Programación Avanzada

Conceptos Básicos de Orientación a Objetos (1era parte)

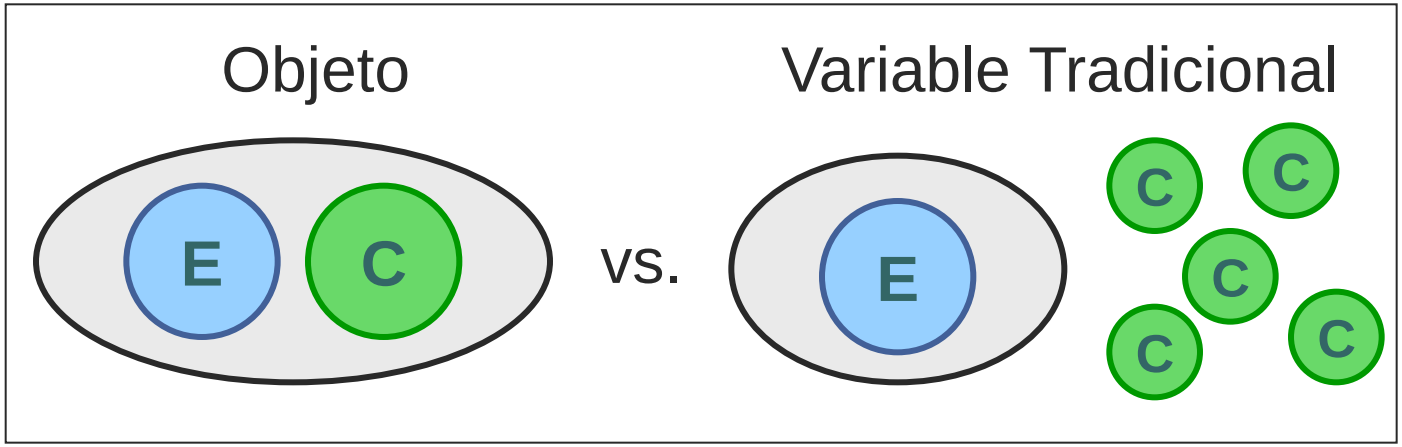
Contenido

- Construcciones Básicas
- Relaciones

Construcciones Básicas

Objeto

- Un objeto es una entidad discreta con límites e identidad bien definidos
- Encapsula estado y comportamiento:



- Es una instancia de una clase

Identidad

- Es una propiedad inherente de los objetos de ser distinguible de todos los demás
- Dos objetos son distintos aunque tengan exactamente los mismos valores en sus propiedades
- Conceptualmente un objeto no necesita de ningún mecanismo para identificarse
- La identidad puede ser realizada mediante direcciones de memoria o claves (pero formando parte de la infraestructura subyacente de los lenguajes)

Clase

- Una clase es un descriptor de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y comportamiento
- Una clase representa un concepto en el sistema que se está modelando
- Dependiendo del modelo en el que aparezca, puede ser un concepto del mundo real (modelo de análisis) o puede ser una entidad de software (modelo de diseño)

Clase (2)

Definición de una clase

```
class CEjemplo {
      ...  // definicion de
      ...  // las propiedades
      ...  // de la clase Ejemplo
}


CEjemplo *e = new CEjemplo();
delete e;
```

Instancia de una clase (i.e. un objeto)

Clase (3)

- Para crear un objeto se definen constructores

```
CEjemplo(); //por defecto
CEjemplo(params);   //común
CEjemplo(CEjemplo *);   //por copia
```

- Para destruir un objeto se define un destructor

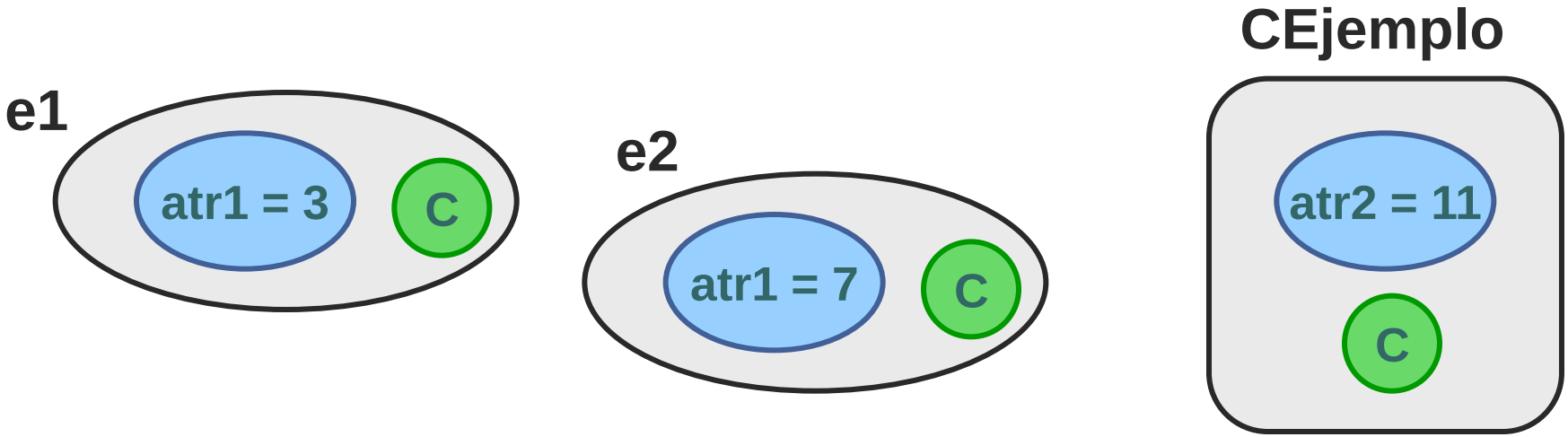
```
~CEjemplo();
```

Atributo

- Es una descripción de un compartimiento de un tipo especificado dentro de una clase
- Puede ser:
  - **De Instancia:** Cada objeto de esa clase mantiene un valor de ese tipo en forma independiente
  - **De Clase:** Todos los objetos de esa clase comparten un mismo valor de ese tipo

Atributo (2)

```
class CEjemplo {
      int atr1;           // Atributo de instancia
      static int atr2;    // Atributo de clase
      ...                 // Otras propiedades
}
```



Operación

- Es una especificación de una transformación o consulta que un objeto puede ser llamado a ejecutar
- Tiene asociada un nombre, una lista de parámetros y un tipo de retorno

Método

- Es la implementación de una operación para una determinada clase
- Especifica el algoritmo o procedimiento que genera el resultado o efecto de la operación

Operación y Método

```
class CEjemplo {
      int atr1;
      static int atr2;

      void oper(char c)
      {
            ... // un cierto algoritmo
      }
}
```

Operación

Método para `oper()` en CEjemplo

Estado

- El estado de una instancia almacena los efectos de las operaciones
- Está implementado por
  - Su conjunto de atributos
  - Su conjunto de links
- Es el valor de todos los atributos y links de un objeto en un instante dado

Comportamiento

- Es el efecto observable de una operación, incluyendo su resultado

Acceso a Propiedades

- Las propiedades de una clase tienen aplicadas calificadores de acceso
- Una propiedad de un objeto calificada con:
  - **`public`**: puede ser accedida desde cualquier punto desde el cual se tenga visibilidad sobre el objeto
  - **`private`**: puede ser accedida solamente desde los métodos de la propia clase
- Existen otros calificadores (i.e. `protected`) pero su semántica depende del lenguaje de implementación

Acceso a Propiedades (2)

- Por defecto, los atributos deben ser privados y las operaciones públicas:

```
class CEjemplo {
      private:
         int atr1;

      public:
         void oper() {
            this.atr1 = 2;    // código valido
         }
}
```

```
e.atr1      // código no valido
e->oper()   // código valido
```

Polimorfismo

- Es la capacidad de asociar diferentes métodos a la misma operación

¡No alcanza con que tengan el mismo nombre, para ser polimorfismo deben ser realmente la **misma** operación!

```
class A {
        void oper() {
                // un método
        }
}
```

```
class B {
        void oper() {
                // otro método
        }
}
```

En este caso **no** se trata de la misma operación (aunque tengan la misma firma) dado que las dos clases no están relacionadas entre sí

Data Type

- Es un descriptor de un conjunto de valores que carecen de identidad
- Data types pueden ser tipos primitivos predefinidos como:
  - Strings
  - Números
  - Fechas
- También tipos definidos por el usuario, como enumerados

Data Type (2)

- Muchos lenguajes de programación no tienen una construcción específica para data types
- En esos casos se implementan como clases:
  - Sus instancias serían formalmente objetos
  - Sin embargo, la identidad de esas instancias es ignorada

Data Type (3)

```
class Racional {
      private: int numerador, denominador;
      public:
            Racional (int = 0, int = 1);
            int getNumerador();
            int getDenominador();

            Racional operator+ (Racional);
            ... ...
}
```

Para que sea un datatype, las operaciones no pueden modificar el estado interno del objeto sino retornar uno nuevo

Data Value

- Es un valor único que carece de identidad, una instancia de un data type
- Un data value no puede cambiar su estado:
  - Eso quiere decir que todas las operaciones aplicables son “funciones puras” o consultas
- Los data values son usados típicamente como valores de atributos

Valores y Cambios de Estado

- El valor “4” no puede ser convertido en el valor “5”
- Se le aplica la operación suma con argumento “1” y el resultado es el valor “5”
- A un objeto persona se le puede cambiar la edad:
  - Reemplazando el valor de su atributo “edad” por otro valor nuevo
  - El resultado es la misma persona con otra edad

Identidad o no Identidad

- ¿Cómo saber si un elemento tiene o no identidad?:
  - Dos objetos separados que sean idénticos lucen iguales pero no son lo mismo (son distinguibles por su identidad)
  - Dos data values separados que sean idénticos son considerados lo mismo (no son distinguibles por no tener identidad)

Relaciones

Asociación

- Una asociación describe una relación semántica entre clasificadores (clases o data types)
- Las instancias de una asociación (links) son el conjunto de tuplas que relacionan las instancias de dichos clasificadores
- Cada tupla puede aparecer como máximo una sola vez en el conjunto

Asociación (2)

- Una asociación entre clases indica que es posible “conectar” entre sí instancias de dichas clases
- Cuando se desea poder conectar objetos de ciertas clases, éstas deben estar relacionadas por una asociación
- Una asociación R entre clases A y B puede entenderse como $\mathrm{R} \subseteq \mathrm{A} \times \mathrm{B}$
  - Los elementos en R pueden variar con el tiempo

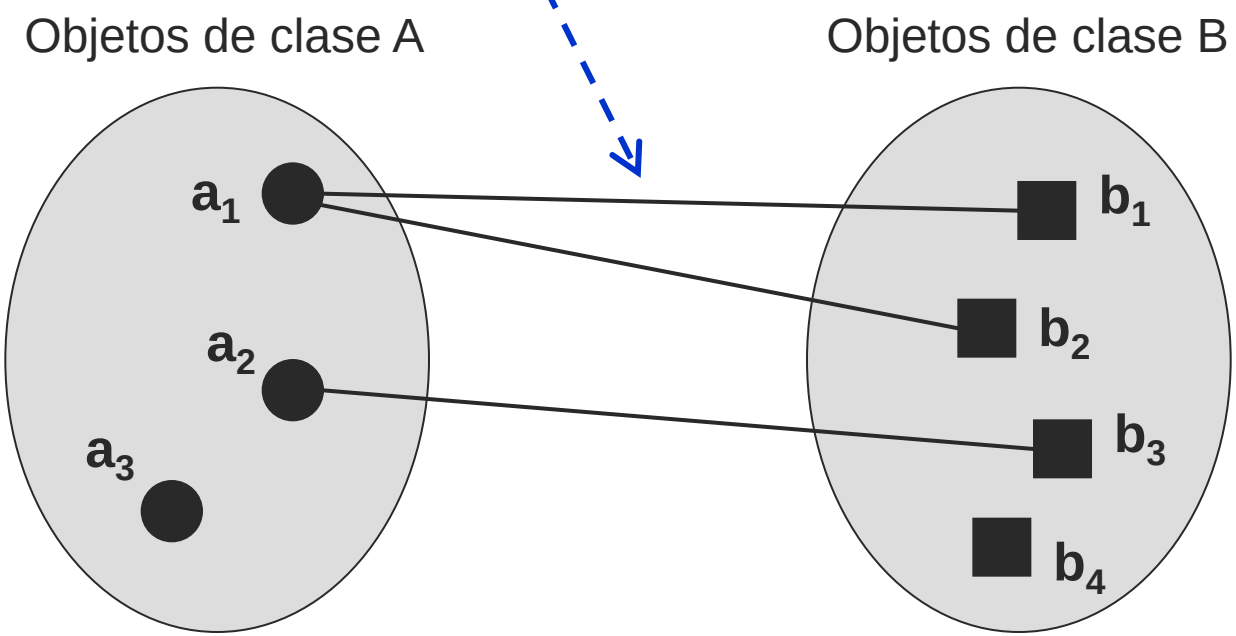
Link

- Es una tupla de referencias a instancias (objetos o data values)
- Es una instancia de una asociación
- Permite visibilidad entre todos las instancias participantes

Link (2)

- Ejemplo: asociación R entre clases A y B

$$R = \{<a_1,b_1>, <a_1,b_2>, <a_2,b_3>\}$$

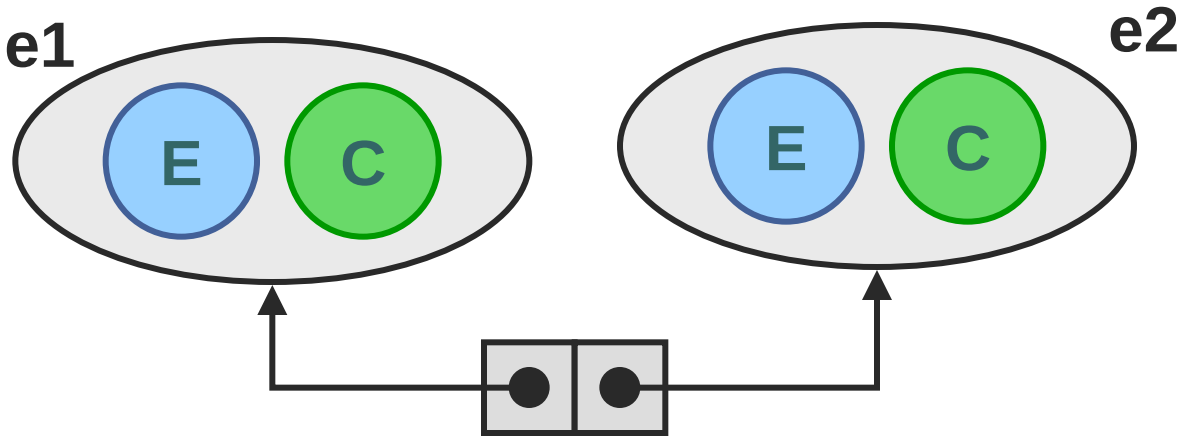


Representación de Asociaciones

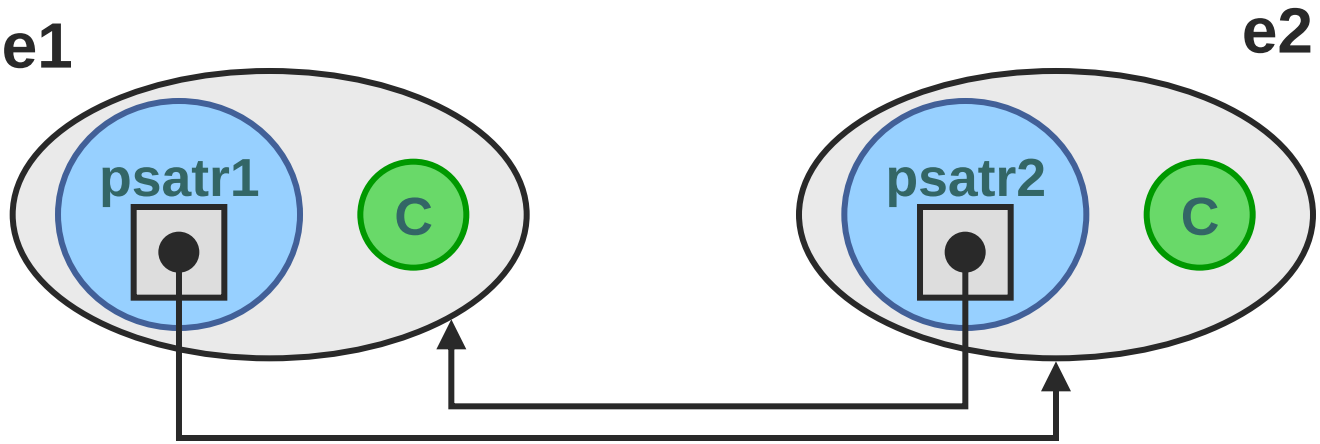
- Casi ningún lenguaje provee construcciones específicas para implementar asociaciones
- Para ello se suelen introducir “pseudoatributos” en las clases involucradas
- De esta manera un link no resulta implementado exactamente igual a su representación conceptual
- Una tupla es dividida y un componente es ubicado en el objeto referenciado por el otro componente de la tupla

Representación de Asocs. (2)

Representación conceptual



Implementación usual



Representación de Asocs. (3)

- Ejemplo: Asociación entre **Persona** y **Empresa**

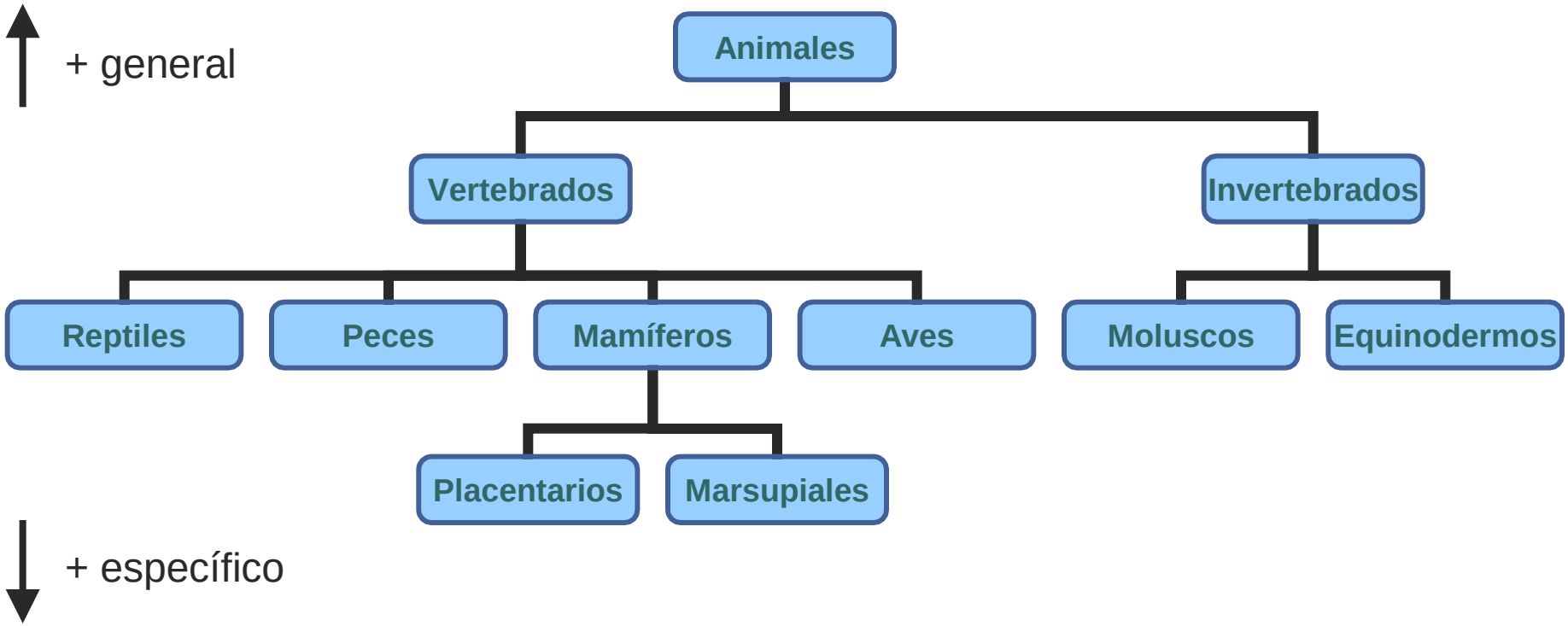
```
class Persona {
      private:
            String nombre;    // atributo
            Empresa *miEmp;   // pseudoatributo
      ...
}
```

- El tipo de un pseudoatributo suele ser una clase, pero el de un atributo debe ser un data type
- Por cuestiones de costo si una de las visibilidades no es necesaria usualmente no se implementa

Generalización

- Una generalización es una relación taxonómica entre un elemento (clase, data type, interfaz) más general y entre un elemento más específico
- El elemento más específico es consistente (tiene todas sus propiedades y relaciones) con el más general, y puede contener información adicional

Taxonomía


+ general
Animales
Vertebrados
Invertebrados
Reptiles
Peces
Mamíferos
Aves
Moluscos
Equinodermos
Placentarios
Marsupiales
+ específico

Clase Base y Clase Derivada

- Cuando dos clases están relacionadas según una generalización, a la clase más general se la denomina *clase base* y a la más específica *clase derivada* de la más general
- A una clase base se la denomina también *superclase* o *padre*
- A una clase derivada se la denomina también *subclase* o *hijo*

Clase Base y Clase Derivada (2)

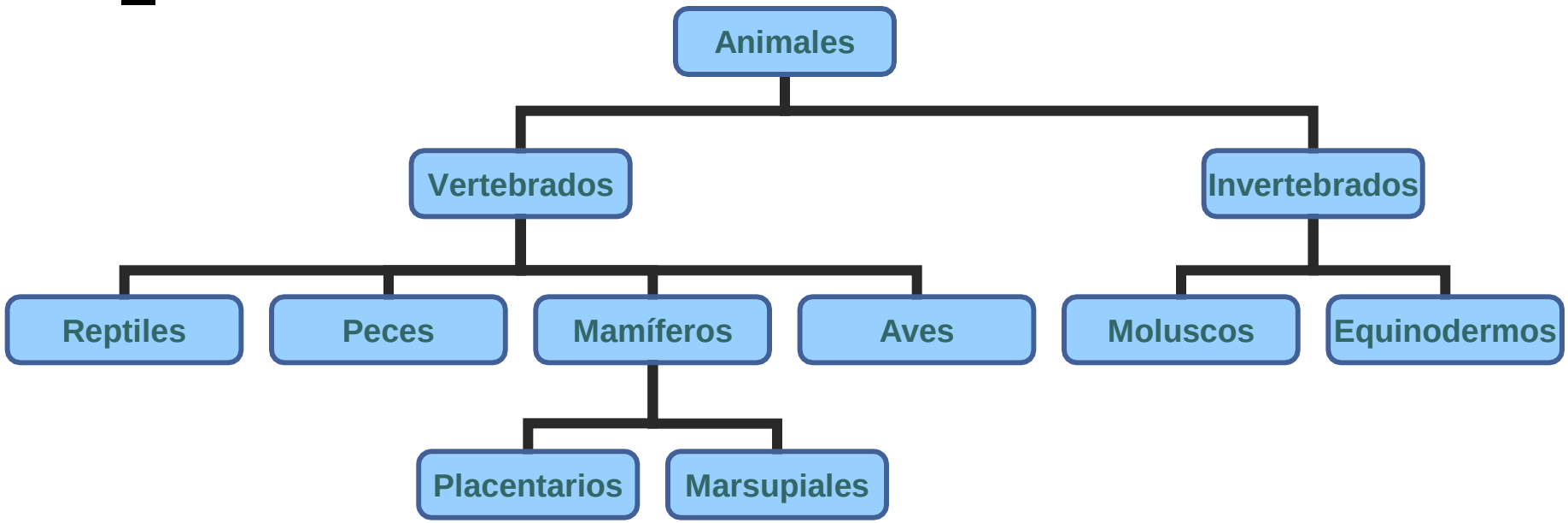
- Una clase puede tener cualquier cantidad de clases base, y también cualquier cantidad de clases derivadas.

```
class Vehiculo {
      ... // propiedades de vehiculo
}

class Auto : public Vehiculo {
      ... // props especificas de auto
}

class Moto : public Vehiculo {
      ... // props especificas de moto
}
```

Ancestros y Descendientes

- Los ancestros de una clase son sus padres (si existen), junto con los ancestros de éstos
- Los descendientes de una clase son sus hijos (si existen), junto con los descendientes de éstos
- Una clase es clase base directa de sus hijos, y una clase es clase derivada directa de sus padres
- Una clase es clase base indirecta de los descendientes de sus hijos, y una clase es clase derivada indirecta de los ancestros de sus padres

Ancestros y Descendientes (2)



- Ancestros de Marsupiales son {Mamíferos, Vertebrados, Animales}
- Descendientes de Invertebrados son {Moluscos, Equinodermos}
- Ave es clase derivada directa de Vertebrados e indirecta de Animales
- Vertebrados es clase base directa de Mamíferos e indirecta de Marsupiales

Subclassing

- Se define la relación entre clases:

  $(<:) : \text{Clase} \times \text{Clase} \rightarrow \text{Prop}$

  donde,

  $(B,A) \in (<:) \Leftrightarrow B$ es clase derivada de A

- **Observación:** La relación <: define un orden parcial entre clases y es idéntica, transitiva y antisimétrica.

Subsumption

- Es una propiedad que deben cumplir todos los objetos, también conocida como *intercambiabilidad*
- Un objeto de clase base puede ser sustituido por un objeto de clase derivada (directa o indirecta)
- Por lo tanto: $b : B \wedge B <: A \Rightarrow b : A$
- Esto se puede leer como: “un objeto instancia de una clase derivada es también instancia de cualquier clase base”
  - Ejemplo: “Todo auto es un vehículo”

Descriptors

- Un *full descriptor* es la descripción completa que es necesaria para describir a un objeto
- Contiene la descripción de todos los atributos, operaciones y asociaciones que el objeto contiene
- Un *segment descriptor* son los elementos que efectivamente se declaran en un modelo o en el código (por ejemplo, clases) y contienen las propiedades heredables que son:
  - los atributos
  - las operaciones y los métodos
  - la participación en asociaciones (los pseudoatributos)

Descriptors (2)

```
class Empleado {
      private: string nombre;
               Empresa miEmp;
      public:  string getNombre() {
                  return nombre;
               }


class Fijo : pulic Empleado {
      private: float sueldo;
      public:  float getSueldo() {
                  return sueldo;
               }
}
```

$SD_{Empleado}$

- ATT_{nombre}
- PSA_{miEmp}
- $OP_{getNombre}$
- $MET_{Empleado::getNombre}$

SD_{Fjio}

- ATT_{sueldo}
- $OP_{getSueldo}$
- $MET_{Fijo::getSueldo}$

Descriptors (3)

- En un lenguaje orientado a objetos, la descripción de un objeto es construida incrementalmente a partir de segmentos
- Los segmentos son combinados mediante herencia para producir el descriptor completo de un objeto
- El mecanismo de herencia define cómo full descriptors son producidos a partir de un conjunto de segment descriptors conectados entre sí por generalización
- Los full descriptors son implícitos pero son quienes definen la estructura de objetos concretos

Herencia

- Es el mecanismo por el cual se permite compartir propiedades entre una clase y sus descendientes
- Define la forma en que el full descriptor de una clase es generado
  - Si una clase no tiene ningún padre entonces su full descriptor coincide con su segment descriptor
  - Si tiene uno o más padres, entonces su full descriptor se construye como la unión de su propio segment descriptor con los de todos sus ancestros

Herencia (2)

- La clase para la cual se genera el full descriptor hereda las propiedades especificadas en los segmentos de sus ancestros
- Para una clase, no es posible declarar un atributo u operación con el mismo prototipo en más de uno de los segmentos:
  - Si eso ocurriera el modelo estaría mal formado

Herencia (3)

- $FD_{Empleado} = SD_{Empleado}$
- $FD_{Fijo} = SD_{Empleado} \oplus SD_{Fijo}$

$FD_{Empleado}$

ATT_{nombre}

PSA_{miEmp}

$OP_{getNombre}$

$MET_{Empleado::getNombre}$

FD_{Fijo}

ATT_{nombre} , ATT_{sueldo}

PSA_{miEmp}

$OP_{getNombre}$, $OP_{getSueldo}$

$MET_{Empleado::getNombre}$, $MET_{Fijo::getSueldo}$

Operación Abstracta

- En una clase, una operación es abstracta si no tiene un método asociado
- Tener una operación abstracta es condición suficiente para que una clase sea abstracta
- Una clase puede ser abstracta aún sin tener operaciones abstractas

Operación Abstracta (2)

```
class Empleado {
        private: string nombre;
        public:  virtual float getSueldo() = 0;
}

class Fijo : public Empleado {
        private: float sueldo;
        public:  float getSueldo(){
                   return sueldo;}
}

class Jornalero : public Empleado {
        private: float valorHora;
                    int cantHoras;
        public:  float getSueldo(){
                   return valorHora*cantHoras;}
}
```

Gracias a la **herencia** es posible que una operación esté en más de una clase

Gracias al **polimorfismo** es posible asociarle métodos diferentes en cada una de ellas

Operación Abstracta (3)

$FD_{Empleado}$

ATT_{nombre}

$OP_{getLiquidacion}$

FD_{Fijo}

ATT_{nombre} , ATT_{sueldo}

$OP_{getLiquidacion}$

$MET_{Fijo::getLiquidacion}$

$FD_{Jornalero}$

ATT_{nombre} , $ATT_{valorHora}$, $ATT_{cantHoras}$

$OP_{getLiquidacion}$

$MET_{Jornalero::getLiquidacion}$

Clase Abstracta

- Algunas clases pueden ser abstractas:
  - Ningún objeto puede ser creado directamente a partir de ellas
  - No son instanciables
- Las clases abstractas existen solamente para que otras hereden las propiedades declaradas por ellas

Clase Abstracta (2)

```
class Empleado {
      private: string nombre;
      public:  virtual float getSueldo() = 0;
}

class Fijo : public Empleado {
      private: float sueldo;
      public:  float getSueldo(){...}
}

class Jornalero : public Empleado {
      private: float valorHora;
               int cantHoras;
      public:  float getSueldo(){...}
}
```

Observación:
Empleado es una clase abstracta por tener todos sus operaciones abstractas. Debido a esto, todo empleado es fijo ó jornalero

Instancia Directa e Indirecta

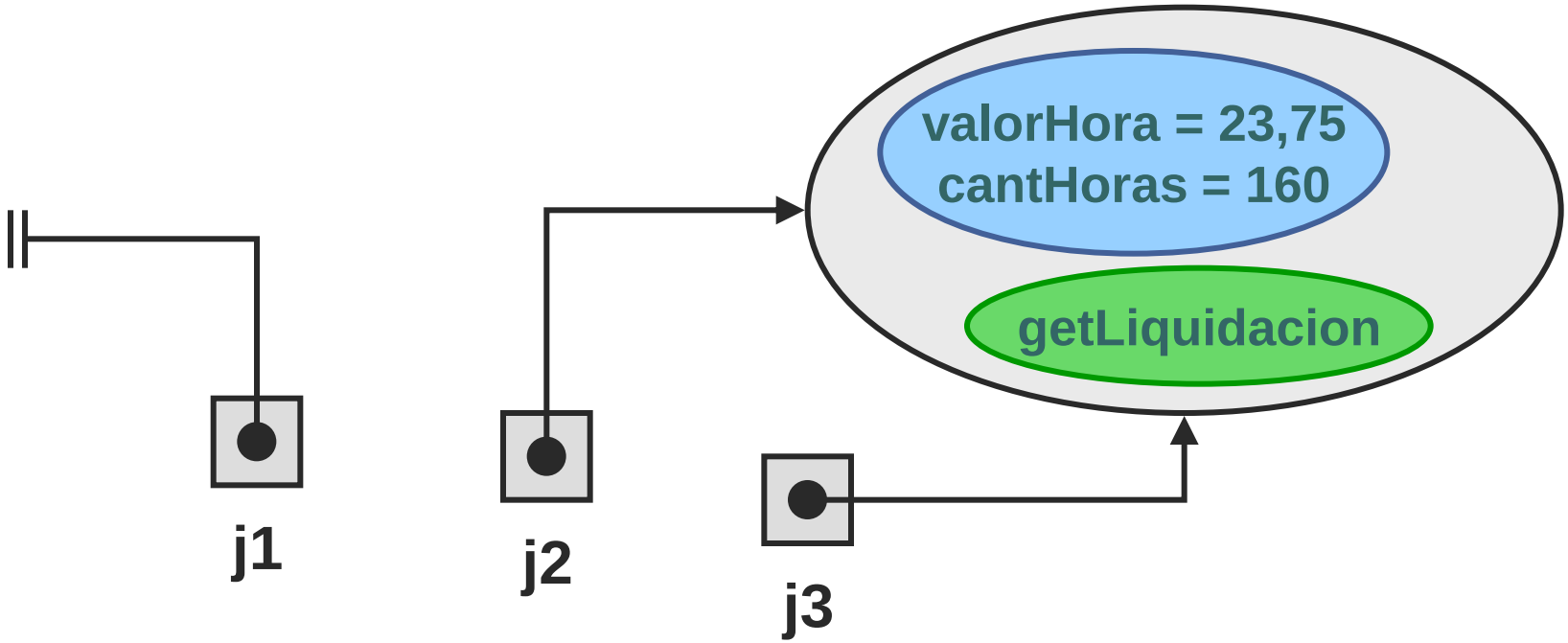
- Si un objeto es creado a partir del full descriptor generado para una cierta clase *C*, entonces se dice que ese objeto es *instancia directa* de *C*
- Además se dice que el objeto es *instancia indirecta* de todas las clases ancestras de *C*
- Ejemplo: `Jornalero *j = new Jornalero();`
  - `j` es instancia directa de Jornalero
  - `j` es instancia indirecta de Empleado

Referencia

- Una referencia es un valor en tiempo de ejecución que es: void ó attached
- Si es attached la referencia identifica a un único objeto (se dice que la referencia está adjunta a ese objeto particular)
- Si es void la referencia no identifica a ningún objeto

Referencia (2)

```
Jornalero *j1 = null;                  // void
Jornalero *j2 = new Jornalero(); // attached
Jornalero *j3 = *j2;                   // attached
```



Tipo Estático y Dinámico

- El *tipo estático* de un objeto es el tipo del cual fue declarada la referencia adjunta a él:
  - Se conoce en tiempo de compilación
- El *tipo dinámico* de un objeto es el tipo del cual es instancia directa
- En ciertas situaciones ambos tipos coinciden por lo que pierde el sentido realizar tal distinción

Tipo Estático y Dinámico (2)

- En situaciones especiales, el tipo dinámico difiere del tipo estático y se conoce en tiempo de ejecución
- Este tipo de situación es en la que la referencia a un objeto es declarada como de una clase ancestra del tipo del objeto:
  - Lo cual es permitido por subsumption
- Se cumple la siguiente relación entre los tipos de *obj* :
  - *TipoDinamico*(*obj*) <: *TipoEstatico*(*obj*)

Tipo Estático y Dinámico (3)

```
Empleado *e = new Jornalero();
```

TipoEstatico(e) = Empleado
TipoDinamico(e) = Jornalero

```
Empleado *e;
if (cond)
      e = new Fijo();
else
      e = new Jornalero();
```

¿Cuál es el tipo dinámico de *e*?