Herencia

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-02-10

Herencia

Problema

Trabajando en un banco, se te pide que desarrolles una función que calcule el total del valor de una serie de `n` `cuentas` de un cliente tras un número `t` de meses en el futuro:

```
double total(Cuenta* cuentas[], int n, int t)
```

Todas las cuentas tienen un `capital`, número real que representa una magnitud monetaria en el mes inicial (`t=0`).
Hay dos tipos de cuentas, cuyo valor a lo largo del tiempo se calcula de forma diferente:

- **Cuenta corriente:** Tienen un tipo de interés nominal mensual.
- **Plazo fijo:** Vienen determinados por un plazo (número de meses) y un tipo de interés a vencimiento. Antes del plazo, da interés 0 %.

```
1  class Cuenta {
2    int    categoria; // 0->Cuenta corriente, 1->Plazo fijo
3    double capital,interes; //Comunes
4    int    plazo; // Solo para plazo
5    double valor(int t) const {
6     switch(categoria) {
7       case 0:
8          return capital*pow(1+(interes/100.0),t);
9       case 1:
10         return capital*(t<plazo?1.0:1.0+interes/100.0);
11     }
12   }
13 };
```

Pregunta

¿Qué limitaciones tiene la solución anterior?

Pregunta

¿Qué limitaciones tiene la solución anterior?

Problema

Adicionalmente incluye un método a una cuenta que te devuelva su **Tasa Anual Equivalente** (TAE), es decir, el tipo de interés equivalente (en tanto por 100) transcurrido un año (12 meses):

```
double tae()
```

La fórmula siguiente nos da el TAE dado el capital en el mes 12 (C_{12}) y el capital inicial (C_0):

$$TAE = 100 \times \left(\frac{C_{12}}{C_0} - 1 \right) \tag{1}$$

Pregunta

¿Qué limitaciones tiene la solución anterior?

Problema

Añade un nuevo tipo de cuenta, la **cuenta nómina**, que ingrese mensualmente una cantidad fija determinada.

Pregunta

¿Qué limitaciones tiene la solución anterior?

Problema

Añade un método que actualice el capital de la cuenta correspondiente tras el transcurso de un mes.
Cuidado con el diferente comportamiento en cuentas corrientes y plazos fijos...

Pregunta

¿Qué limitaciones tiene la solución anterior?

Problema

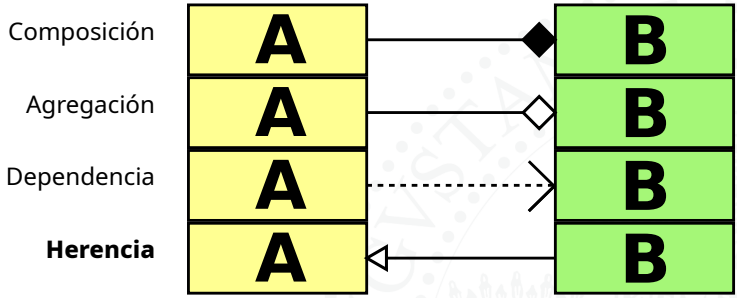
Añade un tipo de cuenta, la **cuenta en divisa**, que, dada otra cuenta en otra divisa y dado un factor de transformación entre divisas, tenga en cuenta el valor de la cuenta en la divisa transformada según dicho factor.

Pregunta

¿Qué limitaciones tiene la solución anterior?

- Keep It Simple, Stupid.
- Responsabilidad única

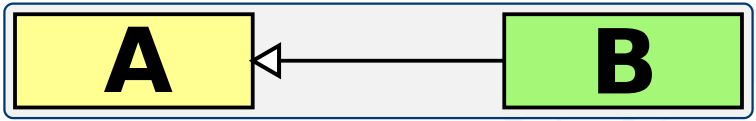
Diferentes tipos de asociación:



Herencia

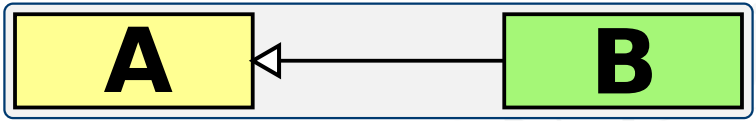
La herencia en POO permite a una clase reutilizar datos y código definidos en otra.

El comportamiento común se recoge en la clase base, y múltiples clases derivadas pueden heredarlo.



C++

```
1 class A { ... };
2 class B : A {
3     ...
4 };
```

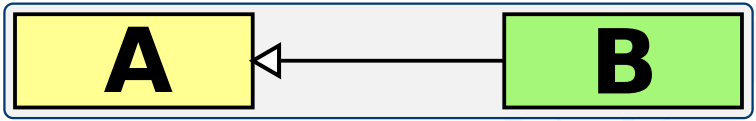


C++

```
1 class A { ... };
2 class B : A {
3     ...
4 };
```

Java

```
1 class A { ... };
2 class B extends A {
3     ...
4 };
```



Terminología

- *B* es clase derivada / clase hija / subclase de *A*.
- *A* es clase padre / superclase de *B*.

Herencia

La clase derivada puede:

- Reutilizar directamente atributos y métodos definidos para la clase padre (polimorfismo de inclusión)
- Definir atributos y/o métodos nuevos
- Usar *this* para acceder a sí mismo y acceder a su/sus superclases.
- Ser referenciada (usada) como un objeto de la clase padre.
- Redefinir métodos existentes en el padre.

Reutilizar directamente atributos y métodos definidos para la clase padre (polimorfismo de inclusión) (C++)

```
1  class A {
2    float atr_a;
3    void  met_a() {  ...  }
4  };
5
6  class B : A {
7    ...
8  };
```

```
1  int main() {
2    B b;
3    b.met_a();
4  }
```

Reutilizar directamente atributos y métodos definidos para la clase padre (polimorfismo de inclusión) (Java)

```java
class A {
  float atr_a;
  void  met_a() {  ...  }
}

class B extends A {
  ...
}
```

```java
public static
    void main(String[] args)
{
  B b = new B();
  b.met_a();
}
```

Definir atributos y/o métodos nuevos (C++).

```
1  class A {
2    float atr_a;
3    void  met_a();
4  };
5
6  void A::met_a()
7  {
8    atr_a = 0.0f;
9  }
```

```
1  class B : A {
2    float atr_b;
3    void  met_b();
4  };
5
6  void B::met_b()
7  {
8    atr_b = 1.0f;
9    atr_a = 0.1f;
10   met_a();
11 }
```

Definir atributos y/o métodos nuevos (Java).

```java
public class A
{
  float atr_a;

  void met_a( ) {
    atr_a = 0.0f;
  }
}
```

```java
public class B extends A
{
  float atr_b;

  void  met_b( ) {
    atr_b = 1.0f;
    atr_a = 0.1f;
    met_a( );
  }
}
```

Usar `this` para acceder a sí mismo y a su superclase (C++).

```
1  class A
2  {
3    float atr_a;
4    void  set(float atr_a);
5  };
6
7  void A::set(float atr_a)
8  {
9    this->atr_a = atr_a;
10 }
```

```
1  class B : A
2  {
3    float atr_b;
4    void  set(float atr_a,float atr_b);
5  };
6
7  void B::set(float atr_a,float atr_b)
8  {
9    this->atr_a = atr_a;
10   this->atr_b = atr_b;
11 }
```

Usar `this` para acceder a sí mismo y a su superclase (Java)

```java
class A
{
  float atr_a;

  void set(float atr_a)
  {
    this.atr_a = atr_a;
  }
};
```

```java
class B extends A
{
  float atr_b;

  void set(float atr_a,float atr_b)
  {
    this.atr_a = atr_a;
    this.atr_b = atr_b;
  }
};
```

Usar ámbitos para acceder a sí mismo y a su superclase (C++)

```
1  class A
2  {
3    float atr_a;
4    void  set(float atr_a);
5  };
6
7  void A::set(float atr_a)
8  {
9    A::atr_a = atr_a;
10 }
```

```
1  class B : A
2  {
3    float atr_b;
4    void  set(float atr_a,float atr_b);
5  };
6
7  void B::set(float atr_a,float atr_b)
8  {
9    A::atr_a = atr_a;
10   B::atr_b = atr_b;
11 }
```

Usar super para acceder a su superclase (Java)

```java
class A
{
  float atr_a;

  void set(float atr_a)
  {
    this.atr_a = atr_a;
  }
};
```

```java
class B extends A
{
  float atr_b;

  void set(float atr_a,float atr_b)
  {
    super.atr_a = atr_a;
    this.atr_b = atr_b;
  }
};
```

Definir la forma de construcción de su superclase (C++)

```
1  class A
2  {
3    float atr_a;
4
5    A(float atr_a);
6  };
7
8  A::A(float a)
9      : atr_a(a)
10 {
11 }
```

```
1  class B : A
2  {
3    float atr_b;
4
5    B(float atr_a,float atr_b);
6  };
7
8  B::B(float a,float b)
9      : A(a), atr_b(b)
10 {
11 }
```

Definir la forma de construcción de su superclase (Java)

```java
class A
{
  float atr_a;

  A(float atr_a)
  {
    this.atr_a = atr_a;
  }
};
```

```java
class B extends A
{
  float atr_b;

  B(float atr_a,float atr_b)
  {
    super(atr_a);
    this.atr_b  = atr_b;
  }
};
```

Ser representada mediante un objeto de la clase padre (Java)

```java
class A {
  float atr_a;
  void  met_a() { ... }
}

class B extends A
{
}
```

```java
public static void main(String[] args)
{
  A a = new B();
  a.met_a();

  B b = new B();
  b.met_a();
}
```

Ser representada mediante un objeto de la clase padre (C++).

Sólo mediante **punteros** o **referencias**.

```
1  class A {
2    float atr_a;
3    void  met_a() { ... }
4  }
5
6  class B : A
7  {
8  }
```

```
1  int main()
2  {
3     B b;
4
5     A* a_ptr = &b;
6     a_ptr->met_a();
7
8     A& a_ref = b;
9     a_ref.met_a();
10 }
```

Redefinir métodos existentes en el padre (C++)

```
1  class A {
2    void met() { ... }
3  };
4  class B : A {
5    void met() { ... }
6  };
```

```
1  int main()
2  {
3    A a;
4    a.met();
5
6    B b;
7    b.met();
8  }
```

Pregunta

¿ A qué método están llamando `a.met()` y `b.met()` ?

Redefinir métodos existentes en el padre (C++)

```
1  class A {
2    void met() { ... }
3  };
4  class B : A {
5    void met() { ... }
6  };
```

```
1  int main()
2  {
3    A* a = new A();
4    a->met();
5
6    B* b = new B();
7    b->met();
8  }
```

Pregunta

¿ A qué método están llamando `a->met()` y `b->met()` ?

Redefinir métodos existentes en el padre (C++)

```
1  class A {
2    void met() { ... }
3  };
4  class B : A {
5    void met() { ... }
6  };
```

```
1  int main()
2  {
3    A* a = new A();
4    a->met();
5
6    B* b = new B();
7    b->met();
8
9    A* ab = new B();
10   ab->met();
11 }
```

Pregunta

¿ A qué método están llamando `a->met()`, `b->met()` y `ab->met()`?

Redefinir métodos existentes en el padre (Java)

```java
class A {
  void met() { ... }
}
class B extends A {
  void met() { ... }
}
```

```java
public static
  void main(String[] args)
{
  A a = new A();
  a.met();

  B b = new B();
  b.met();

  A ab = new B();
  ab.met();
}
```

Pregunta

¿ A qué método están llamando `a.met()`, `b.met()` y `ab.met()`?

La resolución de la asociación nombre⇔entidad se puede decidir mediante:

- Asociación **estática**: `met()` de `A`
  Se basa en el *tipo* de la variable
  y se resuelve en tiempo de compilación (*static binding*).
- Asociación **dinámica**: `met()` de `B`
  Se basa en el *contenido* de la variable
  y se resuelve en tiempo de ejecución (*dynamic binding*).

C++ por defecto usa asociación estática.
El tipo de asociación puede controlarse a nivel de cada método.
Para usar asociación dinámica el método debe marcarse como **virtual** en la clase base.

```
1 class A {
2   virtual void met() { ... }
3 };
```

Java siempre usa asociación dinámica.
No existe asociación estática.

Pero...

- Todo esto de la herencia... ¿para qué vale?
- Las reglas de la herencia son puramente instrumentales y mecánicas. ¿Para qué le sirven al programador?

Pero...

- Todo esto de la herencia... ¿para qué vale?
- Las reglas de la herencia son puramente instrumentales y mecánicas. ¿Para qué le sirven al programador?

Hay que plantearse la **semántica** de la herencia:

- En general, la clase hija suele representar una especialización o una subclasificación de la clase padre.
- En la práctica, no siempre es así.

Inclusión

Las dos reglas siguientes permiten polimorfismo por inclusión:

- Un puntero (referencia) a la clase padre puede estar representando (apuntando) a cualquiera de sus clases hijas.
- Una clase hija puede redefinir métodos de la clase padre (con asociación dinámica).

```
1  class Task {
2  public:
3    virtual void run() const {};
4  };
5
6  void execute(const Task& task) {
7    task.run();
8  }
```

```
1  class Task {
2  public:
3    virtual void run() const {};
4  };
5
6  void execute(const Task& task) {
7    task.run();
8  }
```

- Un puntero (referencia) a la clase padre puede estar representando (apuntando) a cualquiera de sus clases hijas.

```
1  class Task {
2  public:
3    virtual void run() const {};
4  };
5
6  void execute(const Task& task) {
7    task.run();
8  }
```

- Una clase hija puede redefinir métodos de la clase padre (con asociación dinámica).

```
1  class Points : public Task {
2    int times;
3  public:
4    Points(int t) : times(t) { }
5    void run() const {
6      for (int i = 0;i<times;++i) cout<<".";
7    }
8  };
```

- Una clase hija puede redefinir métodos de la clase padre (con asociación dinámica).

```
1  void execute(const Task& task) {
2     task.run();
3  }
4
5  {   ...
6    Points points(1000);
7    execute(points);
8  }
```

- Un puntero (referencia) a la clase padre puede estar representando (apuntando) a cualquiera de sus clases hijas.
- Una clase hija puede redefinir métodos de la clase padre (con asociación dinámica).

```
1  class Shape {
2    virtual float area() const {
3      return 0;
4    }
5    virtual float perimeter() const {
6      return 0;
7    }
8  };
9
10 float aspect_ratio(const Shape& s)
11 {
12   return s.area()/s.perimeter();
13 }
```

Ejemplo (C++)

```
class Shape {
  virtual float area() const {
    return 0;
  }
  virtual float perimeter() const {
    return 0;
  }
};

float aspect_ratio(const Shape& s)
{
  return s.area()/s.perimeter();
}
```

```
class Square : public Shape {
  float side;

  Square(float s) : side(s)
  {}

  float area() const {
    return side*side;
  }

  float perimeter() const {
    return 4*side;
  }
};
```

Ejemplo (C++)

```
class Shape {
  virtual float area() const {
    return 0;
  }
  virtual float perimeter() const {
    return 0;
  }
};

float aspect_ratio(const Shape& s)
{
  return s.area()/s.perimeter();
}
```

```
class Circle : public Shape {
  float radius;

  Circle(float r) : radius(r)
  {}

  float area() const {
    return M_PI*radius*radius;
  }

  float perimeter() const {
    return 2.0*M_PI*radius
  }
};
```

```
1 float aspect_ratio(const Shape& s)
2 {
3   return s.area()/s.perimeter();
4 }
```

La función `aspect_ratio()` se aplica a la clase padre o a cualquiera de sus hijas, incluso aunque no estén definidas

```java
class Sprite
{
  void draw( ) { }
}
```

```java
class Ball extends Sprite {
  int  x,y,radius;
  void draw( ) { ... }
}

class Box extends Sprite {
  int  w,h;
  void draw( ) { ... }
}

class Mario extends Sprite {
  void draw( ) { ... }
}
```

```
1 Sprite[] actors = new Sprite[4];
2 actors[0] = new Ball();
3 actors[1] = new Box();
4 actors[2] = new Box();
5 actors[3] = new Mario();
6 for (int i=0; i<actors.length; i++) actors[i].draw();
7 for (Sprite a : actors) a.draw();
```

El método `draw()` es polimórfico, con comportamiento diferente según el elemento del vector.

Problema

Trabajando en un banco, se te pide que desarrolles un método que calcule el total del valor de una serie de n cuentas de un cliente tras un número t de meses:

```
double total(Cuenta* cuentas[], int n, int t)
```

Hay cuatro tipos de cuentas:

- **Cuentas corrientes :** Tienen un tipo de interés mensual.
- **Plazos fijos :** Vienen determinados por un plazo (número de meses) y un tipo de interés a vencimiento. Antes del plazo, da interés $0\,\%$.
- **Cuenta de nómina :** Se ingresa mensualmente una cantidad fija determinada.
- **Cuenta en divisa :** Dada otra cuenta en otra divisa y un factor de conversión entre divisas, transforme la otra cuenta a la divisa estándar.

Adicionalmente, añade un método que actualice el capital de la cuenta correspondiente tras el transcurso de un mes (nótese el diferente comportamiento entre los diferentes tipos de cuenta).

Seguridad en herencia

Las componentes de una clase pueden ser:

- **public**
- **private**
- **protected**: accesibles para clases derivadas

Se puede aplicar tanto a datos como a métodos.

En el proceso de herencia, se pueden restringir más los permisos de acceso para herencias posteriores (C++):

- class D : **public** B
- class D : **protected** B
- class D : **private** B

Determina la restricción de acceso mínima con el que una clase derivada de D verá las componentes de B.

Verificando la herencia (C++)

```
1  class Shape {
2    virtual void scale(float s)
3    { }
4  };
5
6  void duplicate(Shape& s) {
7    s.scale(2.0);
8  }
```

```
1  class Circle : public Shape {
2    double radius;
3    void scale(double s) {
4      radius*=s;
5    }
6  };
7
8  class Rectangle : public Shape {
9    double width, height;
10   void scale(double s) {
11     width*=s; height*=s;
12   }
13 };
```

¿Qué hace la función `duplicate`?

Verificando la herencia (C++)

```
1  class A {
2   virtual void met() const;
3  };
4
5  class B : public A {
6   void met();
7  }
```

```
1  void f(const B& b)
2  {
3    b.met();
4  }
5
6  void g(B& b)
7  {
8    b.met();
9  }
```

¿ Qué método se utiliza en cada caso ?

¿ Por qué ?

```
1  class Shape {
2  public:
3    virtual void scale(float s) { }
4  };
5
6  class Circle : public Shape {
7    double radius;
8  public:
9    void scale(double s) override {
10     radius*=s;
11   }
12 };
```

Error de compilación.

Seguridad en herencia

```
1  class Shape {
2  public:
3    virtual void scale(float s) { }
4  };
5
6  class Circle : public Shape {
7    double radius;
8  public:
9    void scale(double s) override {
10     radius*=s;
11   }
12 };
```

Error de compilación.

override

```
1  class Shape {
2  public:
3    virtual void scale(float s) { }
4  };
5
6  class Circle : public Shape {
7    float radius;
8  public:
9    void scale(float s) override {
10     radius*=s;
11   }
12 };
```

Compila correctamente.

```
1  class A {
2    virtual void met() const;
3  };
4
5  class B : public A {
6    void met() override;
7  };
```

Error de compilación.

```
1  class A {
2    virtual void met() const;
3  };
4
5  class B : public A {
6    void met() override;
7  };
```

Error de compilación.

```
1  class A {
2    virtual void met() const;
3  }:
4
5  class B : public A {
6    void met() const override;
7  };
```

Compila correctamente.

```
class LastClass final : BaseClass { .... };
```

No se puede extender la clase.

```
class Class {
  void met() final;
};
```

No se puede redefinir el método.

```java
final class ClaseFinal { ... }
```

No se puede extender la clase.

```java
class Clase {
  final void metodoFinal() { ... }
}
```

No se puede redefinir el método.

- Por seguridad: el comportamiento de una clase podría alterarse inesperadamente al cambiar el comportamiento de un método.
- Por eficiencia (en teoría): no hay que *buscar* un método que redefina el método.

Herencia

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-02-10