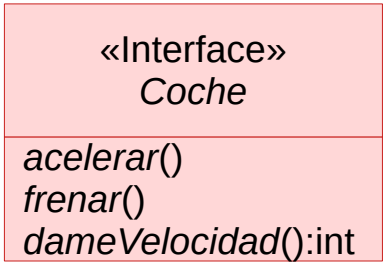
El método acelerar de la clase Coche se deja abstracto.

Las clases que derivan de ella se encargan de implementarlo.

Clases abstractas e interfaces

Interfaz

- Una interfaz es una clase abstracta pura y sin propiedades.
- Java permite definir interfaces mediante la palabra reservada interface.
- En una interfaz, todos los métodos son públicos por defecto.
- Para que una interfaz herede de otra interfaz se usa la palabra extends.
- Si una clase extiende una interfaz debe implementar todos los métodos o indicar que es abstracta.
- En UML las interfaces se distinguen por el estereotipo interface.

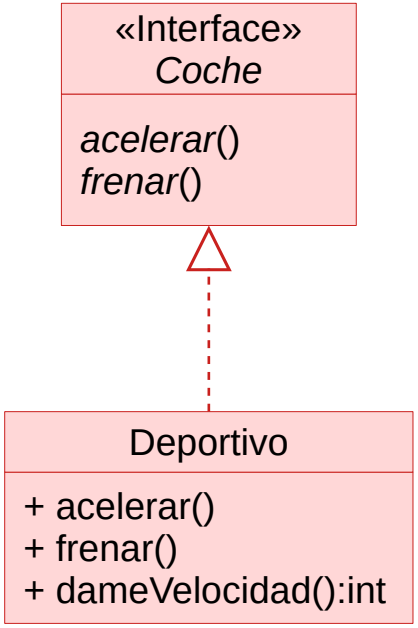



```
public interface Coche {
  void acelerar();

  void frenar();

  int dameVelocidad();
}
```

Implementación de interfaces

En Java se indica que una clase implementa una o varias interfaces utilizando la palabra reservada implements.

En UML la herencia de interfaz se representa con una flecha de punta hueca y línea punteada.



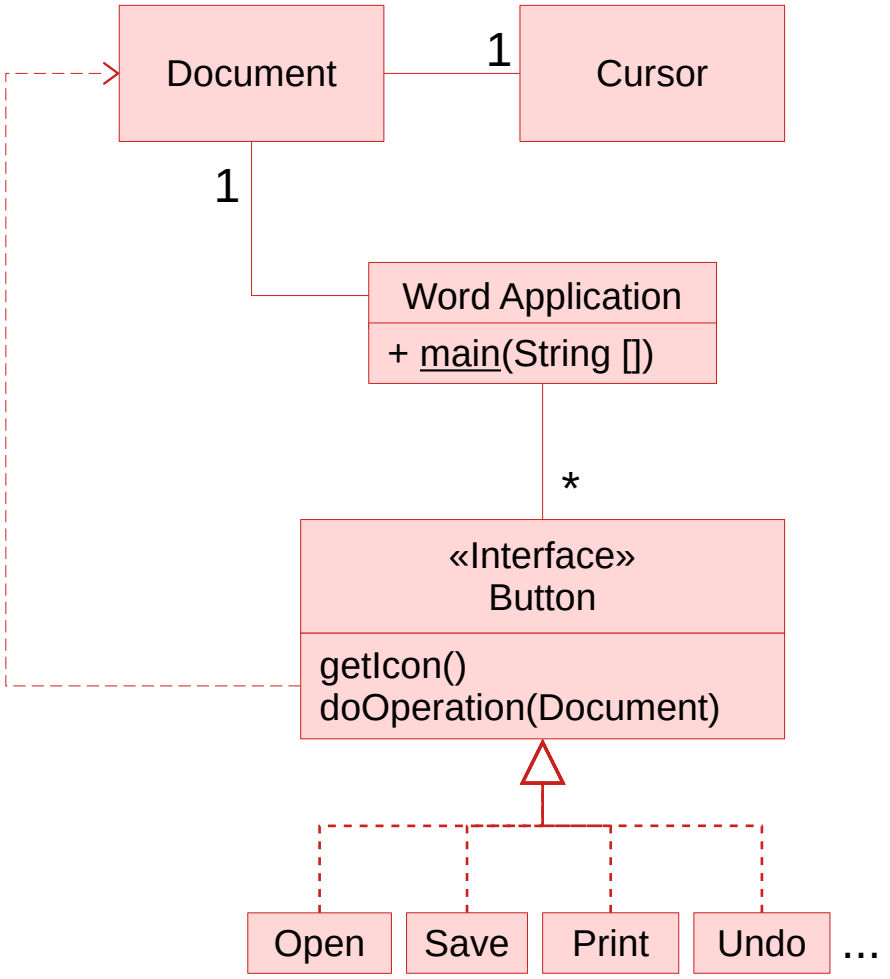

```
public interface Coche {
  void acelerar();
  void frenar();
}
```

```
public class Deportivo implements Coche{
  private int velocidad = 0;
  public void acelerar() {
    velocidad++;
  }
  public void frenar() {
    velocidad = 0;
  }
  public int dameVelocidad() {
    return velocidad;
  }
}
```

Ejemplo de uso de interfaces

Button pasa a ser una interfaz sin implementación.

Las clases que derivan de ella se encargan de implementar el comportamiento adecuado para getIcon y doOperation.



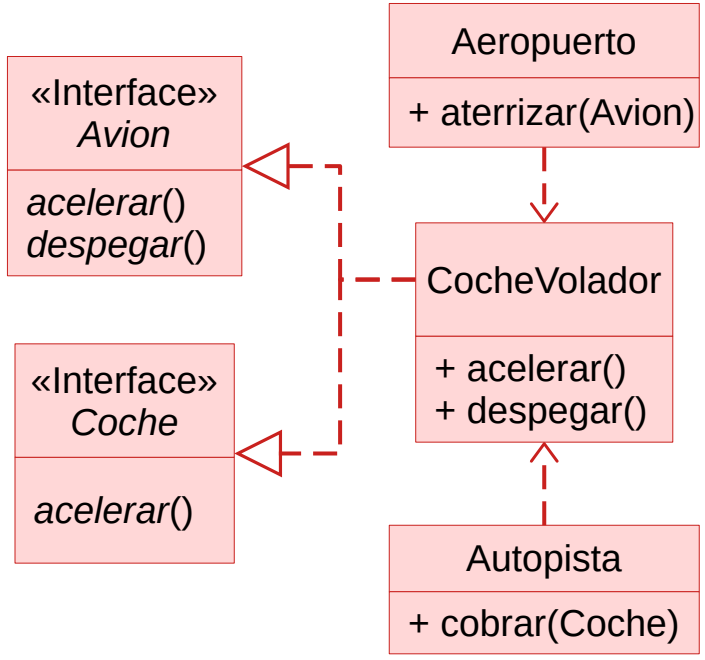
Herencia múltiple de interfaz

Java permite que una clase implemente más de una interfaz.

Que una clase implemente más de una interfaz permite a diferentes tipos de clientes utilizar la clase.

El problema del diamante no aparece porque solo hay una implementación.

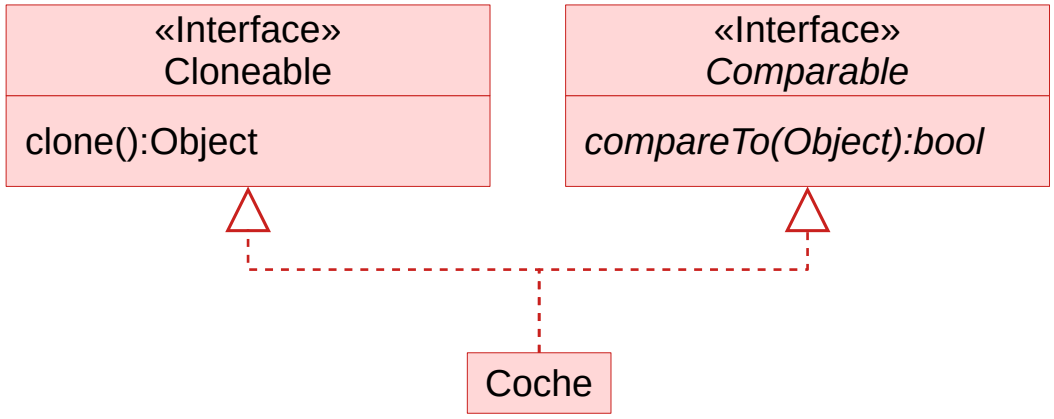
```
public class CocheVolador implements Coche, Avion{
  private int velocidad = 0;
  private int altura = 0;
  public void acelerar() {
    velocidad++;
  }
  public void despegar() {
    altura++;
  }
}
```



Uso real de herencia múltiple de interfaz

Imaginemos una interfaz Cloneable que define a los objetos que se pueden clonar y otra interfaz Comparable que define a los objetos que se pueden comparar.

Al programador de la clase Coche le interesa poder comparar un coche con otros y también le interesa poder sacar copias de un coche. Por ello decide que Coche implemente ambas interfaces.

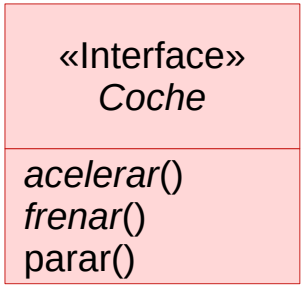


Métodos por defecto en las interfaces

Desde JDK 8 en las interfaces se permite implementar métodos usando el modificador default.

Esto se hizo para permitir añadir un método a una interfaz y que se mantenga la compatibilidad hacia atrás.

En el ejemplo de Coche se introduce el método parar() y las clases derivadas de Coche no tienen que ser modificadas gracias a que dicho método tiene una implementación por defecto.




```
public interface Coche {
  void acelerar();

  void frenar();

  int dameVelocidad();

  default void parar() {
    while (dameVelocidad > 0)
      frenar();
  }
}
```

Problema del diamante en las interfaces

La inclusión de implementación en la interfaces, unida a la posibilidad de herencia múltiple de interfaces conduce de nuevo al problema del diamante.

Java detecta el problema del diamante y fuerza al programador a que lo resuleva implementando el método conflictivo en la clase derivada.

```
interface Avion{
  default void acelerar() {
    System.out.println("Ziuuuu...");
  }
}

interface Coche{
  default void acelerar() {
    System.out.println("Puf, puf...");
  }
}

public class CocheVolador implements Coche, Avion{
```

types Coche and Avion are incompatible;
class CocheVolador inherits unrelated defaults for acelerar() from types Coche and Avion
---
(Alt-Enter shows hints)

Referencias

- "El lenguaje de programación Java", 3ª Edición, Arnold Gosling, Addison Wesley, 2001.
- "Piensa en Java". Eckel, 2ª Edición, Addison Wesley, 2002.
- "Introducción a la programación orientada a objetos con Java", C. Thomas Wu, Mc Graw Hill, 2001.