



Polimorfismo

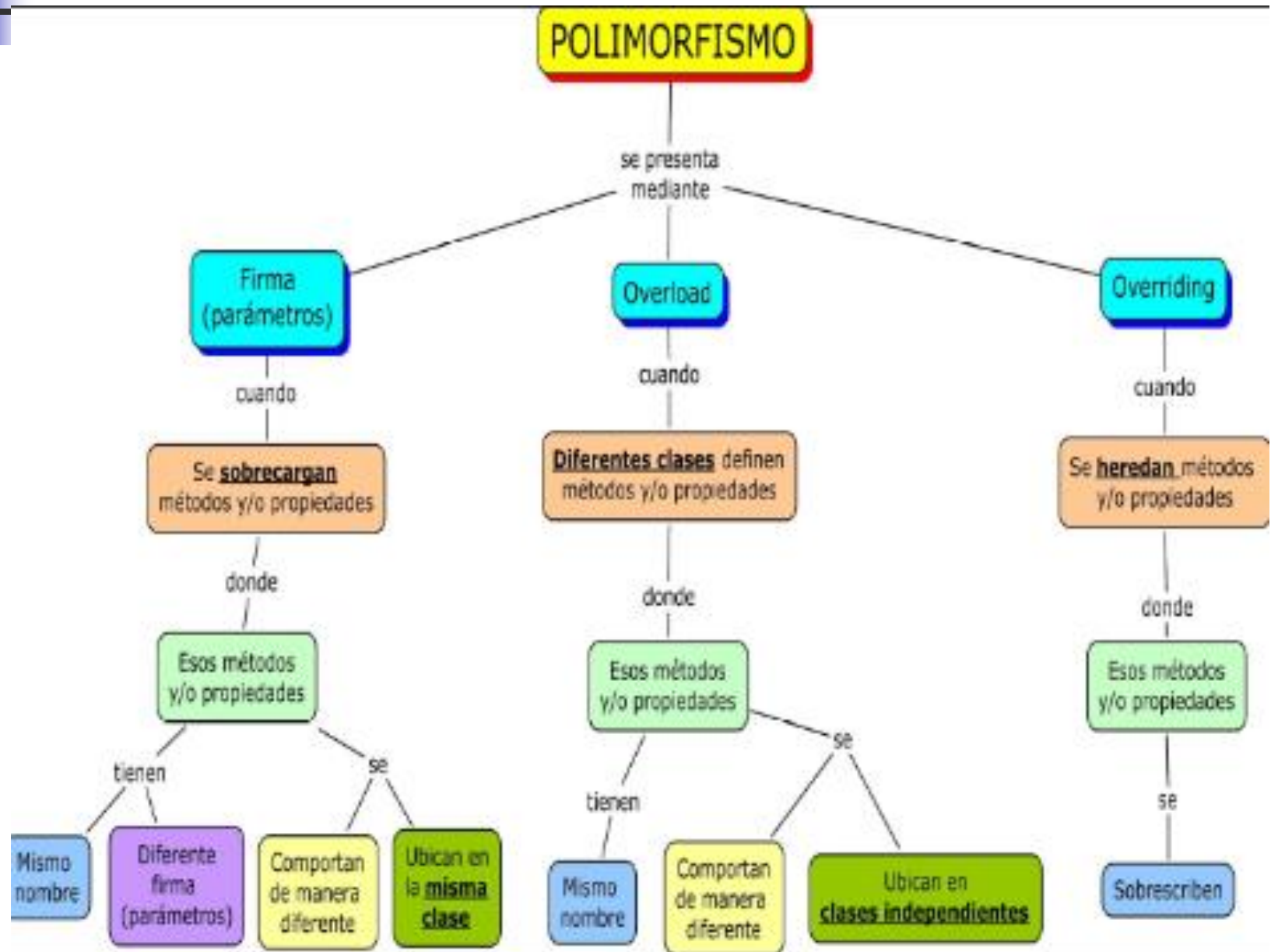
POO – FCyT UADER -



Contenido

- Encadenamiento Tardío.
 - Caso de Análisis: Cuadrado – Rectángulo.
- Tipificado.
- Caso de Análisis: Instrumentos.
 - Herencia.
 - Funciones Virtuales
 - Tabla de métodos virtuales (TMV).
 - Funciones Virtuales Puras. Clases abstractas.
 - Definición de Polimorfismo.
 - Pasaje de clases que poseen polimorfismo
 - Constructores y destructores.

Tipos de Polimorfismo



POLIMORFISMO

se clasifica en

Paramétrico

cuando

Una **misma** clase tiene

varios

Métodos y/o propiedades

donde

Mismo nombre

con

Diferente firma (parámetros)

dependiendo de

Elige automáticamente el método y/o propiedad a aplicar

De sobrecarga

cuando

Diferentes clases

tienen

Métodos y/o propiedades

donde

Mismo nombre

con

dependiendo de

El objeto en el que se trabaja

Elige automáticamente el método y/o propiedad a aplicar

De subtipo

cuando

Las clases **heredan** métodos y/o propiedades

donde

Esos métodos y/o propiedades

se

Sobrescriben



Encadenamiento Tardío

```
class Cuadrado {  
protected:  
    int lado, area;  
public:  
    void Ingresar(int l){lado = l;};  
    void CalcularArea(){area = lado * lado;};  
    int Mostrar(){ CalcularArea(); return area; }  
};
```

```
class Rectangulo : public Cuadrado {  
private:  
    int altura;  
public:  
    void Ingresar(int l, int a){lado = l; altura = a;};  
    void CalcularArea(){area = lado * lado;};  
};
```



Encadenamiento Tardío

```
int main()
{Cuadrado Cuad;
  cout << "Ingrese el lado : ";
  int lado;
  cin >> lado;
  Cuad.Ingresar(lado);
  Cuad.CalcularArea();
  cout << "El area del cuadrado es "
        << Cuad.Mostrar() << endl;

  Rectangulo Rec;
  cout << "Ingrese el lado y altura: ";
  int altura;
  cin >> lado >> altura;
  Rec.Ingresar(lado, altura);
  Rec.CalcularArea();
  cout << "El area del rectangulo es "
        << Rec.Mostrar() << endl; return 0;}
```

Ingrese el lado: 1

El area del cuadrado es 1



Encadenamiento Tardío

```
int main()
{Cuadrado Cuad;
 cout << "Ingrese el lado : ";
 int lado;
 cin >> lado;
 Cuad.Ingresar(lado);
 Cuad.CalcularArea();
 cout << "El area del cuadrado es "
      << Cuad.Mostrar() << endl;
```

```
Rectangulo Rec;
 cout << "Ingrese el lado y altura: ";
 int altura;
 cin >> lado >> altura;
 Rec.Ingresar(lado, altura);
 Rec.CalcularArea();
 cout << "El area del rectangulo es "
      << Rec.Mostrar() << endl; return 0;}
```

Ingrese el lado: 1

El area del cuadrado es 1

Ingrese el lado y la altura: 1 2

El area del rectangulo es 1

¿POR QUÉ?



Encadenamiento Tardío

- La función **Mostrar()** de la clase **cuadrado** posee una llamada al método **CalcularArea()**.
- A su vez, al invocar **Mostrar()** de la clase **rectángulo**, el mismo se encuentra encadenado tempranamente al método **CalcularArea()** de la clase ancestro.

NECESIDAD DE ENCADENAMIENTO TARDÍO!!!



Miembros virtuales

- Los miembros *virtuales* son aquellos que están definidos en la clase madre, **y se redefinirán en las clases descendientes**



Encadenamiento Tardío

```
class Cuadrado {  
protected:  
    int lado, area;  
public:  
    void Ingresar(int l){lado = l;};  
    virtual void CalcularArea(){area = lado * lado;};  
    int Mostrar() {CalcularArea(); return area;}  
};
```

```
class Rectangulo : public Cuadrado {  
private:  
    int altura;  
public:  
    void Ingresar(int l, int a){lado = l; altura = a;};  
    void CalcularArea(){area = lado * altura;};  
};
```

Encadenamiento Tardío

```
int main()
{Cuadrado Cuad;
 cout << "Ingrese el lado : ";
 int lado;
 cin >> lado;
 Cuad.Ingresar(lado);
 Cuad.CalcularArea();
 cout << "El area del cuadrado es "
      << Cuad.Mostrar() << endl;

Rectangulo Rec;
 cout << "Ingrese el lado y altura: ";
 int altura;
 cin >> lado >> altura;
 Rec.Ingresar(lado, altura);
 Rec.CalcularArea();
 cout << "El area del rectangulo es "
      << Rec.Mostrar() << endl; return 0;}
```

Ingrese el lado: 1

El area del cuadrado es 1

Ingrese el lado v la altura: 1 2

El area del rectangulo es 2

Fin



Tipificado

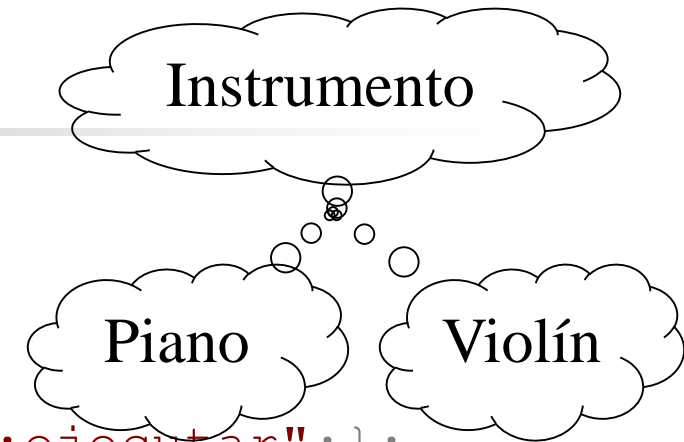
- Dada dos instrumentos, un Piano y un Violín.
- ¿Cómo hacemos para decidir en tiempo de ejecución que tipo de instrumento quiero que toque una nota?

Tipificado

```
class Instrumento {
public:
    void ejecutar() {cout << "Inst::ejecutar";};
    string nombre() {return "Instrumento";}};

class Piano : public Instrumento {
public:
    void ejecutar() {cout << "Piano::ejecutar";};
    string nombre() {return "Piano";}};

class Violin : public Instrumento {
public:
    void ejecutar() {cout << "Violin::ejecutar";}
    string nombre() {return "Violin";}
};
```





Tipificado

```
switch(opcion){
```

caso 1:

- Creo un Piano
- Lo hago Tocar

caso 2:

- Creo un Violín
- Lo hago Tocar

```
}
```



Tipificado

```
switch(opcion){
```

caso 1:

- Creo un Piano
- Lo hago Tocar

caso 2:

- Creo un Violín
- Lo hago Tocar

```
}
```

```
switch(opcion){
```

caso 1:

- Creo un Piano

caso 2:

- Creo un Violín

```
}
```

Hago tocar el

INSTRUMENTO



Herencia

```
int main()
{cout << "Caso Violin\n";
  Violin v, *pv=&v;
  pv->ejecutar();

return 0;}
```

Caso del Violin
Violin::ejecutar



Herencia

```
int main()
{cout << "Caso Violin\n";
  Violin v, *pv=&v;
  pv->ejecutar();

  cout <<"Caso del Instrumento\n";
  Instrumento i,*pi=&i;
  pi->ejecutar();

return 0; }
```

Caso del Violin
Violin::ejecutar

Caso del Instrumento
Instrumento::ejecutar



Herencia

```
int main()
{cout << "Caso Violin\n";
  Violin v, *pv=&v;
  pv->ejecutar();

  cout <<"Caso del Instrumento\n";
  Instrumento i,*pi=&i;
  pi->ejecutar();

  cout <<"Caso Mixto\n";
  Instrumento *pi_violin=&v;
  pi_violin->ejecutar();

  return 0; }
```

Caso del Violin
Violin::ejecutar

Caso del Instrumento
Instrumento::ejecutar

Caso Mixto
Instrumento::ejecutar

**pi_violin invoca los miembros
correspondientes a su
declaración**



Herencia

- **Upcasting:** es posible asignar a un puntero de tipo *clase_madre* la dirección de memoria de un objeto de tipo *clase_hija*.
- Sin embargo, mediante este puntero sólo podemos invocar a los métodos definidos en la clase madre.



Herencia

```
int main()
{cout << "Caso Violin\n";
  Violin v, *pv=&v;
  pv->ejecutar();

  cout <<"Caso del Instrumento\n";
  Instrumento i,*pi=&i;
  pi->ejecutar();

  cout <<"Caso Mixto\n";
  Instrumento *pi_violin=&v;
  pi_violin->ejecutar();

  cout <<"Caso contra Mixto\n";
  Violin *pv_inst = pi;
  pv_inst->ejecutar();

  return 0;}
```

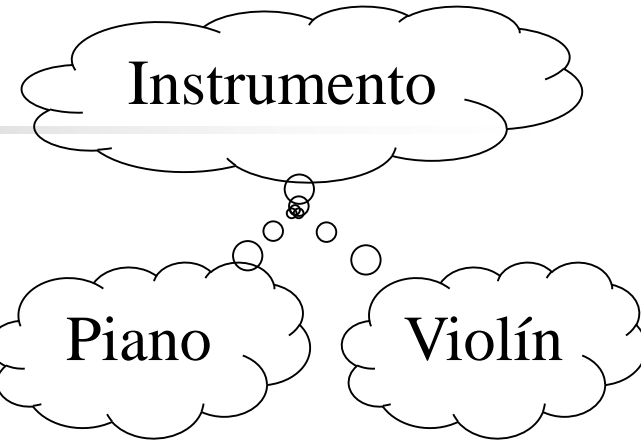
Caso del Violin
Violin::ejecutar

Caso del Instrumento
Instrumento::ejecutar

Caso Mixto
Instrumento::ejecutar

**ERROR: NO PUEDE
REALIZARSE LA
CONVERSIÓN!!!!**

Funciones Virtuales



```
class Instrumento {
public:
    virtual void ejecutar() {cout << "Inst::ejecutar";};
    virtual string nombre() {return "Instrumento";}};

class Piano : public Instrumento {
public:
    void ejecutar() {cout << "Piano::ejecutar";};
    string nombre() {return "Piano";};
};

class Violin : public Instrumento {
public:
    void ejecutar() {cout << "Violin::ejecutar";}
    string nombre() {return "Violin";}
};
```



Polimorfismo

```
int main()
{cout << "Caso Violin\n";
  Violin v, *pv=&v;
  pv->ejecutar();

  cout <<"Caso del Instrumento\n";
  Instrumento i,*pi=&i;
  pi->ejecutar();

  cout <<"Caso Mixto\n";
  Instrumento *pi_violin=&v;
  pi_violin->ejecutar();

return 0;}
```

Caso del Violin
Violin::ejecutar

Caso del Instrumento
Instrumento::ejecutar

Caso Mixto
Violin::ejecutar



Tabla de Métodos Virtuales

- Para la administración de los métodos virtuales, el compilador crea en ***tiempo de ejecución*** una TMV.
- Ahora creando los objetos en forma dinámica, resolvemos el problema originalmente planteado.



Polimorfismo

```
switch(opcion){
```

caso 1:

- Creo un Piano
- Lo hago Tocar

caso 2:

- Creo un Violín
- Lo hago Tocar

```
}
```

```
switch(opcion){
```

caso 1:

- Creo un Piano

caso 2:

- Creo un Violín

```
}
```

Hago tocar el

INSTRUMENTO



Objetos dinámicos

- Para referirnos a un objeto dinámico utilizamos un *puntero* a su instancia

```
Piano * p;
```

```
...
```

```
p->ejecutar();
```

```
cout << p->nombre() << endl;
```

```
...
```



Objetos dinámicos

- Para crear un objeto en tiempo de ejecución es necesario utilizar el operador *new*

```
Piano * p;  
p=new Piano;  
p->ejecutar();  
cout << p->nombre() << endl;  
...
```



Objetos dinámicos

- Para destruir un objeto dinámico utilizamos el operador *delete*

```
Piano * p;  
p=new Piano;  
p->ejecutar();  
cout << p->nombre() << endl;  
delete p;
```



Upcasting y polimorfismo

- Ahora podemos asignar una instancia de la clase hija a una variable de la clase madre
- *upcasting*

```
Instrumento * p;
```

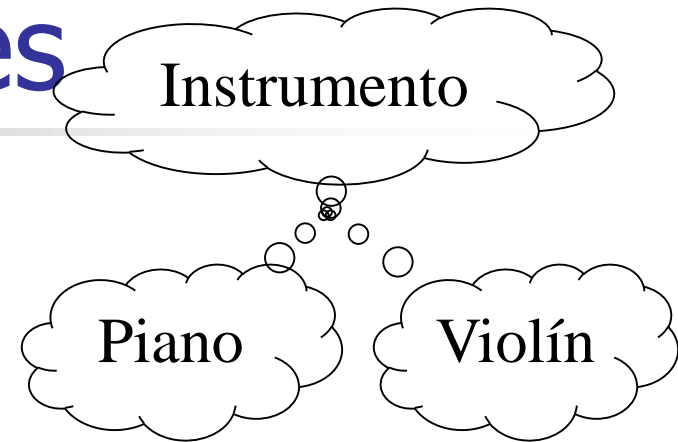
```
p=new Piano;
```

```
p->ejecutar();
```

```
cout << p->nombre() << endl;
```

```
delete p;
```

Funciones Virtuales Puras



```
class Instrumento {
public:
    virtual void ejecutar() = 0;
    virtual string nombre() = 0;
};

class Piano : public Instrumento {
public:
    void ejecutar() {cout << "Piano::ejecutar";};
    string nombre() {return "Piano";};
};

class Violin : public Instrumento {
public:
    void ejecutar() {cout << "Violin::ejecutar";}
    string nombre() {return "Violin";}
};
```



Clases abstractas

- Las clases abstractas son aquellas en las que al menos uno de sus miembros es virtual puro
- Cuando todos los miembros son virtuales puros se habla de clases abstractas *genuinas*
- Las clases abstractas *no pueden instanciarse...* pues tampoco tiene sentido hacerlo...
- *Instrumento* es ahora una clase abstracta genuina



Polimorfismo: definición 1

Dos o más objetos de una *misma clase* se comportan de formas diferentes.




Polimorfismo: definición 2

Una variable puede tomar diferentes tipos, dependiendo de la necesidad en tiempo de ejecución.



Polimorfismo: definición 3

Un puntero a un objeto de la superclase puede ser instanciado como *cualquiera* de sus clases descendientes.



Pasaje de clases polimórficas como parámetro de funciones

```
void EjecutaNotas (Instrumento& i) {  
    i.ejecutar (DO) ;  
    i.ejecutar (RE) ;  
    i.ejecutar (SOL) ;  
} } ;
```

Podemos independizarnos del
instrumento para que sea utilizado
para colaborar en otras
responsabilidades



Constructores y destructores

```
class Instrumento {  
public:  
    Instrumento()  
        {cout << "Constructor de Instrumento" << endl;}  
    virtual void ejecutar(nota) =0;  
    virtual string nombre() =0;  
    virtual ~Instrumento()  
        {cout << "Destructor de Instrumento" << endl;}  
};
```

- Los constructores no pueden ser virtuales.
- Los destructores pueden ser virtuales

¿POR QUÉ?



Constructores y destructores

Supongamos:

```
int main() {  
    Instrumento *I;  
    I= new Piano;  
    ...  
}
```

```
delete I;  
return 0; }
```

■ Siempre se conoce al objeto que se desea crear.

■ Sin embargo, no se sabe cual es el objeto que se destruye dado que estamos referenciado un ancestro.



¿Y que pasa si...?

```
class Violin : public Instrumento {  
    public:  
        void ejecutar();  
        string nombre();  
        void CambiarCuerdas()  
        { cout << "Cambio de cuerdas" << endl; }  
};  
  
int main()  
{Instrumento *i = new Violin;  
  i->CambiarCuerdas();  
  
return 0;  
}
```



¿Y que pasa si...?

```
class Violin : public Instrumento {  
public:  
    void ejecutar();  
    string nombre();  
    void CambiarCuerdas()  
    { cout << "Cambio de cuerdas" << endl; }  
};
```

```
int main()  
{Instrumento *i = new Violin;  
  i->CambiarCuerdas();  
  
return 0;  
}
```

- No se puede acceder dado que CambiarCuerdas() no es un miembro de Instrumento



Luego... downcasting

Cambio de cuerdas

```
int main()
{Instrumento *i = new Violin;

...

Violin *v;

v=(Violin*) i;

v->CambiarCuerdas();

return 0;
}
```



Polimorfismo y contenedoras

- ¿Cómo podríamos simular una orquesta formada por una *lista* de instrumentos?
- Necesitaremos una lista de *punteros* a Instrumento

```
vector<Instrumento*> orquesta;  
//new...
```

- ¿Cuál es la ventaja de que sean polimórficos?

```
for (i=0; i<orquesta.size(); i++)  
    cout << orquesta[i]->ejecutar();  
//...delete
```




Veamos el ejemplo completo
