6. Herencia y polimorfismo

Julio Vega

julio.vega@urjc.es

Contenidos

- Una de las principales virtudes de la POO es la herencia.
- Es una forma de reutilización de software.
- Para crear la nueva clase, se *heredan* datos/comportamientos de otra.
  - Clase base = clase existente VS. clase derivada = clase nueva.
  - Métodos de la derivada no acceden a atributos `private` de la base.
  - Y las funciones `friend` no se heredan (no eran de la clase...).
- La clase derivada contiene lo del padre + funcionalidad adicional.
  - Ya depende de si es herencia `public`, `protected` o `private`.
  - De momento, nos centraremos en la herencia de tipo `public`.
    - Todo objeto de clase derivada es objeto de la clase base.
- C++ soporta herencia múltiple (heredar de varias).
  - No así en Java, con buen criterio; evita complejidad y errores.
  - Para ello, en Java, manejan el concepto de `interfaces`.

- Las relaciones de herencia forman estructuras jerárquicas.
  - Una clase se convierte en base a otras, derivada de otras, o en ambas.
- E.g. `Vehiculo` (clase base) y `Moto` (clase derivada):
  - Todas las motos son vehículos, pero no todos los vehículos son motos.
- Para evitar confusión, léase: una moto **es un** vehículo $\Longrightarrow$ ¡herencia!
  - VS. *tiene un* $\Longrightarrow$ composición. E.g. una moto *tiene un* manillar.
- Otro ejemplo, con la clase `PersonaUniversidad`.
  - Definición clase: `class Profesor : public PersonaUniversidad`
  - La relación de herencia, en UML, se representa por esa flecha *hueca*.
  - La lectura siempre en el sentido de la flecha.

Figura: Un profesor *es una* persona/figura de la universidad

Figura: Una persona de la comunidad universitaria *es una* persona

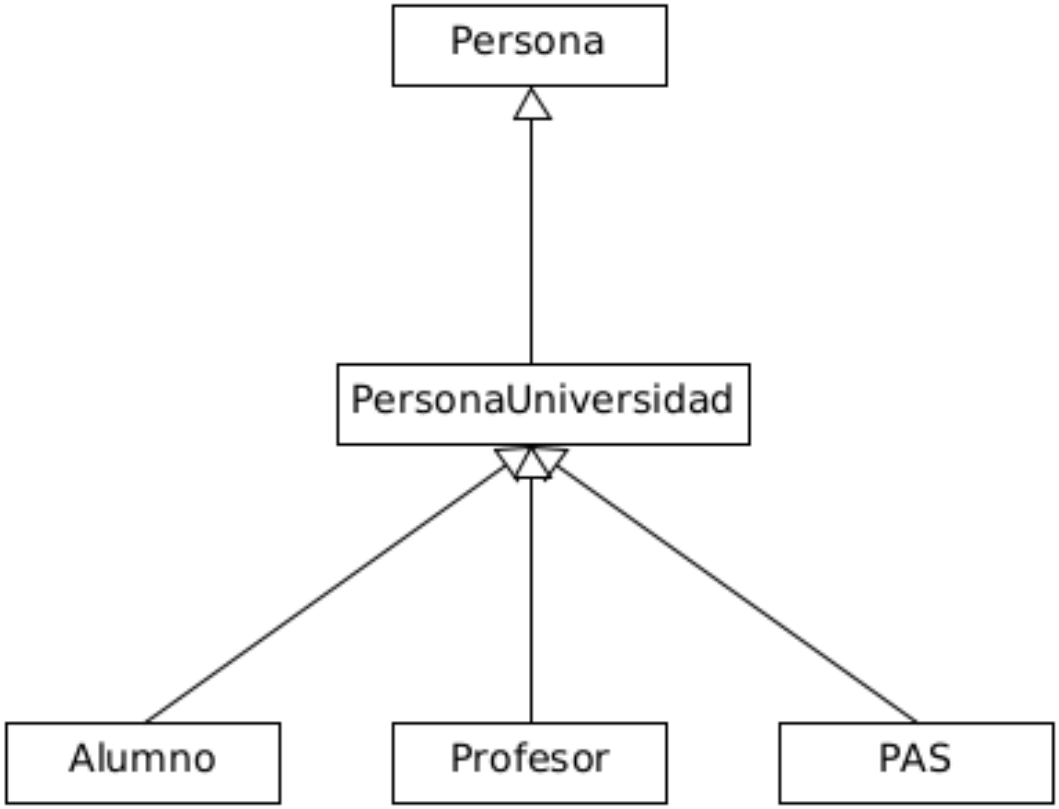

Figura: Alumno y PAS *son* personas de la universidad

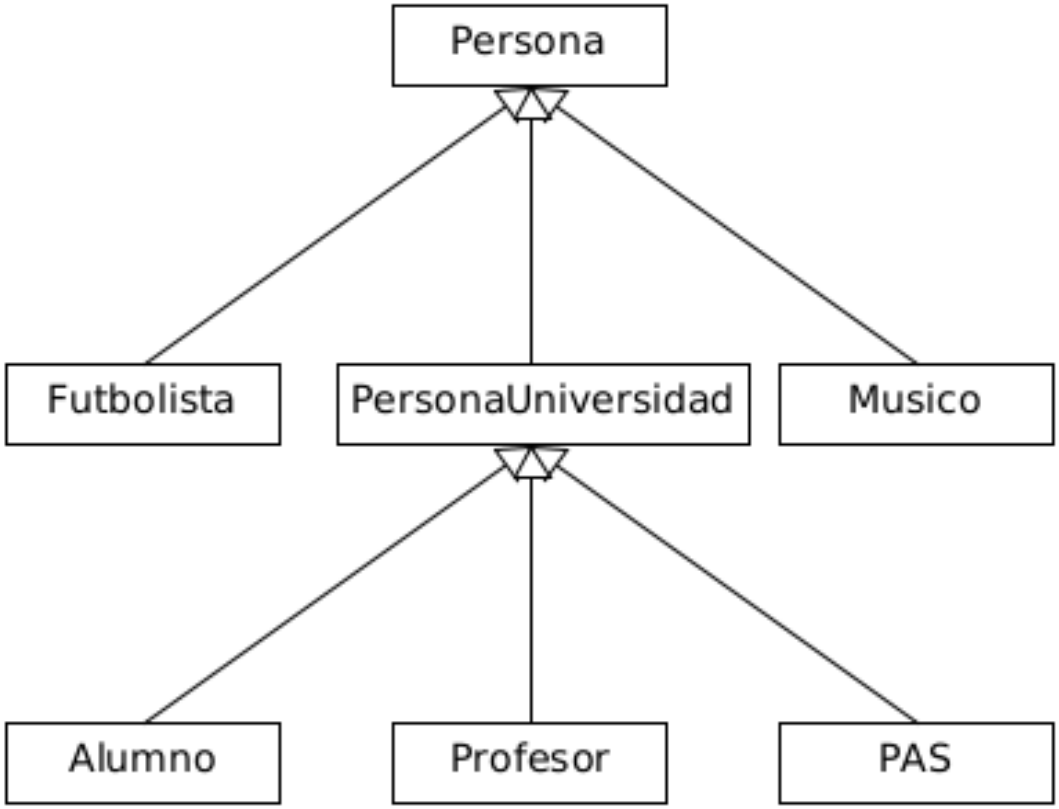

Figura: Un futbolista o un músico también *son* personas

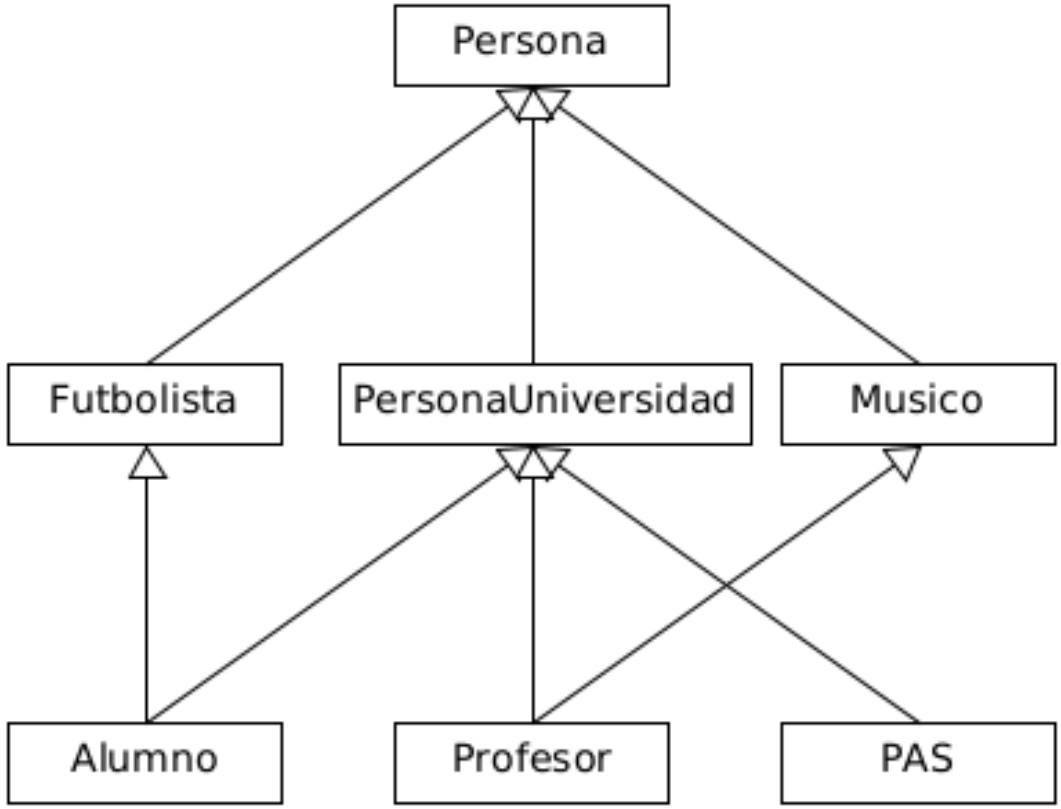

Figura: Un alumno *es*, además, futbolista. Un profesor *es*, además, músico

- Miembros `public` de clase base son accesibles desde esta.
- También desde las funciones `friend` de esta clase base.
  - Y desde cualquier parte donde haya un manejador (objeto, puntero).
- Miembros `private` de clase base solo accesibles desde dentro.
  - Y también desde las funciones `friend` de esta clase base.
- Miembros `protected` son como `public` para las clases derivadas.
  - Se accede desde dentro, mediante `friend`, e igual para las derivadas.
  - Si derivada redefine método, accede a este con `nombreBase::metodo`

```
class CommissionProgrammer {
public:
  CommissionProgrammer (const string &, const string &, double =
      0.0, double = 0.0);
  void setName (const string &);
  string getName () const;
  void setSSN (const string &);
  string getSSN () const;
  void setNumberOfProjects (double);
  double getNumberOfProjects () const;
  void setCommissionRate (double);
  double getCommissionRate () const;
  double earnings() const; // calculate earnings
  void print() const; // print CommissionProgrammer object
private:
  string name;
  string ssn; // Social Security Number
  double numberOfProjects;
  double commissionRate;
}; // end class CommissionProgrammer
```

```
CommissionProgrammer::CommissionProgrammer (const string &name,
    const string &ssn,
  double projects, double rate) : name (name), ssn (ssn) {
  setNumberOfProjects (projects); // validate/store no. projects
  setCommissionRate (rate); // validate and store commission rate
}

void CommissionProgrammer::setName (const string &name) {
  this->name = name;
}
string CommissionProgrammer::getName () const {
  return name;
}

void CommissionProgrammer::setSSN (const string &ssn) {
  this->ssn = ssn;
}
string CommissionProgrammer::getSSN () const {
  return ssn;
}
```

```
void CommissionProgrammer::setNumberOfProjects (double projects)
    {
  if (projects >= 0.0)
    numberOfProjects = projects;
  else
    throw invalid_argument ("No. of projects must be >= 0.0" );
}

double CommissionProgrammer::getNumberOfProjects () const {
  return numberOfProjects;
}

void CommissionProgrammer::setCommissionRate (double rate) {
  if (rate > 0.0 && rate < 100.0)
    commissionRate = rate;
  else
    throw invalid_argument ("Commission rate must be > 0.0 and <
        100.0");
}
```

```
double CommissionProgrammer::getCommissionRate () const {
   return commissionRate;
}

double CommissionProgrammer::earnings() const {
  return getCommissionRate() * getNumberOfProjects();
}

void CommissionProgrammer::print() const {
  cout << "commission programmer: " << getName()
    << "\nsocial security number: " << getSSN()
    << "\nnumber of projects: " << getNumberOfProjects()
    << "\ncommission rate: " << getCommissionRate();
}
```

```
class SalariedProgrammer : public CommissionProgrammer {
public:
  SalariedProgrammer (const std::string &, const std::string &,
    double = 0.0, double = 0.0, double = 0.0);

  void setSalary (double);
  double getSalary() const;

  double earnings() const; // calculate earnings
  void print() const; // print SalariedProgrammer object
private:
  double salary; // base salary
}; // end class SalariedProgrammer
```

```
SalariedProgrammer::SalariedProgrammer(const string &name, const
    string &ssn,
   double projects, double rate, double salary)
   // explicitly call base-class constructor
   : CommissionProgrammer (name, ssn, projects, rate) {
   setSalary (salary); // validate and store base salary
}

void SalariedProgrammer::setSalary (double salary) {
  if (salary >= 0.0)
    this->salary = salary;
  else
    throw invalid_argument ("Salary must be >= 0.0");
}

double SalariedProgrammer::getSalary () const {
  return salary;
}
```

```
double SalariedProgrammer::earnings () const {
  // call base-class earnings function:
  return getSalary() + CommissionProgrammer::earnings();
}

void SalariedProgrammer::print () const {
  cout << "salaried ";

  // call base-class print function:
  CommissionProgrammer::print();

  cout << "\nbase salary: " << getSalary();
}
```

- Podemos observar la limpieza y el ahorro de código al heredar.
  - Si en vez de ello duplicáramos código, los errores se multiplicarían.
- Con herencia, atrib. y métodos comunes se declaran solo en la base.
  - Si hubiera que modificarlos, se modifican solo una vez, en la base.
  - Y la clase derivada automáticamente se beneficia de esos cambios.
- En el ejemplo, la derivada solo ha de añadir la cualidad de asalariado.
  - Este es el ejemplo típico de cuándo y cómo usar herencia.
- La herencia, en la clase derivada, se representa por `: nombreBase`
- Se usa `this` para evitar confusión entre parámetros y atributos.
  - E.g. `this->salary = salary;`

- En la clase derivada se incluye el `.h` de la clase base debido a que:
  - La clase derivada utiliza el nombre de la clase base.
  - El compilador usa la def. de clase (.h) para saber el tamaño del objeto.
    - Con herencia, tal tamaño depende de los atrib. de su clase y de la base.
- Cuando se redefine una función, esta puede llamar a la función base.
  - `SalariedProgr::print()` $\Longrightarrow$ `CommissionProgr::print()`
  - La llamada de func. derivada a func. base es `claseBase::funcion()`.
    - Si no, ¡la función derivada se estaría llamando a sí misma en bucle!

- La derivada no hereda const., dest. ni op. asignación sobrecargados.
  - Pero todo ello de la derivada puede llamar a sus homónimos de la base.
- Al crear ob. de clase derivada se genera cadena de llamadas a const.
  - El constructor de la derivada, antes de sus tareas, invoca al de la base.
    - Explícita/ (ejemplo), o implícita/ (se llama al const. pred. de la base).
  - Si base deriva de otra clase, se invoca a const. hacia arriba sucesiva/.
  - El último const. llamado será el de la *cima* de la jerarquía.
    - Y el cuerpo del const. de la derivada termina de ejecutarse el último.
- En ejemplo, el const. de la clase base inicializa atrib. de la base.
  - Una mejora a esto sería validar tanto el nombre como el NSS.
- Al destruir objeto de clase derivada, se llama a su destructor.
  - Este realiza su tarea y después invoca al destr. de la base.
  - La cascada de llamadas de destr. se ejecutan inversa/ a los constr.

- Al compilar, hemos de tener en cuenta la jerarquía de la herencia.
- Por comodidad, lo habitual es implementar un archivo `Makefile`:

```
CC = g++              # specified the compiler g++
CFLAGS = -Wall -g # shows all Warnings and adds debugging symbols
.RECIPEPREFIX = > # uses '>' as prefix of my recipes since all...
# actions of rules are identified by tabs and I do not use them

main: main.o CommissionProgrammer.o SalariedProgrammer.o
>$(CC) $(CFLAGS) -o main main.o CommissionProgrammer.o
    SalariedProgrammer.o
main.o: main.cpp CommissionProgrammer.h SalariedProgrammer.h
>$(CC) $(CFLAGS) -c main.cpp
CommissionProgrammer.o: CommissionProgrammer.h
SalariedProgrammer.o: SalariedProgrammer.h CommissionProgrammer.h
```

- El polimorfismo permite programar en general en lugar de específica/.
- El polimorfismo es un concepto que va muy unido a la herencia.
  - Permite procesar obj. de clases que formen parte de una jerarquía...
  - ...como si todos fueran objetos de la clase base de dicha jerarquía.
- E.g.: clase base `Robot` y derivadas: `Nao`, `Pepper` y `Whiz`.
  - Son los tres robots de la compañía *SoftBank Robotics*.
  - Tenemos una colección (vector) de punteros a varios de estos robots.
  - Para moverlos hacia adelante hay una función `moveForward()`.
  - Programa *los mueve* iterativa/ llamando siempre a la misma función.
  - Pero cada objeto sabe cómo *moverse* según su morfología.
  - ¡Esto es el polimorfismo: 1 función $=$ X formas de resultados!
- El polimorfismo facilita la reutilización de código.
  - Cuando empresa cree nuevo robot, solo necesita crear la nueva clase.
  - Pero el programa podrá seguir manteniendo su estructuras.
    - E.g.: el vector podrá albergar al nuevo robot y todo funcionará igual.

- **En tiempo de compilación**. Lo que habíamos tratado hasta ahora.
  - Sobrecargando las funciones o los operadores.
  - Recuerda: mismo nombre o símbolo y distintos comportamientos.
    - Para ello, debían tener diferentes $n.^o$ y/o tipos de parámetros.
    - O redefiniendo comportamiento (en herencia): e.g. `print` Sec. 17.
- **En tiempo de ejecución**. Este tipo solo se puede dar en herencia.
  - Sobrescribiendo las funciones de tipo `virtual` en las clases derivadas.
    - Es la idea que se introducía con el ejemplo de los robots.
  - Ahora veremos ejemplo con funciones `virtual` y no virtual.

Clase base con dos funciones: virtual y no virtual

```
class Base {
public:
  virtual void print () {
    cout << "print base class" << endl;
  }

  void show () {
    cout << "show base class" << endl;
  }
};
```

Clase derivada sobrecargando las funciones

```
class Derived : public Base {
public:
  void print () { // print () is already virtual function in
      derived class. We could also declared as virtual void
      print () explicitly
    cout << "print derived class" << endl;
  }

  void show () {
    cout << "show derived class" << endl;
  }
};
```

Uso del polimorfismo con funciones `virtual` vs. *no-virtual*

```
int main() {
  Base *bptr;
  Derived d;
  bptr = &d;

  // virtual function, binded at runtime (runtime polymorphism)
  bptr->print();

  // non-virtual function, binded at compile time
  bptr->show();

  return 0;
}
// ******** output:
// print derived class
// show base class
```

- Asignado direc. de obj. de clase derivada a puntero de la clase base.
  - `Base *bptr; Derived d; bptr = &d;`
  - C++ lo permite: obj. de clase derivada *es un* obj. de la clase base.
- Lo contrario, con toda lógica, genera error de compilación en C++.
  - Por ello, tpc se pueden invocar métodos derivados desde obj. base.

- Las funciones virtuales son las que hacen brillar al polimorfismo.
  - Permiten que el programa determine dinámica/ (al usar puntero `->`).
  - ...de qué clase derivada se va a ejecutar una determinada función.
    - Proceso denominado **vinculación dinámica o en tiempo de ejecución**.
  - También se pueden invocar usando el operador punto `.`
    - La invocación se resuelve en t. de compilación: **vinculación estática**.
    - En el ejemplo, sería: `d.print();`
- Si función `virtual` $\Longrightarrow$ lo será siempre (hacia abajo de jerarquía).
  - Aunque, para evitar confusión, al heredar, se puede poner `virtual`.
- Diferencia entre **redefinir** y **sobrescribir** al heredar un método:
  - Si el método es no virtual, podemos redefinirlo.
  - Si el método es virtual, podemos sobrescribirlo.

- Al definir una clase, se supone que se van a crear objetos de ese tipo.
- Pero a veces es útil definir clases de las que no se van a crear objetos.
  - A estas clases se las conoce como **clases abstractas**.
- Lo habitual es crear *clases base abstractas*, que estarán incompletas.
  - Y son las clases derivadas las que deben definir *lo que falta*.
- ¿Qué sentido tienen las clases abstractas?
  - Establecer un agrupamiento para aquellas clases que sean *hermanas*.
- Vemos ejemplo claro en la clasificación jerarquizada de los seres vivos.
  - Por ejemplo, todos los mamíferos tienen características comunes.
    - Pero no veremos a un mamífero por la calle $\Longrightarrow$ concepto abstracto.
    - Veremos alguno de sus tipos derivados: persona, perro, etc.
  - `Mamifero` = clase base abstracta y `Persona`/`Perro` = derivadas.

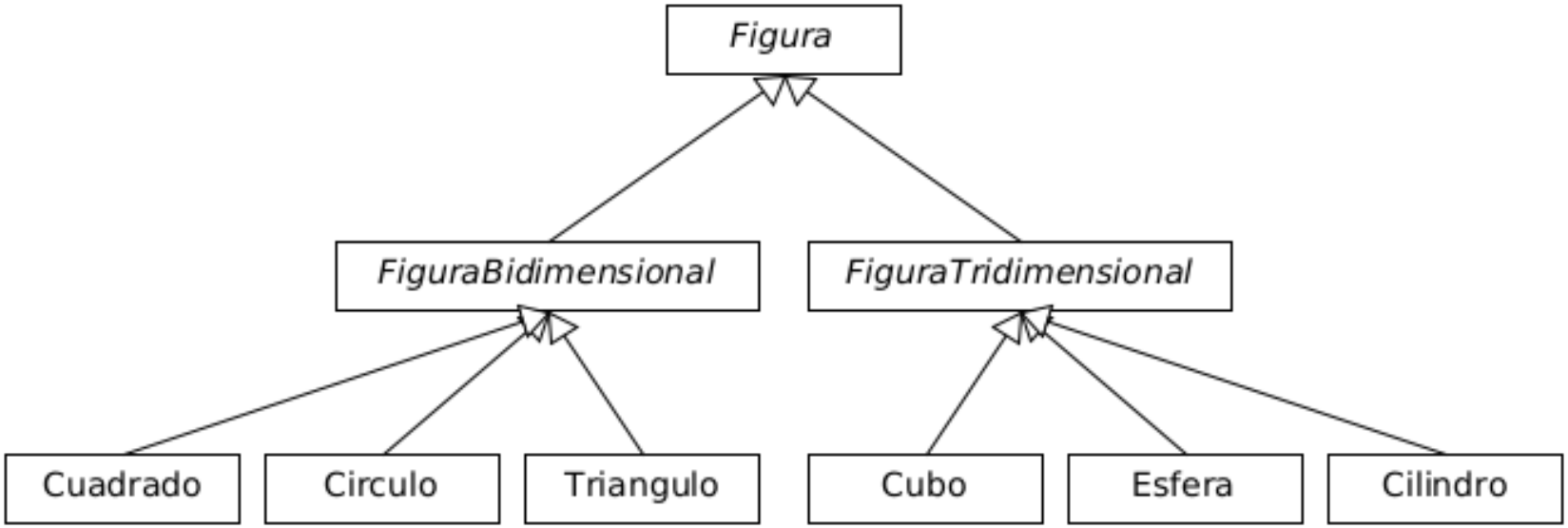

Figura: Abstractas: `Figura`, `FiguraBidimensional` y `FiguraTridimensional`

- Nótese que las clases abstractas se escriben en cursiva en UML.
- No podemos dibujar figura bidimensional $\Longrightarrow$ concepto abstracto.
  - Podríamos dibujar un cuadrado, o un círculo o un triángulo.

- Clase abstracta se da cuando 1 o más funciones virtuales son puras.
- Función virtual pura: `virtual void dibujar() const = 0;`
  - El `=0` se conoce como un **especificador puro**.
- Las funciones virtuales puras no proporcionan implementación.
  - Cada clase derivada concreta **debe** sobrescribirlas todas.
  - Esta es la diferencia entre virtual *pura* y *no pura*.
    - Recuerda que en función virtual no pura *se puede* sobrescribir.
- Instanciar un objeto de clase abstracta produce error de compilación.
  - Pero la clase base abstracta sí se puede usar para declarar punteros.
    - Recuerda que esta es la magia del polimorfismo.

- Hasta hemos visto los destructores *normales* (no virtuales).
- En polimorfismo, al destruir obj. derivado con `delete basePtr;`
  - Se genera comportamiento indefinido según el estándar de C++.
- Solución: implementar un destructor virtual en la clase base.
  - Así, al hacer `delete basePtr;` se invoca al destr. derivado concreto.
  - Y después, recuerda, se ejecuta el destructor de la base automática/.

6. Herencia y polimorfismo

Julio Vega

julio.vega@urjc.es