

Programación 4

Conceptos Básicos de Orientación a
Objetos (1^{era} parte)

Contenido

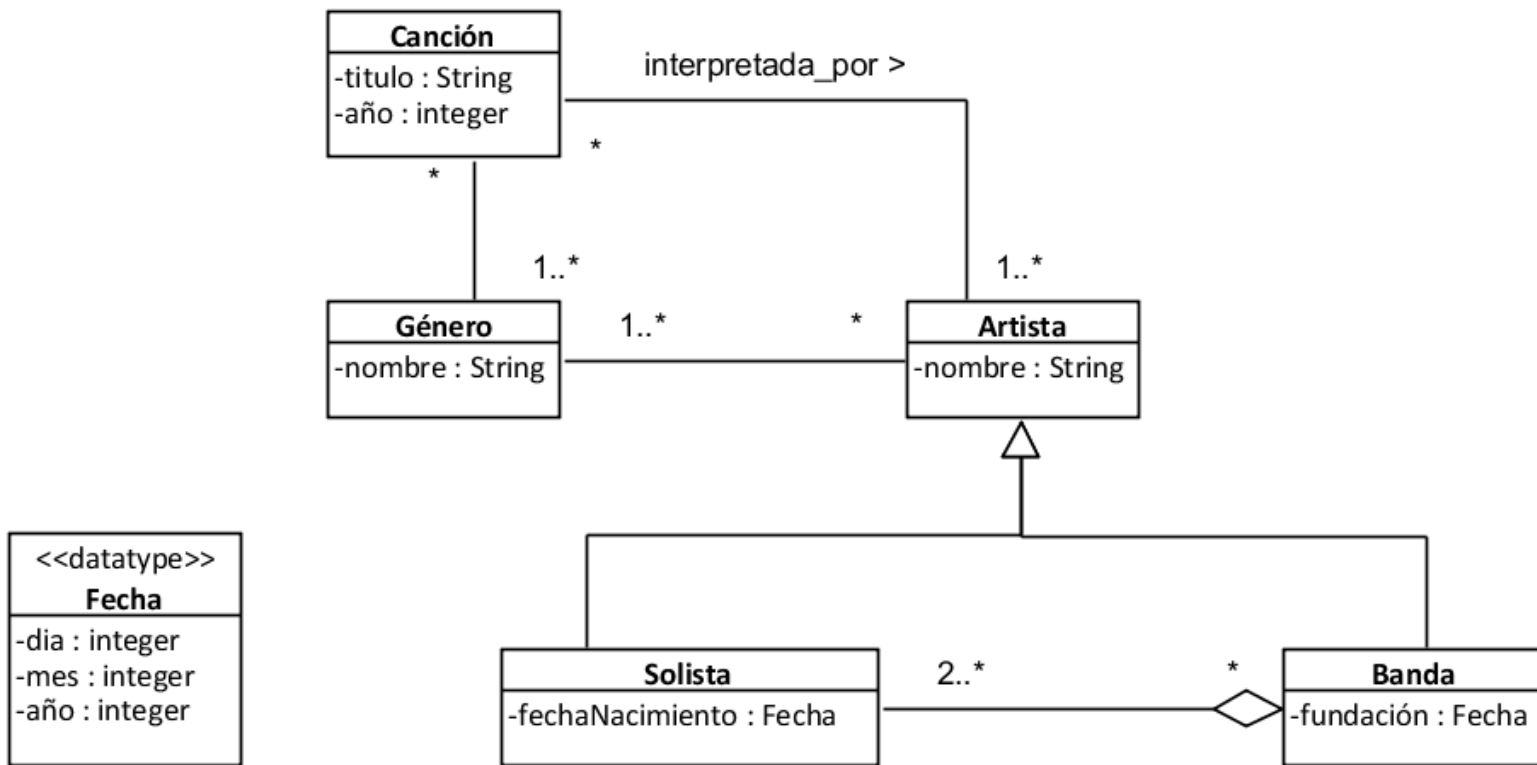
- Construcciones Básicas
- Relaciones
- Despacho

Construcciones Básicas

Caso de Estudio

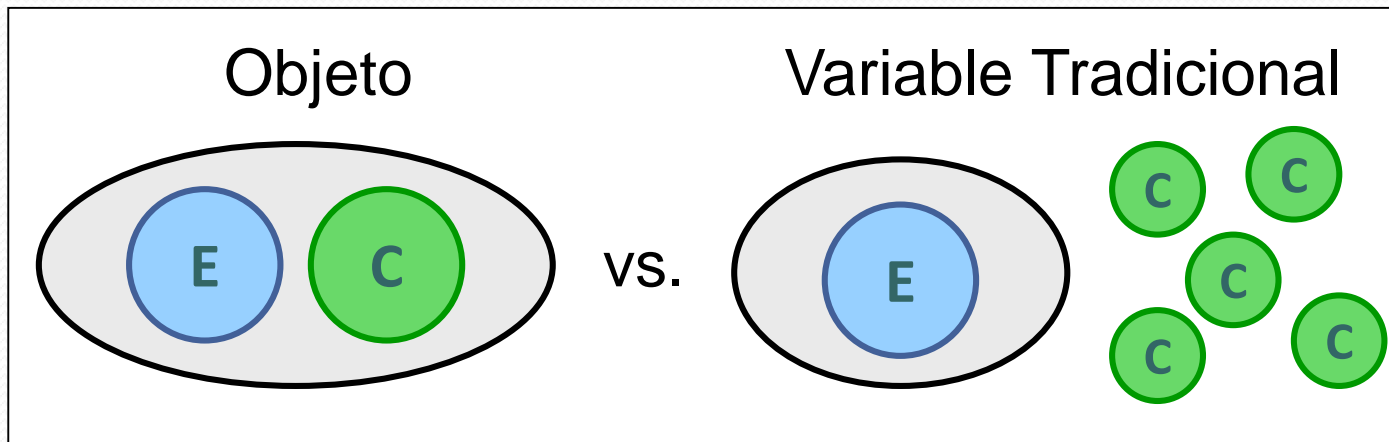
- Caso: **Usuario** crea **Canciones** en la biblioteca de música
- Responsabilidad: Crear la canción en la biblioteca y asociarla correctamente con el resto con los géneros y artistas correspondientes

Caso de Estudio (2)



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 Cancion {  
    ... // definicion de  
    ... // los atributos /  
    ... // operaciones  
};
```

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

```
Cancion *c = new Cancion();  
delete c;
```

Clase (3)

- Para crear un objeto se definen constructores

```
//por defecto (sin parámetros)
```

```
Cancion::Cancion ()
```

```
//común (con parámetros)
```

```
Cancion::Cancion(string titulo, int anio)
```

```
//por copia
```

```
Cancion::Cancion(Cancion &)
```

- Para destruir un objeto se define un destructor

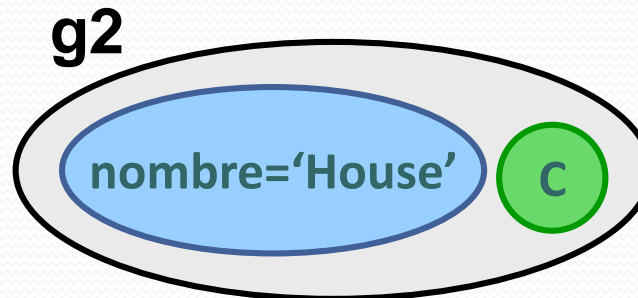
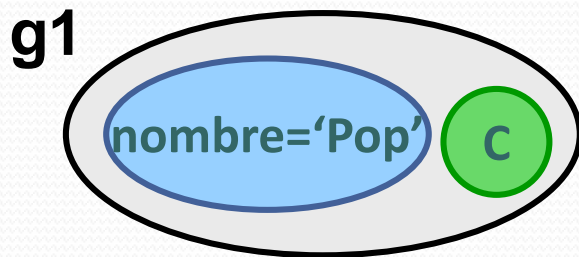
```
Cancion::~~Cancion();
```

Atributo

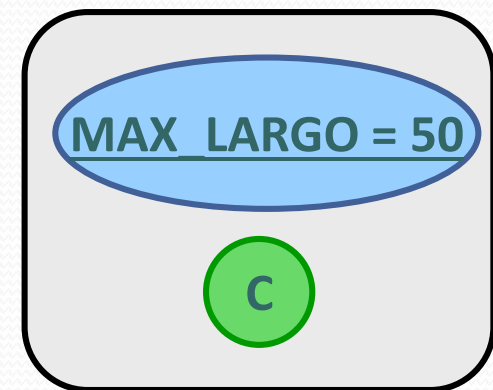
- Es una descripción de un compartimiento de un tipo especificado dentro de una clase
- Puede ser:
 - **De Instancia:** Cada objeto de esa clase mantiene un valor de ese tipo en forma independiente
 - **De Clase:** Todos los objetos de esa clase comparten un mismo valor de ese tipo

Atributo (2)

```
class Genero {  
private:  
    string nombre;           // Atributo de instancia  
    static string MAX_LARGO; // Atributo de clase  
}
```



Clase Genero



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 Genero {  
    private:  
        string nombre;  
    public:  
        string getNombre();  
};
```



Operación

```
string Genero::getNombre(){  
    return "el genero es" + nombre;  
}
```



Método para getNombre() en Genero

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
- En C++ 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
 - **protected**: en C++ permite visibilidad a la clase y su derivada
- Otros lenguajes pueden definir diferentes tipos de acceso

Acceso a Propiedades (2)

- Los atributos deberían ser privados y las operaciones públicas:

```
class Genero {  
    private:  
        string nombre;  
  
    public:  
        string getNombre();  
};  
string Genero::getNombre() {  
    return nombre;  
}
```

```
Genero *g = ...;
```

```
// código no válido fuera de la clase
```

```
g->nombre
```

```
// código válido
```

```
c->getNombre()
```

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

Data Type (3)

```
class Complejo {  
private:  
    float i, j;  
public:  
    Complejo (int i, int j);  
    Complejo operator+ (Complejo &);  
    // . . . . .  
};
```

```
Complejo operator -(Complejo i, Complejo d);
```

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” (sin efectos secundarios) 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 $R \subseteq A \times B$
 - Los elementos en R pueden variar con el tiempo

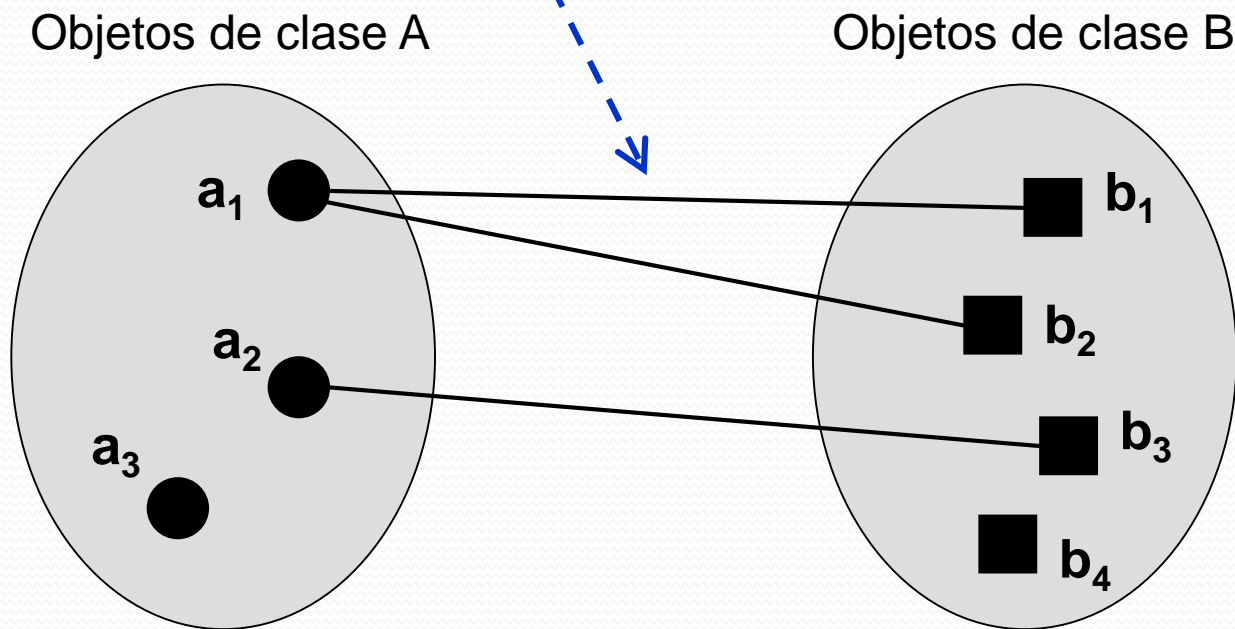
Link

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

Link (2)

- Ejemplo: asociación R entre clases A y B

$$R = \{ \langle a_1, b_1 \rangle, \langle a_1, b_2 \rangle, \langle a_2, b_3 \rangle \}$$

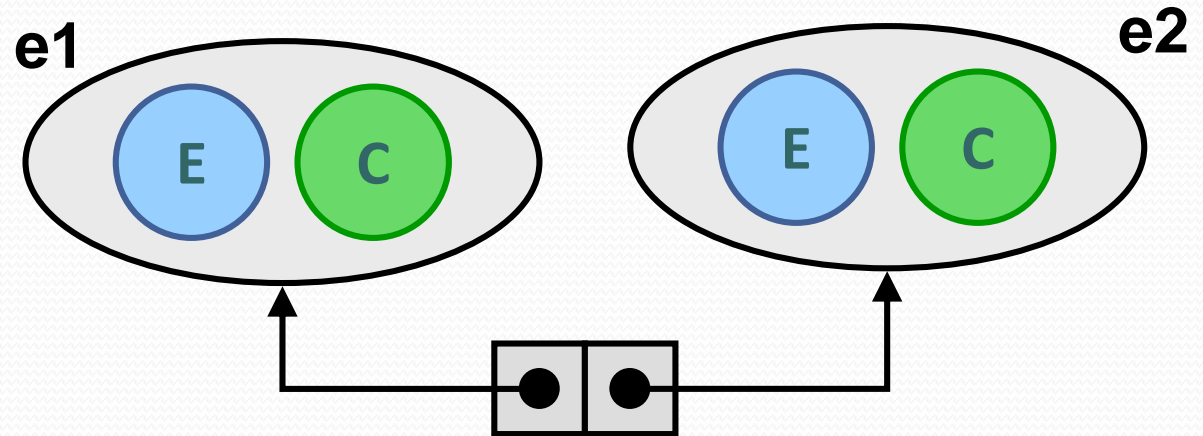


Representación de Asociaciones

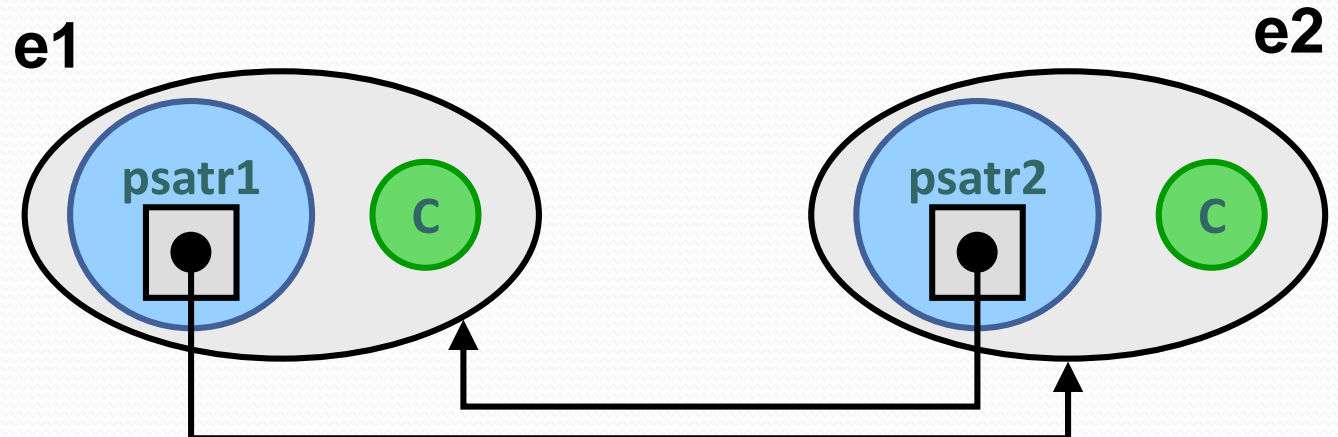
- Casi ningún lenguaje provee construcciones específicas para implementar asociaciones
- Para ello se suelen introducir “pseudoatributos” en las clases involucradas
- De esta manera un link no resulta implementado exactamente igual a su representación conceptual
- Una tupla es dividida y un componente 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 **Cliente** y **Transaccion**

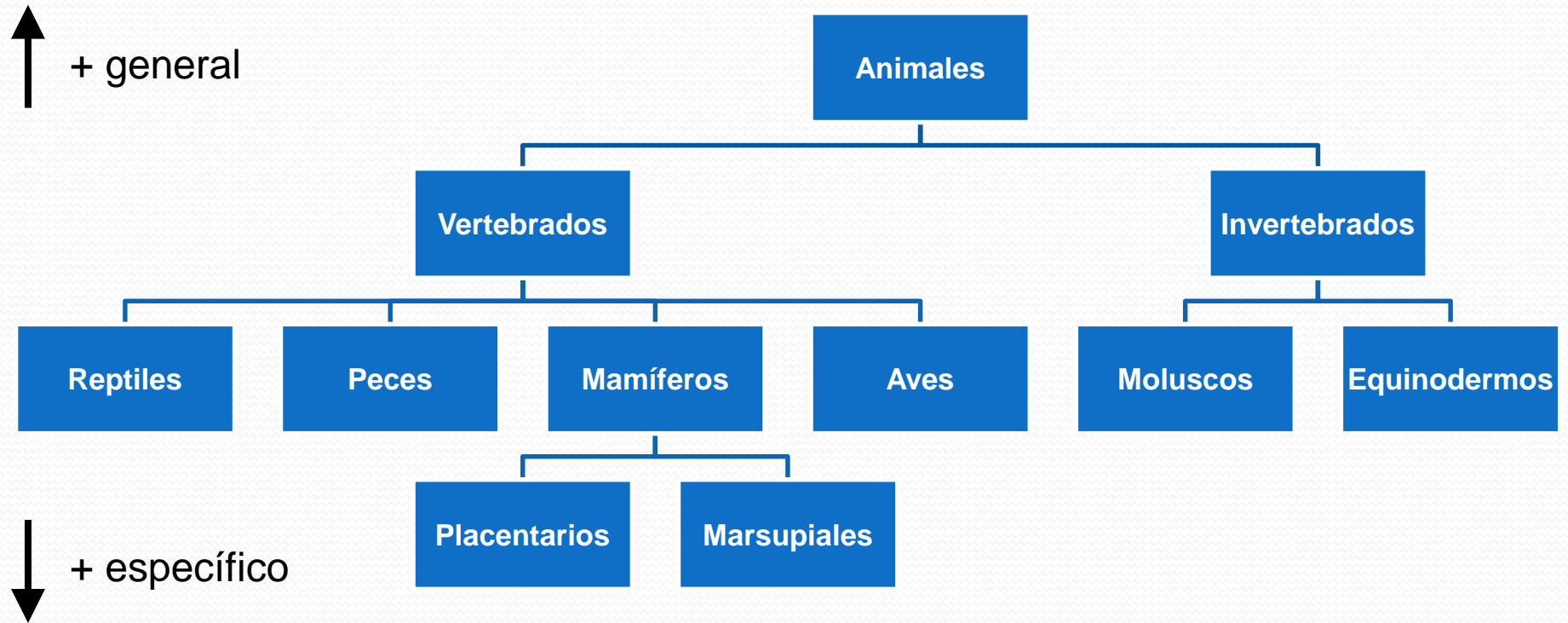
```
class Genero {  
    private:  
        string nombre;           // atributo  
        Vector<Cancion> canciones; // pseudoatributo  
        ...  
};
```

- 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



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 Artista {  
    ...  
};
```

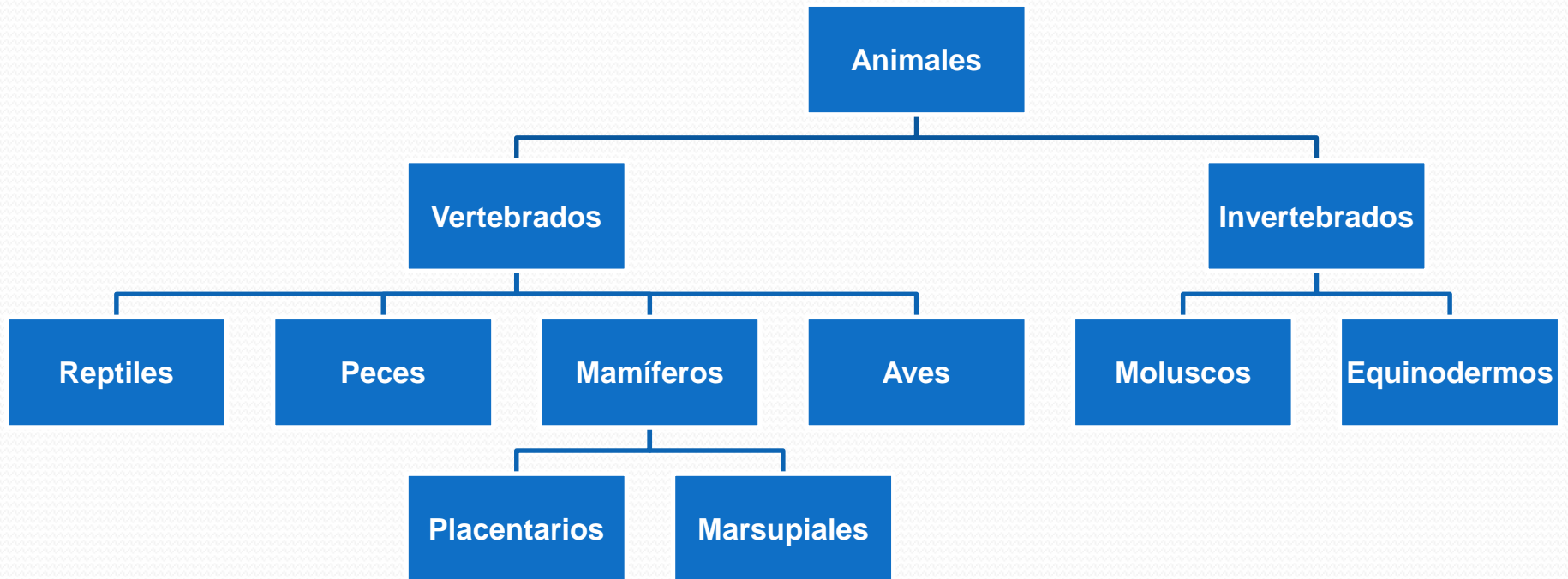
```
class Solista: public Artista {  
    ...  
};
```

```
class Banda: public Artista{  
    ...  
};
```

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:
($<:$) contenido en Clase \times Clase
donde,
 $B <: A \Leftrightarrow B$ es clase derivada de A
- **Observación:** La relación $<:$ define un orden parcial entre clases.

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: $\forall b: B, \text{ si } B <: A \Rightarrow b: A$
- Esto se puede leer como: “un objeto instancia de una clase derivada es también instancia de cualquier clase base”
 - Ejemplo: “Todo Entero es un Real”

Herencia

- Es el mecanismo por el cual se permite compartir propiedades entre una clase y sus descendientes
 - Si una clase no tiene ningún padre entonces sus propiedades son las definidas en la misma clase.
 - Si tiene padre, entonces sus propiedades son las de la unión de las propias junto con las de su padre

Herencia (2)

- Se dice que la clase hereda las propiedades especificadas por sus ancestros
- Entonces la los atributos y operaciones de una clase son los que declara más los de su clase base
- Se puede decir que una clase derivada *extiende* a su clase base.

Herencia (3)

```
class Artista {  
    private: string nombre;  
    public:  string getNombre();  
};  
string Artista::getNombre(){  
    return nombre;  
}
```

```
class solista: public Artista {  
    public: string toString();  
};
```


```
string solista::toString (){  
    return "solista: nombre=" + getNombre();  
}
```

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

Redefinición de Operaciones

- Cuando en una jerarquía de generalizaciones se encuentra más de un método asociado a la misma operación, se dice que dicha operación está redefinida
- Para una clase determinada, el método asociado a dicha operación será aquel que se encuentre más próximo en la jerarquía

Redefinición de Operaciones (2)

```
class Artista {  
    private: string nombre;  
    public: virtual string toString();  
};  
string Artista::toString(){  
    return nombre;  
}
```

```
class Banda: public Artista {  
    private: Fecha *fundacion;  
    public: virtual string toString();  
};  
string Banda::toString(){  
    return "Banda ->" + Artista::toString();  
}
```

Artista

ATT *Artista::nombre*
Op *Artista::toString()*
MET *Artista::toString()*

Banda

ATT *Artista::nombre*
ATT *Banda::fundacion*
Op *Artista::toString()*
MET *Artista::toString()*
MET *Banda::toString()*

Redefinición de Operaciones (3)

- Para la clase *Banda* el método asociado a *toString()* es el de la clase *Banda* (ocultando al heredado desde la clase *Artista*, para usos desde fuera de la clase)

```
ATTArtista::nombre,  
ATTBanda::fundacion  
OpArtista::toString()  
METArtista::toString(), METBanda::toString()
```

- El método heredado puede ser referenciado en la clase *Banda* usando su firma completa (*Artista::toString()*)

Sobrecarga

- Es la capacidad que tiene un lenguaje de permitir que varias operaciones tengan el mismo nombre sintáctico, pero recibiendo diferente cantidad/tipo de parámetros
- Ejemplos de sobrecarga:
 - `string Banda::toString()`
 - `String Banda::toString(bool conFundacion)`
- La sobrecarga no es un concepto exclusivo de la orientación a objetos

Sobrecarga vs. Redefinición

- La redefinición trata de la **misma operación**, con diferentes métodos
- La sobrecarga trata de **diferentes operaciones**, con diferentes métodos

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 Numero {
    // operación sin método (abstracta)
    public: virtual float tofloat() = 0;
};

class Racional : public Numero {
    private: int numerador, denominador;
    public: float toFloat ();
};

float Racional::toFloat(){
    if(denominador == 0) return NAN;
    else return numerador / (float) denominador;
}

class Entero: public Numero {
    private: int valor;
    public: float toFloat ();
}

float Entero::toFloat(){
    return (float) valor;
}
```

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)

Clase_{Lista}

OP *Lista::tamano()*

OP *Lista::getElemento(int indice)*

Clase_{Arreglo}

ATT *Arreglo::tam*

ATT *Arreglo::elementos*

OP *Lista::tamano()*

MET *Arreglo::tamano()*

OP *Lista::getElemento(int indice)*

MET *Arreglo::getElemento(int indice)*

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 Lista {  
public:  
    virtual int tamaño() = 0;  
    virtual string getElemento(int idx) = 0;  
};
```

```
class Arreglo {  
private:  
    int tam;  
    string *elementos;  
public:  
    int tamaño();  
    string getElemento(int idx);  
};  
int Arreglo::tamaño(){  
    return tam;  
}  
string Arreglo::getElemento(int idx){  
    return elementos[idx];  
}
```

Observación:

Lista es una clase abstracta por tener una operación abstracta.

Debido a esto, toda instancia concreta de Lista es un Arreglo

Despacho

Instancia Directa e Indirecta

- Si un objeto es creado 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:
Arreglo *a = new Arreglo();
 - a es instancia directa de Arreglo
 - a es instancia indirecta de Lista (y sus ancestros, si tiene)

Invocación

- Una invocación se produce al acceder a una propiedad de una instancia que sea una operación
- El resultado es la ejecución del método que la clase de dicha instancia le asocia a la operación accedida (despacho)

Despacho

- Con la introducción de subsumption es necesario reexaminar el significado de la invocación de operaciones
- Suponiendo $b : B$ y $B <: A$ es necesario determinar el significado de $b.f()$ cuando B y A asocian métodos distintos a la operación $f()$
- Por tratarse de $b : B$ resultaría natural que el método despachado por la invocación sea el de la clase B
- Sin embargo por subsumption también $b : A$, por lo que sería posible que el método a despachar sea el de la clase A

Despacho (2)

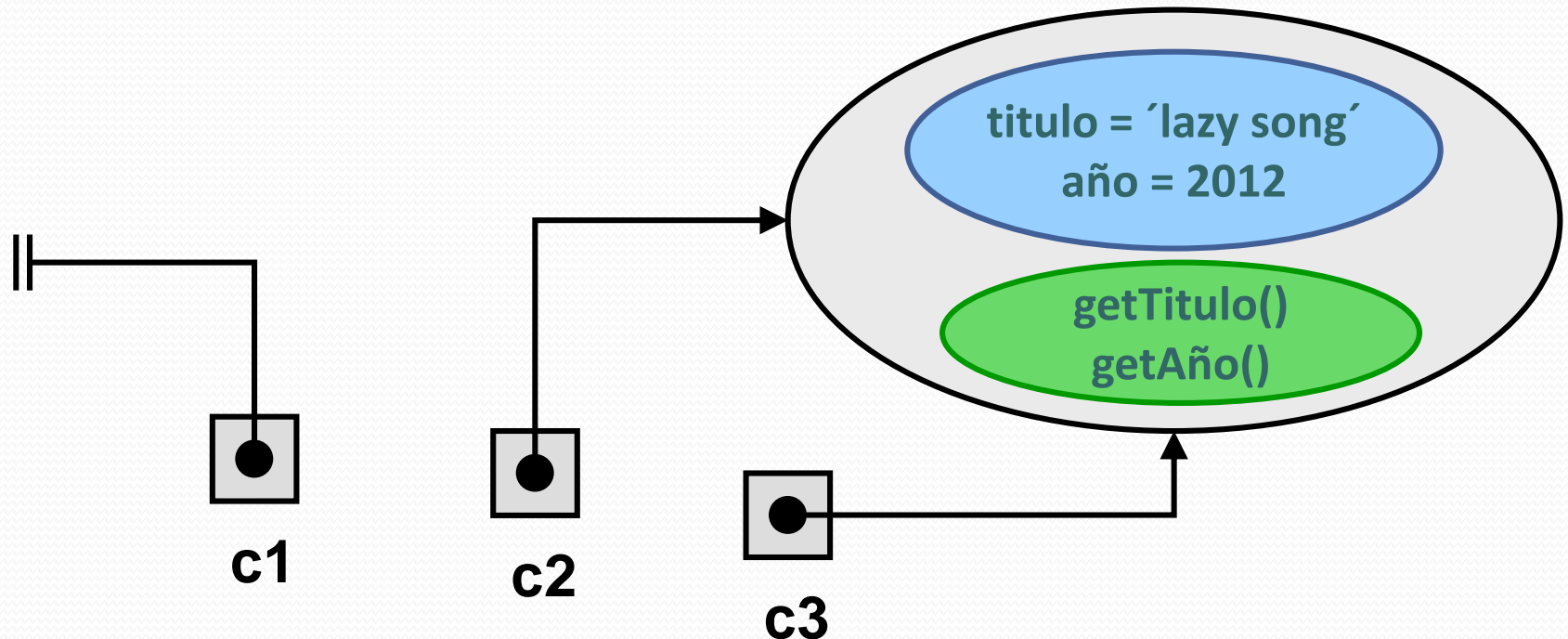
- En este tipo de casos lo deseable es que el método a despachar sea el asociado a la clase de la cual el objeto al que le es aplicada la operación es instancia directa
 - En el ejemplo anterior, *B*
- Siempre que es posible el despacho se realiza en tiempo de compilación denominándose despacho estático

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)

```
Cancion *c1 = null; // void  
Cancion *c2 = new Cancion('lazy song', 2012); // attached  
Cancion *c3 = *c2; // 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 ancestral 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)

```
Numero *x = new Racional(1,2);
```

TipoEstatico(t) = Numero

TipoDinamico(t) = Racional

```
bool esRacional;  
// el usuario ingresa en la consola 1 o 0  
cout << "Quiere un numero racional? ";  
cin >> esRacional;  
Numero *x;  
if (esRacional)  
    x = new Racional(1,3);  
else  
    x = new Entero(432);
```

¿Cuál es el tipo dinámico de x?

Despacho Dinámico

- Los lenguajes de programación orientados a objetos permiten que el tipo dinámico de un objeto difiera del tipo estático
- Cuando se realiza una invocación a una operación polimórfica (que está redefinida) sobre un objeto utilizando una referencia a él declarada como de una de sus clases ancestras puede no ser correcto realizar el despacho en tiempo de compilación

Despacho Dinámico (2)

- De realizarse el despacho en forma estática se utilizaría para ello la única información disponible de él en ese momento:
 - La basada en el tipo estático
- Por lo que se despacharía (eventualmente) el método equivocado:
 - En particular, cuando la operación invocada es abstracta en la clase del tipo estático no hay método que despachar

Despacho Dinámico (3)

- La operación `toFloat()` declarada en `Numero` es polimórfica porque es redefinida en `Entero` y en `Racional`
- Se está invocando a una operación polimórfica sobre un objeto (que será de clase `Entero` ó `Racional`) mediante una referencia declarada como de tipo `Numero` (clase ancestral de las anteriores)
- En esta invocación debería despacharse $MET_{Entero::toFloat()}$ ó $MET_{Racional::toFloat()}$

```
Numero *x;  
if (esRacional)  
    x = new Racional(1,3);  
else  
    x = new Entero(432);  
  
x->toFloat();
```

Despacho Dinámico (4)

- Para que en este tipo de casos el despacho sea realizado en forma correcta es necesario esperar a contar con la información del tipo real del objeto (tipo dinámico):
 - Eso se obtiene en tiempo de ejecución
- El *despacho dinámico* es la capacidad de aplicar un método basándose en la información dinámica del objeto y no en la información estática de la referencia a él

Despacho Dinámico (5)

```
Numero *x;  
if (esRacional)  
    x = new Racional(1,3);  
else  
    x = new Entero(432);  
  
x->toFloat();
```

- En tiempo de compilación: al pasar por la invocación el compilador NO despacha método alguno
- En tiempo de ejecución: al pasar por la invocación el ambiente de ejecución del lenguaje se ocupa de averiguar el tipo dinámico de *e* y despachar al método correcto, es decir a:
 $MET_{Racional::toFloat()}$ o $MET_{Entero::toFloat()}$

Despacho Dinámico (6)


- La decisión de qué tipo de despacho emplear para una operación puede estar preestablecida en el propio lenguaje o definida estáticamente en el código fuente
- En algunos lenguajes de programación el despacho es dinámico para cualquier operación (sea polimórfica o no)
- En otros lenguajes:
 - Las invocaciones a operaciones polimórficas son siempre despachadas dinámicamente
 - Las invocaciones a operaciones no polimórficas son siempre despachadas estáticamente

Despacho Dinámico (7)

```
Numero *arr[3];  
arr[0] = new Racional(1,2);  
arr[1] = new Entero(2);  
arr[2] = new Racional(1,4);
```

```
float total = 0.0;  
for(int i = 0; i < 3; i++){  
    total += (arr[i])->toFloat();  
}
```

```
cout << "El total es" << total;
```



Despacha dinámicamente al método correcto, devolviendo el valor correcto

Despacho Dinámico (8)

- Sólo se realiza si la operación está definida como virtual
- Si es así intenta despachar el método definido en el tipo dinámico
- En caso de que no haya un método allí, busca el primer método en la jerarquía hacia “arriba”

Despacho Dinámico (9)

```
class Natural: public Entero {  
    // no define float toFloat();  
};
```

```
Numero *arr[3];  
arr[0] = new Racional(1,2);  
arr[1] = new Entero(2);  
arr[2] = new Natural(3);
```

```
arr[0]->toFloat(); // Despacha el método de Racional  
arr[1]->toFloat(); // Despacha el método de Entero  
arr[2]->toFloat(); // Despacha el método de Entero
```

Algoritmo de despacho C++ (simplificado)

- El compilador busca la operación según el tipo estático.
- Se va directamente a la clase del tipo estático, al de su padre, luego al de su abuelo, y así sucesivamente hasta encontrar la operación que se invoca.
 - a) Si la operación encontrada **no es *virtual***
 - Se despacha el método de la clase que lo declaró (despacho estático)
 - b) Si la operación encontrada **es *virtual***
 - Se despacha el método dinámicamente según el tipo dinámico (despacho dinámico)
 - Se va directamente a la clase del tipo dinámico (en tiempo de ejecución) y si no hay un método para esa operación se busca en el padre, abuelo, etc. hasta encontrar una clase que defina un método para la operación.

Obs: Una operación **es *virtual*** si tiene el calificador *virtual* en la declaración o si alguna clase base directa o indirecta declara la misma operación con el calificador *virtual*.