



# 4.

# Herencia

Jerarquías de clases, composición y abstracción

# Concepto de herencia

- La **herencia** es la segunda de las características principales de la Programación Orientada a Objetos que incorporan **todos** los lenguajes basados en este paradigma de programación
- Se puede definir la herencia como el mecanismo en virtud del cual una **clase derivada** reutiliza los atributos y los métodos que pertenecen a una **clase base** (o superior)
  - Típicamente, la clase derivada tiene los mismos atributos y métodos que la clase base, de manera que se “**copian**” **implícitamente**, es decir, **se heredan**, desde la clase base a la clase derivada
  - El objetivo de la herencia es la construcción de clases basadas en otras clases que **ya tienen implementada** la conducta deseada

# Concepto de herencia

- Los constructores de una clases no serán heredados por las clases derivadas, ya que se consideran que son **específicos** de la clase a la que pertenecen
- Una clase derivada se considera una **extensión** de la clase base en la que sus atributos y métodos son los siguientes
  - Los métodos y atributos que se heredan de la clase base
  - Los métodos y atributos que se definen **explícitamente** como propios de la clase base
- Entre una clase derivada (p.e., CartaDeMision) y una clase base (p.e., Carta) se dice que existe una **relación del tipo “es-un/a”** (p.e., una CartaDeMision **es una** Carta)

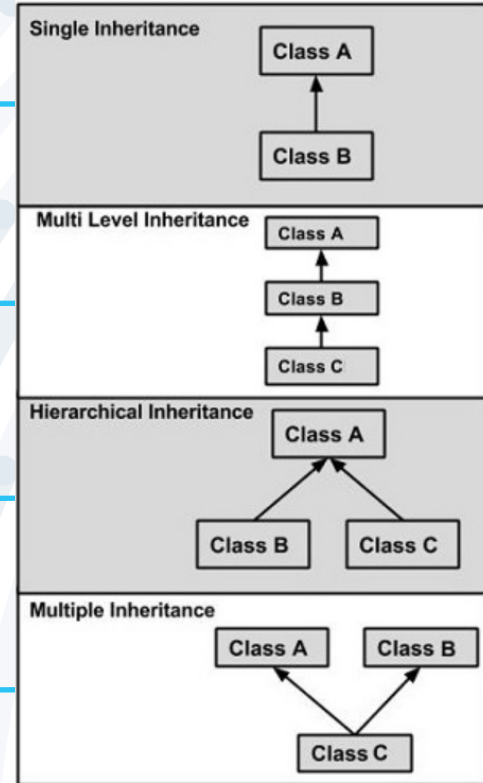
# Concepto de herencia

Una clase **B** hereda todos los atributos y métodos de una clase **A**. Es el tipo de herencia más **básico**, en la que están basados los demás tipos de herencia

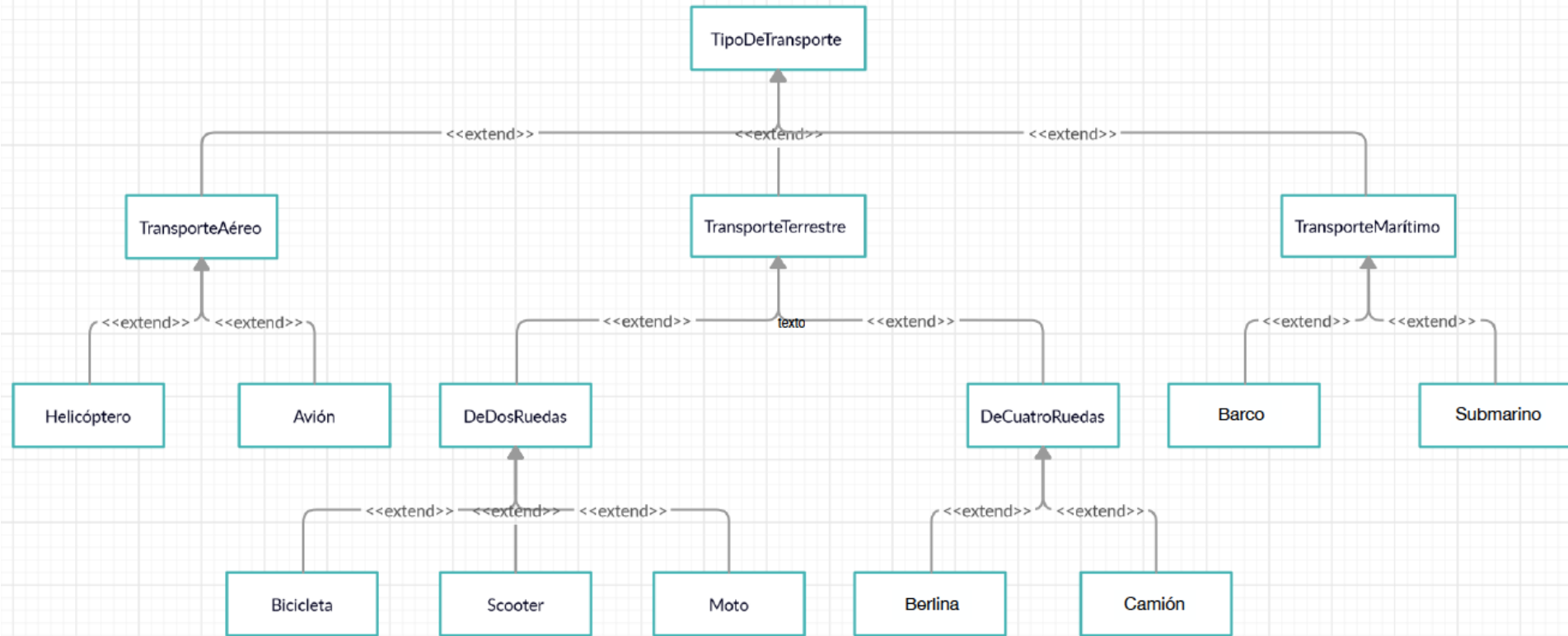
Una clase **C** hereda los atributos y métodos de una clase **B** que, a su vez, hereda de una clase **A**, de modo que la clase **C** hereda los métodos y atributos de la clase **A**

Las clases **B** y **C** heredan los atributos y los métodos de la clase **A**, de modo que **ambas clases se diferencian** en los atributos y métodos específicos de cada una de ellas

La clase **C** hereda los métodos y atributos de las clases **A** y **B**, de modo que es necesario definir **políticas de herencia** cuando **A** y **B** tienen métodos comunes



# Concepto de herencia



# Concepto de herencia

- El principal beneficio de la herencia es la **reutilización de código**, en la que un trozo de código que ha sido previamente desarrollado, depurado y validado en una clase se usa en otra clase **sin ser necesaria una modificación adicional**
- Adicionalmente, el uso de la herencia **tiene otros beneficios**
  - Simplifica el código, no siendo necesario implementar varias veces el mismo método en lo que serían las clases derivadas
  - Facilita el mantenimiento de los programas, ya que el cambio o reparación de los métodos tiene lugar en las clases base
  - Facilita la extensibilidad de los programas, ya que las nuevas clases (derivadas) se construyen a partir de otras clases (base)

?

# Concepto de herencia

- La herencia también tiene **algunas desventajas** que pueden desaconsejar su uso generalizado en el desarrollo de las clases del programa
  - Si el **nivel de jerarquización es muy profundo** el código es mucho más difícil de entender, tanto las relaciones entre las clases como los métodos que pertenecen a cada clase
  - Cambios en las clases base tienen un claro impacto en las clases derivadas, de manera que esos cambios **podrían ser inconsistentes** con el código necesario para las clases base
  - Para facilitar la herencia es posible que los atributos deban tener un modificador de acceso diferente de privado que, por tanto, sería **contrario al concepto de encapsulación**

# Concepto de herencia

## Class MetalScrollButton

java.lang.Object

java.awt.Component

java.awt.Container

javax.swing.JComponent

javax.swing.AbstractButton

javax.swing.JButton

javax.swing.plaf.basic.BasicArrowButton

javax.swing.plaf.metal.MetalScrollbar

### All Implemented Interfaces:

ImageObserver, ItemSelectable, MenuContainer, Serializable, Accessible, SwingConstants

Si se cambia algún método de la clase JComponent puede que ese cambio no sea consistente con las otras clases derivadas

### Method Summary

Methods	
Modifier and Type	Method and Description
int	<code>getWidth()</code>
Dimension	<code>getMaximumSize()</code> Returns the maximum size of the BasicArrowButton.
Dimension	<code>getMinimumSize()</code> Returns the minimum size of the BasicArrowButton.
Dimension	<code>getPreferredSize()</code> Returns the preferred size of the BasicArrowButton.
void	<code>paint(Graphics g)</code> Invoked by Swing to draw components.
void	<code>setFreeStanding(boolean freeStanding)</code>

Methods inherited from class `javax.swing.plaf.basic.BasicArrowButton`

getDirection, isFocusTraversable, paintTriangle, setDirection

Methods inherited from class `javax.swing.JButton`

```
getAccessibleContext, getUIClassID, isDefaultButton, isDefaultCapable, paramString, removeNotify, setDefaultCapable, updateUI
```

Methods inherited from class [javax.swing.AbstractButton](#)

Methods inherited from class `javax.swing.JComponent`

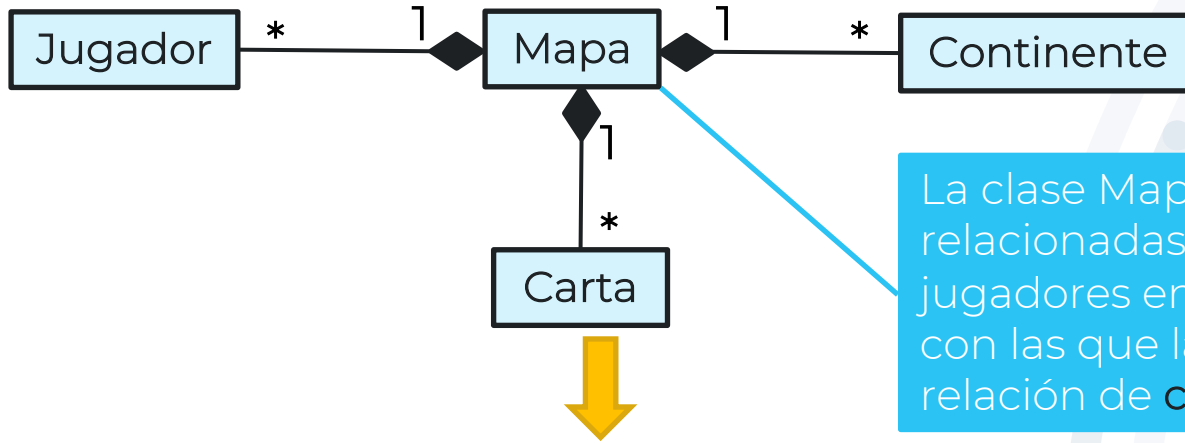
Methods inherited from class [java.awt.Container](#)

Methods inherited from class `java.awt.Component`



# Concepto de composición

- La **composición** es el mecanismo mediante el cual una clase contiene objetos de otras clases a los que se les **delega** ciertas operaciones para conseguir una funcionalidad dada



La clase Mapa **delega** las operaciones relacionadas con continentes, cartas y jugadores en las correspondientes clases con las que la clase Mapa establece una relación de **composición**

```
public class Mapa {  
    private ArrayList<Continente> continentes;  
    private ArrayList<Carta> cartas;  
    private ArrayList<Jugador> jugadores;
```

# Concepto de composición

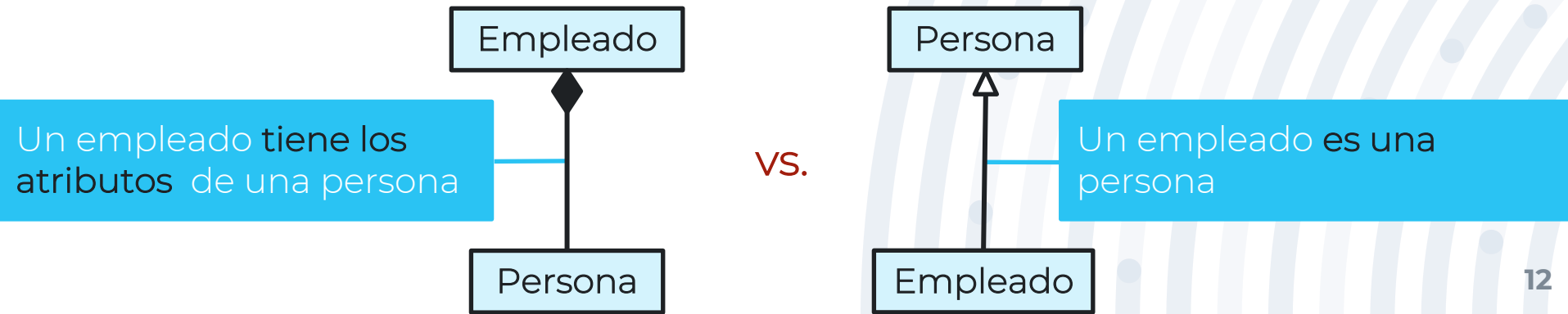
- El principal beneficio de la composición es que **reparte las responsabilidades entre los objetos**, es decir, cada objeto se encarga de realizar unas operaciones dadas sobre los atributos, que después podrán invocar los demás objetos
  - **Ejemplo:** la clase Continente es la **única responsable** de obtener la frontera de un continente, de modo que si se cambia el criterio sobre lo que se considera frontera, solo se cambiará dicha clase
- Adicionalmente, la composición tiene **otros beneficios**
  - Facilita el **mantenimiento** de los programas
  - Facilita la **extensibilidad** de los programas a través de la composición de clases

# Concepto de composición

- Otro beneficio importante es que en composición **las clases están más desacopladas entre sí**, de modo que un cambio en una de ellas tiene un impacto reducido en las otras
- El mecanismo de composición presenta también **desventajas**
  - Tiende a generar **mucho más código**, y de mayor complejidad, para soportar la misma funcionalidad que se consigue cuando se aplica el mecanismo de herencia
  - Se necesita más tiempo de desarrollo en comparación con el mecanismo de herencia, puesto que la construcción de nuevas clases no está basadas en otras clases y, por tanto, es necesario **implementar los métodos en todas las clases**

# Herencia vs Composición

- Se dice que la composición entre dos clases es una relación de tipo “**tiene-un/a**”
- Podría parecer fácil determinar **cuándo** una relación entre dos clases es de herencia o de composición, pero en muchos casos **no es tan sencillo**
  - Ejemplo: ¿**cuál** es la relación entre Hotel y Edificio?



# Herencia vs Composición

- Desde el punto de vista de la programación, **un empleado no debería de ser una persona**
  - Si un objeto empleado deja de ser empleado también dejará de ser persona, puesto que, de acuerdo con la relación de herencia, **un objeto empleado es una persona porque es un empleado**
  - La identidad de un empleado **no depende** de la de persona
- Es **más consistente** considerar que un empleado **tiene** los atributos de una persona
  - Las funcionalidades de las dos clases **no se “mezclan”**, sino que la clase Empleado **delega** las operaciones relacionadas con personas en la clase Persona

# Composición sobre herencia

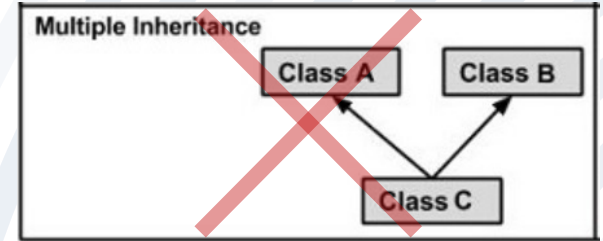
- ¿Cuándo no se debería de usar herencia (y sí composición)?
  - Si las clases no están relacionadas desde un punto de vista lógico [ → la herencia no tiene sentido ]
  - Si una clase base tiene **una sola clase derivada** [ → la herencia no tiene sentido ]
  - Si las clases derivadas heredan código que **no** necesitan [ → la herencia no tiene sentido ]
  - Si existe la posibilidad de que las clases base **cambien** [ → la herencia complica el desarrollo del programa ]
  - Si es necesario **sobrescribir** muchos métodos de las clases base [ → la herencia complica el desarrollo del programa ]

# Composición sobre herencia

Caso	Diseño basado en herencia	Diseño basado en composición
Inicio del desarrollo	Más rápido	Más lento
Diseño del software	Más sencillo	Más complejo
Efectos no deseados	Ocurren con más frecuencia, sobre todo en cuando se trata de jerarquías profundas	Se reducen y están más localizados, en los métodos delegados
Adaptación a cambios	Para jerarquías profundas y con múltiples sobreescritura es más complicado	Más sencillo de cambiar, puesto que solamente afectan a las clases delegadas
Validación	Es difícil de realizar, sobre todo en jerarquías profundas y con múltiples sobreescrituras	Más sencillo de validar al estar el código muy localizado en las clases delegadas
Extensibilidad	Es sencillo, aunque se complica en jerarquías profundas	Tiene lugar de forma más sencilla a través de la composición de clases

# Herencia en Java

- En la herencia de Java se hace énfasis en que una clase derivada **extiende** los atributos y métodos de una clase base, aunque impone una serie de **restricciones** entre los dos tipos de clases
  - No existe **herencia múltiple**, es decir, una clase derivada solo puede tener una única clase base de cual extenderá sus atributos y métodos
- Solamente se heredan los atributos y métodos de la clase base **que son públicos para la clase derivada**, es decir, se heredarán en función de los tipos de acceso de los métodos y atributos de la clase base





# Herencia en Java

```
1 package risk.componentes;
2
3 public class CartaDeEquipamiento {
4     protected String id;
5     protected String descripcion;
6     private String tipo;
7
8     public String getTipo( ) {
9         return this.tipo;
10    }
11
12    public void setTipo(String tipo) {
13        this.tipo= tipo;
14    }
15
16    public String getId( ) {
17        return this.id;
18    }
19
20    public void setId(String id) {
21        this.id= id;
22    }
23
24    public String getDescripcion( ) {
25        return this.descripcion;
26    }
27
28    public void setDescripcion(String descripcion) {
29        this.descripcion= descripcion;
30    }
31 }
```

Infanteria  
es una  
CartaDeEquipamiento

```
1 package risk.componentes;
2
3 public class Infanteria extends CartaDeEquipamiento {
4 }
```

Aunque no se hayan creado explícitamente ni atributos ni métodos en la clase Infanteria, en realidad esta clase **tiene** los siguientes métodos:

- public String getTipo()
- public void setTipo(String tipo)
- public String getId()
- public void setId(String id)
- public String getDescripcion()
- public void setDescripcion(String desc)

# Revisitando <tipo-acceso>

Tipo de acceso	Comportamiento
<code>private</code> (privado)	La clase derivada <b>nunca heredar�</b> los atributos y los m�todos, independientemente del paquete en el que est� la clase base en relaci�n a la clase derivada.
<code>public</code> (p�blico)	La clase derivada <b>siempre heredar�</b> los atributos y los m�todos, independientemente del paquete en el que est� la clase base en relaci�n a la clase derivada.
<code>�</code> (acceso a paquete)	La clase derivada <b>solamente heredar�</b> los atributos y los m�todos de la clase base si ambas clases se encuentran en el mismo paquetes.
<code>protected</code> (protegido)	La clase derivada <b>siempre heredar�</b> los atributos y los m�todos, independientemente del paquete en el que est� la clase base en relaci�n a la clase derivada.

# Revisitando <tipo-acceso>

```
1 package risk.componentes;
2
3 public class CartaDeEquipamiento {
4     protected String id;
5     protected String descripcion;
6     private String tipo;
7     private Jugador jugador;
8
9     public CartaDeEquipamiento(String id, String desc, String tipo)
10     {
11         this.identificador= id;
12         this.descripcion= desc;
13         this.tipo= tipo;
14         this.jugador= new Jugador ( );
15     }
16
17     public String getTipo( ) {
18         return tipo;
19     }
20
21     public void setTipo(String tipo) {
22         this.tipo= tipo;
23     }
24
25     public String getJugador( ) {
26         return
27     }
28
29     public void setJugador (String jugador) {
30         this.jugador= jugador;
31     }
32 }
```

Infanteria  
es una  
CartaDeEquipamiento

```
1 package risk.componentes;
2
3 public class Infanteria extends CartaDeEquipamiento {
4     public Infanteria(String id, String desc) {
5         this.id= id;
6         this.descripcion= descr;
7     }
8 }
```

Infanteria **no hereda** los atributos jugador y tipo de CartaDeEquipamiento, aunque sí hereda los setters y getters que acceden a esos atributos

Infanteria **hereda** atributos id y descripcion, así como los setters y getters de esos atributos

El constructor de CartaDeEquipamiento **no se hereda**, de manera que es necesario reservar memoria para los atributos heredados

# Revisitando <tipo-acceso>

- Como los atributos privados no se heredan, para favorecer la herencia **se podría pensar** en que los atributos fuesen o bien públicos o bien protegidos
- Aún con esa restricción los atributos **deben ser privados**, ya que se considera que la encapsulación es una característica que ofrece mayores beneficios que la herencia
  - Sin encapsulación el **desarrollo** de los programas se hace mucho más complejo y es mucho más difícil de **mantener y validar**
  - La **composición** no tiene sentido sin encapsulación
- Si los atributos son protegidos o públicos, la **encapsulación desaparece** (acceso público ) o **se debilita** (acceso protegido)

# Revisitando <tipo-acceso>

- **Pregunta:** si la clase derivada no hereda los atributos privados, pero sí hereda los métodos públicos de esa clase, ¿cómo es posible que la clase derivada tenga esos métodos públicos si no hereda los atributos **que son usados por esos métodos**?

```
public static void main(String[] args) {  
    Infanteria infChina= new Infanteria("Infantería&China", "Infantería en China");  
    System.out.println("Tipo: " + infChina.getTipo());  
}
```

¿A qué atributo accede el método `getTipo()`?

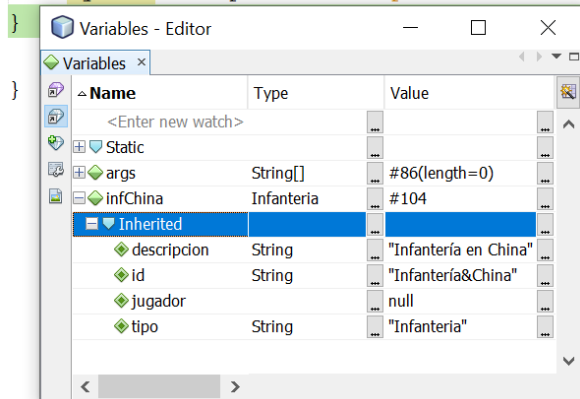
¿En qué clase se encuentra ese atributo?

¿Cómo se le asigna un **valor** a ese atributo?

# Revisitando <tipo-acceso>

- **Respuesta:** en realidad en la clase derivada hereda **todos los atributos**, independientemente de su tipo de acceso, es decir, se heredan los atributos públicos, privados, protegidos y de acceso a paquete

```
public static void main(String[] args) {  
    Infanteria infChina= new Infanteria("Infantería&China", "Infantería en China");  
    System.out.println("Tipo: " + infChina.getTipo());  
}
```



Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#86(length=0)
infChina	Infanteria	#104
Inherited		
descripcion	String	"Infantería en China"
id	String	"Infantería&China"
jugador		null
tipo	String	"Infanteria"

Todos los atributos de CartaDeEquipamiento son heredados por Infanteria, incluyendo los que tienen un tipo de acceso privado, es decir, jugador y tipo

El objeto infChina reserva memoria para **todos los atributos**, aunque sean de tipo privado

# Revisitando <tipo-acceso>

- **Respuesta:** en realidad en la clase derivada hereda **todos los atributos**, independientemente de su tipo de acceso, es decir, se heredan los atributos públicos, privados, protegidos y de acceso a paquete
  - El tipo de acceso especifica el **nivel de visibilidad/accesibilidad** que los métodos de la clase derivada tienen sobre los atributos y métodos de la clase base, es decir, si son privados, **se heredan pero no se puede acceder a ellos**
  - **El comportamiento es el habitual:** los objetos reservan memoria e inicializan los atributos de la clase derivada a la que pertenecen, y los métodos que se heredan de la clase base acceden a dichos atributos para cada objeto

# Revisitando <tipo-acceso>

- **Pregunta:** si la clase derivada no hereda los métodos privados de la clase base, pero sí hereda los métodos públicos, ¿cómo es posible que la clase derivada acceda a esos métodos públicos si no hereda los métodos privados **que son usados por dichos métodos?**
- **Respuesta:** ocurre lo mismo que en el caso de los atributos, es decir, la clase derivada **hereda todos los métodos** de la clase base, independientemente del tipo de acceso que tengan
  - El tipo de acceso **no evita o limita la herencia de los métodos**, sino que impone restricciones sobre la visibilidad que tienen los métodos de la derivada sobre los métodos que se heredan de la clase base



# Revisitando <tipo-acceso>

- Buenas prácticas de programación (XXII)

Los atributos deben de ser siempre **privados**, incluso aún cuando se esté aplicando herencia para favorecer la reutilización de código



- Buenas prácticas de programación (XXIII)

En una jerarquía de clases, **idealmente** los métodos deberían de estar implementados en la clase a la que pertenecen los atributos que se utilizan en su implementación, evitando hacerlo en las clases derivadas de dicha clase



# Constructores

- Los **constructores de la clase base no se heredan**, puesto que se considera que contienen código específico de la clase a la que pertenecen
  - Los constructores tienen que reservar memoria e inicializar los atributos de la clase **a la que pertenecen**, pero el constructor de la clase base no tiene acceso a los atributos de la clase derivada, con lo cual si se hereda el constructor de la clase base, **no podría ser invocado** para crear objetos de la clase derivada
- **Problema:** si la clase base hereda todos los atributos, pero no tiene acceso a los atributos privados de la clase base, ¿cómo se reserva memoria para estos atributos y se inicializan a los valores apropiados?

# Constructores

- Si el constructor de la clase base **no tiene argumentos**
  - Cuando se invoca al constructor de la clase derivada, **se invocará automáticamente** el constructor de la clase base
  - Si se implementa un constructor sin argumentos para la clase derivada, la clase base deberá tener **obligatoriamente** un constructor sin argumentos, en caso contrario, el compilador de Java **generará un error**

```
public class CartaDeEquipamiento {  
    ❶ public CartaDeEquipamiento() {  
        jugador= new Jugador();  
        System.out.println("Carta de equipamiento");  
    }  
}
```

```
public class Infanteria extends CartaDeEquipamiento {  
    ❷ public Infanteria() {  
        System.out.println("Carta de infantería");  
    }  
}
```

Infanteria infan= new Infanteria();

Primero se ejecuta CartaDeEquipamiento() y se reserva memoria para jugador, que es un atributo con acceso privado

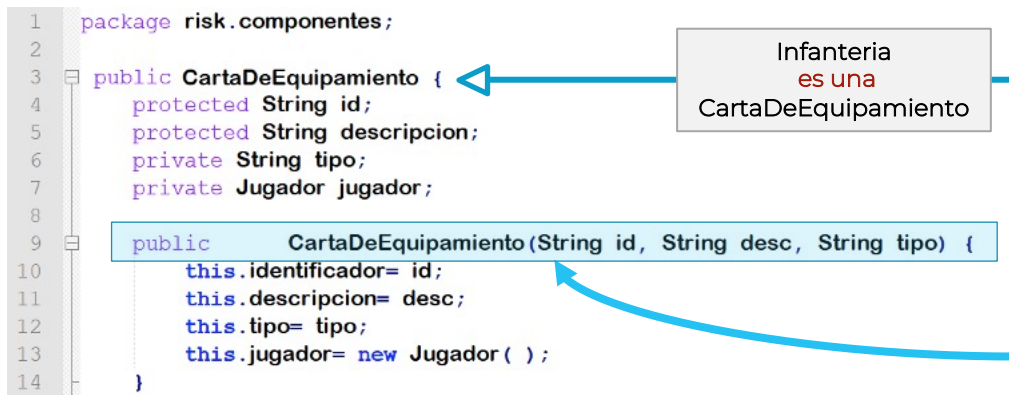
Después se ejecuta Infanteria()

# Constructores

- Si el constructor de la clase base **tiene argumentos**
    - **Solamente** se invocará a dicho constructor, puesto que **no se puede garantizar** que en la clase base exista un constructor con los mismos argumentos
    - ¿Cómo se logra la invocación de los constructores de la clase base que tienen argumentos?
      - **Solución:** el programador **invoca manualmente** al constructor de la clase base que considera oportuno en cada caso, lo que incluye al constructor que tenga los mismos argumentos que el constructor de la clase base desde la cual se invoca
- Para ello, **se hace uso de super**

# Constructores

- **super** permite acceder desde la clase derivada a los atributos, métodos y constructores de la clase base sobre los que existe cierto nivel de visibilidad



```
1 package risk.componentes;
2
3 public class CartaDeEquipamiento {
4     protected String id;
5     protected String descripcion;
6     private String tipo;
7     private Jugador jugador;
8
9     public CartaDeEquipamiento(String id, String desc, String tipo) {
10         this.identificador= id;
11         this.descripcion= desc;
12         this.tipo= tipo;
13         this.jugador= new Jugador ( );
14     }
15 }
```

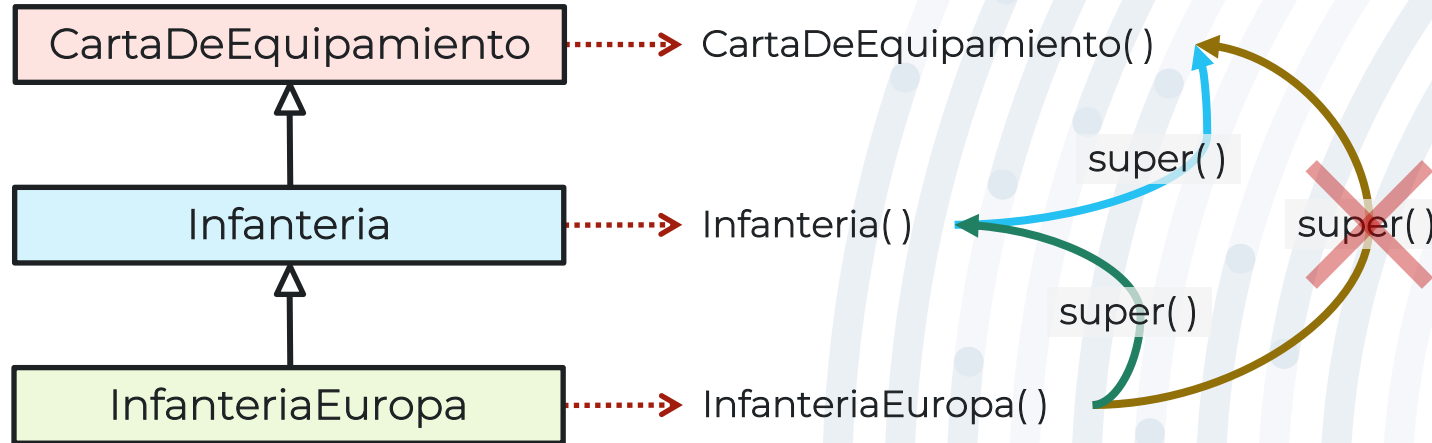
```
1 package risk.componentes;
2
3 public class Infanteria extends CartaDeEquipamiento {
4
5     public Infanteria(String id, String desc) {
6         super(id, desc, "Infanteria");
7     }
8 }
```

Se puede invocar cualquier constructor, no solamente el que tiene los mismos argumentos que el constructor base

La invocación debe tener lugar en la primera línea del constructor

# Constructores

- **super** únicamente puede acceder a los elementos de la clase base que es **inmediatamente superior** a la clase derivada
  - Con **super** **no se puede acceder directamente** a los elementos de la clase que es la clase base (CartaDeEquipamiento) de la clase base (Infanteria) de la clase derivada (InfanteriaEuropa)



# Constructores

- Buenas prácticas de programación (XXIV)

Se debe de reservar memoria e inicializar los atributos en los constructores de las clases a las que pertenecen dichos atributos, evitando hacerlo en las clases derivadas de dicha clase



- Buenas prácticas de programación (XXV)

Se debe de hacer uso de super para invocar a los constructores de las clases base para reservar memoria e inicializar los atributos (privados) que están definidos en ellas



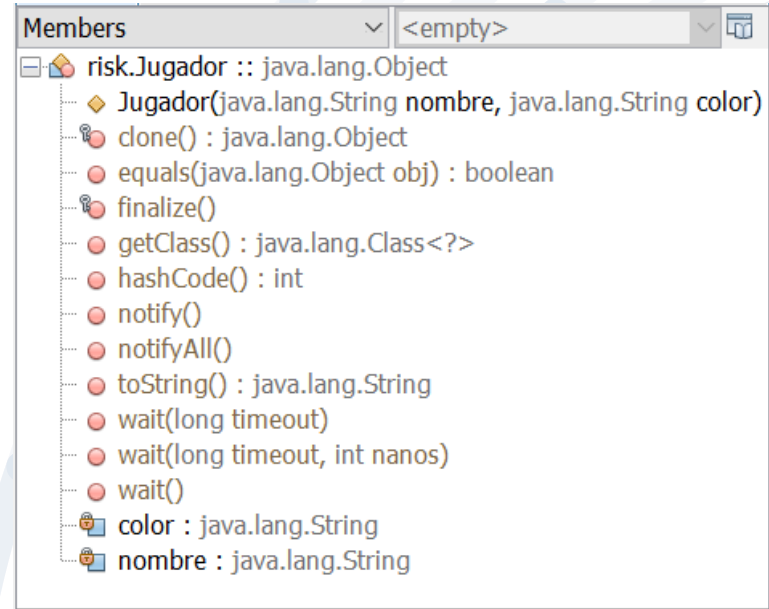
# Métodos y sobrescritura

- La sobrescritura de métodos es un mecanismo mediante el cual un método heredado de una clase base se **implementa nuevamente** en la clase derivada
  - La sobrescritura es necesaria cuando la implementación de un método de la clase base **no es válida** en la clase derivada o es conveniente adaptarla a las **características** específicas de la clase
    - Ejemplos de sobrescritura son los métodos **toString**, **equals**, **hashCode** y **clone** de la clase **Object**, que son implementados en las demás clases para adaptarlos a sus características
  - El nombre del método, sus argumentos y el tipo que devuelve deben ser los mismos en la clase base y en la derivada, mientras que el tipo de acceso debe ser **el mismo o superior**



# Métodos y sobrescritura

- **Todas** las clases que se crean en un programa de Java son clases derivadas de la clase **Object**, lo que implica que heredan todos sus métodos
  - **getClass** indica la clase a la que pertenece el objeto que lo invoca
  - **notify** y **wait** están orientados a la gestión de los **threads**
  - **finalize** se invoca cuando el recolector de basura determina que no hay más referencias al objeto y lo elimina de la memoria del programa



Es necesario  
sobreescribirlo

# Métodos y sobreescritura

- Una forma de sobrescribir los métodos en las clases derivadas es hacer uso de **super** para reutilizar la implementación del método que está en la clase base
  - Desde el método de la clase derivada **super** invoca **al mismo método** de la clase base, es decir, al método que se sobrescribe y que tiene el mismo nombre y argumentos
  - Se usa super porque la clase base **no es capaz de distinguir** entre la implementación de la clase base y la implementación de la clase derivada, con lo cual la única forma de reutilizarlo es a través de super
    - El método de la clase base **se hereda**, pero al sobrescribirse no se puede acceder a su implementación en la clase base

# Métodos y sobrescritura

```
public class Empleado {  
    private String nombre;  
    private float antigüedad;  
    private ArrayList<Proyecto> proyectosParticipado;  
    private float base;  
  
    public float calcularSueldo( ) {  
        float sueldo= this.base;  
        float factor= (this.antigüedad > 15) ? 0.02f : 0.01f;  
        for(int i=0;i<this.proyectosParticipado.size();i++) {  
            sueldo+= factor*this.proyectosParticipado.get(i).getPresupuesto();  
        }  
        return sueldo;  
    }  
}
```

float calcularSueldo( )

El cálculo del sueldo de un empleado depende de los proyectos en los cuales ha participado, así como de la antigüedad

Este cálculo es válido para cualquier empleado

## Sobrescritura del método float calcularSueldo( )

```
public class Directivo extends Empleado {  
    @Override  
    public float calcularSueldo( ) {  
        float sueldo= super.calcularSueldo();  
        return 1.1f*suelo;  
    }  
}
```

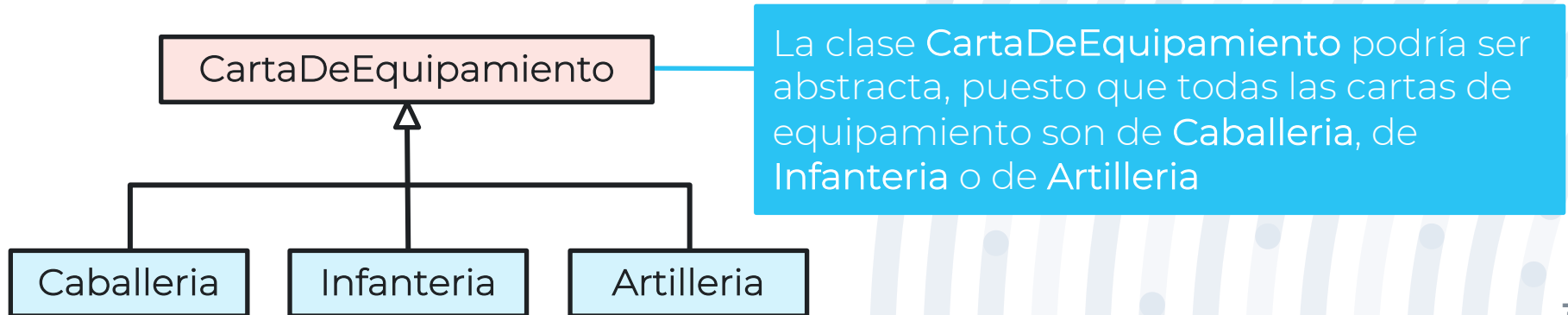
El sueldo de un directivo es un 10% mayor que el sueldo de un empleado genérico y también depende de los proyectos en los que ha participado y de la antigüedad

# Clases abstractas

- Las **clases abstractas** son clases que no se pueden instanciar
  - Pueden tener constructores, pero **no se pueden invocar a través de new**, de modo que únicamente se pueden invocar mediante **super** cuando son las clases base de otras clases
    - Ejemplo: si `CartaDeEquipamiento()` es un constructor de una clase abstracta, entonces **`new CartaDeEquipamiento()`** no está permitido
  - Pueden tener atributos y métodos implementados **de la misma forma** en la que los tienen clases que no son abstractas
  - Habitualmente, se usan **como clases base** de otras clases de las que sí se pueden crear objetos, ocupando los primeros niveles de las jerarquías de clases y facilitando con ello la **reutilización** de sus atributos, métodos y constructores en las clases derivadas

# Clases abstractas

- Las clases abstractas **deben tener implementados** la mayor cantidad posible de métodos para que las clases derivadas puedan heredarlos y así favorecer la reutilización de código
- Una clase debería ser abstracta cuando **todos los objetos** pertenecen a cualquiera de las otras clases de la jerarquía, no siendo necesario crear un objeto de la clase abstracta



# Clases abstractas

- Las clases abstractas pueden tener **métodos abstractos**
  - Son métodos que **no tienen cuerpo**, es decir, métodos que no están implementados y en los que solamente se especifica su nombre, lo que devuelve y los argumentos de entrada

```
public abstract <tipo_dato> nombre_método(<tipo_dato> nombre *) ;
```

Los métodos abstractos deben acabar siempre en “.”;

- Si una clase tiene al menos un método abstracto entonces **debe ser necesariamente** una clase abstracta
- Los métodos abstractos deben ser implementados en las clases derivadas de la clase abstracta a la que pertenecen, **siempre que dichas clases no sean abstractas**

# Clases abstractas

```
public abstract class CartaDeEquipamiento {  
    private String id;  
    private String descripcion;  
    private String tipo;  
    private Jugador jugador;  
  
    public CartaDeEquipamiento(String id, String desc, String tipo) {  
        this.identificador= id;  
        this.descripcion= desc;  
        this.tipo= tipo;  
        this.jugador= new Jugador( );  
    }  
  
    public abstract void atacar ( );  
}
```

El método `atacar()` es abstracto, lo que inmediatamente implica que la clase en la que se encuentra debe ser abstracta y que además deberá implementarse en las clases derivadas

```
public class Infanteria extends CartaDeEquipamiento {  
    public Infanteria(String id, String desc) {  
        super(id, desc, "Infanteria");  
    }  
  
    @Override  
    public void atacar( ) {  
        // Implementar código de ataque para ejércitos de infantería  
    }  
}
```

Infanteria es una clase que derivada de CartaDeEquipamiento, con lo que debe de implementar el método `atacar()`

`atacar()` deberá implementar el ataque para la clase Infanteria, que será **distinto** que el ataque para la clase Caballeria o Artilleria. Si fuese igual para todas las clases, se implementaría en la clase base

# Clases abstractas

- Una clase abstracta...
  - Puede tener todos los métodos abstractos  
Debería tener tantos métodos implementados como sea posible
  - Puede tener todos los métodos implementados
  - Puede ocupar cualquier lugar de la jerarquía de clases  
Debería ocupar los niveles superiores de la jerarquía de clases
  - Puede ser una clase derivada de una clase no abstracta  
Su clase base debería ser una clase abstracta
  - Puede no tener constructores  
Debería tener tantos constructores como sea necesario
  - Puede tener atributos de cualquier tipo



# Clases abstractas

## Puede tener todos los métodos abstractos

- En este caso el énfasis en el diseño del programa se hace en que todas las clases derivadas de la clase abstracta **tengan todas los mismos métodos** que la clase abstracta, es decir, que tengan una interfaz común
- Facilita el **mantenimiento y extensibilidad** del programa a través de la independencia entre las clases

## Debería tener tantos métodos implementados como sea posible

- En este caso el énfasis en el diseño del programa se pone en la **reutilización de código** donde todas las clases comparten la misma implementación de los métodos que la clase base, reduciendo los tiempos de desarrollo

# Clases, atributos y métodos finales

- Típicamente, una jerarquía de clases se construye con clases **abstractas** (primeros niveles), con clases **instanciables** (niveles intermedios y finales) y con clases **finales**



- Las clases finales son clases de las cuales **no se pueden crear** clases derivadas, es decir, son las últimas clases o nodos de la jerarquía de clases
- Una clase solamente debería ser final cuando se garantiza que **no será necesario crear** clases derivadas de ella

# Clases, atributos y métodos finales

- La palabra **final** en Java indica que el elemento que cualifica **no** se puede modificar
  - Si el elemento es un método, entonces si el método pertenece a una clase base, no podrá ser **sobrescrito** en una clase derivada, es decir, una vez tiene una implementación, no se podrá modificar
  - Si el elemento es un atributo, entonces **una vez** el atributo haya tomado un determinado valor, no podrá ser modificado nuevamente

Los atributos finales **no pueden tener setters** porque solo pueden modificar su valor una única vez

```
public class Empleado {  
    private String nombre;  
    private float antiguedad;  
    private ArrayList<Proyecto> proyectosParticipado;  
    private final float base;  
  
    public void setBase(float base) {  
  
        cannot assign a value to final variable base  
        ----  
        (Alt-Enter shows hints)  
  
        this.base= base;  
  
    }  
}
```

# Clases, atributos y métodos finales

- Los atributos **finales** se usan para implementar las **constantes** del programa, puesto que al establecer un valor para ellos, ya no se podrán volver a modificar
  - Típicamente, una constante se define con **final static**, ya que así es posible hacer uso de la constante **sin que sea necesario crear un objeto de la clase** en la que está definida

Declarar: `public final static <tipo_dato> <nombre_atrib> = <valor>;`

Usar: `<nombre_clase>.<nombre_atrib>`

- La palabra `static` usada como modificador en atributos o métodos hace que se **almacenen en la memoria estática**, permitiendo que estén disponibles desde el inicio del programa sin necesidad de tener que crear objetos, es decir, de instanciar las clases

# Clases, atributos y métodos finales

```
public abstract class Valor {  
    // Errores  
    public final static int CONTINENTE_NO_EXISTE= 102;  
    public final static int JUGADOR_NO_EXISTE= 103;  
    public final static int JUGADOR_YA_EXISTE= 104;  
    public final static int JUGADORES_NO_CREADOS= 105;  
    public final static int MAPA_NO_CREADO= 106;  
    public final static int MAPA_YA_CREADO= 107;  
    public final static int NO_MAS_JUGADORES= 108;  
    public final static int PAIS_NO_EXISTE= 109;  
    public final static int PAIS_NO_PERTENECE= 110;  
    public final static int PAIS_PERTENECE= 111;  
    public final static int PAISES_NO_SON_FRONTERA= 112;  
    public final static int PAIS_YA_ASIGNADO= 113;  
    public final static int COLOR_YA_ASIGNADO= 114;  
}
```

*MAPA\_NO\_CREADO* es un atributo final static, es decir, es una constante que puede ser accedida sin que sea necesario instanciar la clase **Valor**, que de hecho es una **clase abstracta**

El atributo *MAPA\_NO\_CREADO* se puede utilizar de modo directo en cualquier método de cualquier clase que haya importando la clase **Valor**

```
if(this.mapa==null) {  
    // COMANDO: crear mapa  
    if(orden.equals("crear mapa")) {  
        this.mapa= new Mapa(this.errores);  
        this.crearMapa();  
        Salida.setSalida(this.mapa.toString());  
    } else Salida.setSalida(GestionErrores.getDescripcion(Valor.MAPA_NO_CREADO));  
}
```

# Clases, atributos y métodos finales

- Buenas prácticas de programación (XXVI)

Típicamente, todas las constantes del programa se deberían de definir en una o varias clases abstractas que son accesibles por todas las clases en las que se hace uso de esas constantes



- Buenas prácticas de programación (XXVII)

Las clases y métodos finales se deberían de usar cuando se garantice que no será necesario crear clases derivadas o realizar modificaciones en los métodos de las clases derivadas, respectivamente

