Introdución a Programación Orientada a Obxectos

Índice

PROGRAMACIÓN ORIENTADA A OBXECTOS

1. Clases e Obxectos

1.1 - As clases e os obxectos

Unha **Clase** indica as características e funcionalidades que debe ter calquera obxecto para ser considerado membro de esa clase. As características se representan mediante variables, mentres que os métodos se definen mediante funcións. Un **Obxecto** é un elemento que pertence a unha clase e polo tanto dispón das características e funcionalidades definidas nesa clase.

> O proceso de creación dun obxecto se denomina **creación de instancia**, e consiste en crear en memoria o conxunto de atributos que distinguirá ese obxecto dos demais. Os valores que teñen eses atributos nun instante determinado se coñecen como ***estado do obxecto***.

1.2 - Clases e os Tipos de Datos

> **Unha Clase é un tipo de datos complexo**, e polo tanto indica canto espazo de memoria se debe utilizar e como interpretar os bytes almacenados en esas posicións de memoria.

A funcionalidade dos obxectos está definida mediante algoritmos en forma de funcións. Eses algoritmos se almacenan nunha área de memoria común a todos os métodos xa que non é necesario que cada obxecto garde unha copia de código que é exactamente igual entre todos obxectos da clase.
Os atributos en cambio representan as características propias de cada obxecto da clase. Cada obxecto debe ter valores propios en esas características que se coñecen como ***estado do obxecto***, e polo tanto cada obxecto debe almacenar na memoria os valores das súas características.

> Cando creamos un obxecto, se reservará un área de memoria onde se almacenará o ***estado do obxecto*** que denominamos "obxecto".

A todos os efectos, unha clase é un tipo de datos. Cando indicamos que unha variable é un obxecto estamos a indicar a cantidade de memoria que vamos a utilizar (a necesaria para almacenar o estado do obxecto) e como interpretar esa información (os bits almacenados representan os valores dos atributos do obxecto).
Si o que definimos é: ***Square c;***
O resultado depende un pouco da linguaxe orientada a obxectos que empreguemos. En C++ estamos creando un espazo en memoria capaz de almacenar o estado (o valor dos atributos) dun obxecto da clase Square. En Java, estamos creando un espazo en memoria capaz de almacenar a dirección (referencia) do estado dun obxecto da clase Square que debemos crear mediante o operador "*new*".

1.3 - Definición de Clases

Cando definimos unha clase debemos indicar: O seu nome, os atributos e a súa protección, e a funcionalidade ofrecida (métodos). Na maior parte das linguaxes de programación a definición dunha clase se fai mediante ***class*** indicando o nome da clase para posteriormente definir as variables que van a formar os atributos e as funcións que implantarán os métodos. O seguinte exemplo define a clase "***Rectangle***" na linguaxe Java:

```
class  Rectangle {
   private double lenhorizontal;
   private double lenvertical;

   public Rectangle(double lenhorizontal,double lenvertical) {
       this.lenhorizontal=lenhorizontal;
       this.lenvertical=lenvertical;
   }

   public double getArea() {
       return lenhorizontal*lenvertical;
   }

   public double getPerimetro() {
       return lenhorizontal*2+lenvertical*2;
   }
}
```

1.3.1 – A Encapsulación

Co obxecto de realizar programas de fácil desenvolvemento, corrección e mantemento se deben crear as clases de xeito que sexan como unha "caixa negra": *Os métodos do obxecto deben facer sempre ben o seu traballo de xeito independente a calquera acción que se poda levar a cabo no resto do programa.*

> Polo tanto *é imprescindible evitar que se poda alterar o estado do obxecto (valores dos atributos) dende fora da clase de modo incontrolado*. Isto se coñece como ***encapsulación***

Para poder deseñar deste xeito as clases, as linguaxes de programación orientadas a obxectos permiten definir a accesibilidade ou *protección* dos atributos, distinguíndose en xeral os niveis ***public*** (O atributo é accesible dende calquera parte da aplicación), ***private*** (O atributo so é accesible dende dentro da propia clase), ***protected*** (O atributo so é accesible dende dentro da propia clase ou calquera clase derivada[1]).

Esta protección *tamén se aplica ós métodos*, co que podemos limitar o uso dos métodos ao ámbito desexado evitando así comportamentos non desexados.

> Si os atributos son *private* o único modo de cambiar ou obter o seu valor é a través de métodos con menor protección de acceso, como *public* ou *protected*. Os métodos que únicamente sirven para cambiar o valor dun atributo privado se chaman ***"setters"***, mentres que os que so sirven para obter o valor almacenado nun atributo se denominan ***"getters"***

1.3.2 – Atributos e Atributos de Clase ou estáticos

Unha clase é unha definición das características e funcionalidades que deben ter os obxectos membros da clase. As características ou atributos se definen mediante variables.

> Os valores que ten un obxecto nos seus atributos nun instante determinado se coñecen como ***estado do obxecto***

Por exemplo, na clase ***Rectangle*** definida anteriormente o estado dun obxecto ***Rectangle*** concreto sería o valor que teñen os seus atributos *lenhorizontal* e *lenvertical*. Distintos obxectos *Rectangle* poderán ter distintos estados (cada un ten o seu propio valor para *lenhorizontal* e *lenvertical*). Os atributos non teñen existencia fora dos obxectos.

As linguaxes orientadas a obxectos permiten tamén definir atributos comúns a todos os obxectos da clase, son os ***atributos estáticos*** **(static).** Cando definimos un atributo ***static*** :

- O atributo existe de xeito independente a que se creen obxectos ou non: **O atributo pertence a clase**.
- O valor almacenado no atributo é común a todos os obxectos que se creen. Os atributos estáticos NON forman parte do estado do obxecto, se non que representan un valor común a todos os membros da clase.

1.3.3 - Métodos e métodos de clase

Do mesmo xeito que se utilizan variables para gardar a información relativa aos atributos da clase (estado do obxecto), se empregan funcións para codificar os algoritmos que lle darán as funcionalidades que deben ter. Esas funcionalidades son propias dos obxectos membros da clase e normalmente modifican o seu estado. Polo tanto:

- Os métodos dun obxecto normalmente alteran o seu estado
- Non existe método si non temos obxecto. Os métodos pertencen a cada obxecto e interactúan co seu estado.

Sen embargo, de xeito similar aos atributos, tamén se poden definir ***métodos de clase*** ou estáticos (***static***):

- Se poden chamar sen necesidade de crear ningún obxecto antepoñendo a clase ao nome do método (xa que o método pertence a clase, non ao obxecto)
- **Non poden alterar nin interactuar co estado do obxecto** xa que non pertencen ao obxecto, se non que son comúns a todos os obxectos da clase e non precisan da creación de ningún obxecto.

1.3.4 – Acceso a métodos e atributos.

Para acceder a un atributo ou método dun obxecto se utiliza habitualmente o operador punto (.), aínda que algunhas linguaxes utilizan outros operadores diferentes (por exemplo, PHP utiliza o operador frecha ->). Este operador separa a variable que almacena o obxecto (ou a referencia do obxecto) do método o función.
Na clase ***Rectangle*** anterior, si temos un obxecto ***r*** de tipo ***Rectangle*** e desexamos acceder ao seu atributo *lenhorizontal* escribiríamos: *r.lenhorizontal* (neste caso non é posible, xa que a protección do atributo ***private*** o impide). Para acceder a un método bastaría con facer por exemplo *double p=r.getPerimetro();*

1 Se entende por "clase derivada" unha clase que herda os atributos e métodos de outra, denominada clase pai ou clase base (ver herdanza)

Si estamos codificando o algoritmo de un método da clase e queremos acceder a un atributo ou a outro método, deberíamos antepoñer o nome do atributo ou método coa variable reservada ***this***, que fai referencia ao obxecto actual.

> O uso de ***this*** dentro da clase depende da linguaxe de programación. A maioría non precisa que especifiquemos ***this*** de xeito explícito salvo unha colisión entre o nome de un atributo / método e outro identificador do algoritmo.

1.4 - A Sobrecarga de operadores e métodos

Unha característica propia das linguaxes orientadas a obxectos é a posibilidade de *sobrecargar* métodos e operadores.

> A **sobrecarga de métodos** consiste na codificación de distintas versións do mesmo método diferenciándose por ter distinto número ou tipos de parámetros. O sistema chamará de xeito automático a versión correcta do método en función dos parámetros que se lle pasen.
>
> Mediante a **sobrecarga de operadores** podemos definir o xeito de comportarse dos distintos operadores sobre datos de distintas clases. Por exemplo non é igual como se debe comportar o operador "+" cando actúa sobre números enteiros que cando actúa sobre obxectos String.

1.5 – A creación de obxectos: Instancia

Cando definimos unha clase estamos deseñando un ***tipo de datos*** definindo que atributos e funcionalidade terán os datos de ese tipo. Un programa deseñado mediante orientación a obxectos consiste en unha serie de ***obxectos*** (pertencentes a distintas clases) utilizando uns as funcionalidades de outros para acada o obxectivo perseguido polo programa.

> O proceso de creación de un obxecto de unha clase determinada se coñece co nome de *creación de instancia*, e habitualmente fai uso dun operador denominado ***new***
>
> Un ***construtor*** e un método "especial" encargado de construír un obxecto da clase e de inicializar os seus atributos. Habitualmente se chama empregando o operador ***new***

> Un ***destrutor*** é un método "especial" que se chama de xeito automático cando un obxecto é destruído (porque finaliza o seu ámbito de vida ou porque se elimina de xeito específico (***dispose***)

this é unha variable do sistema que dentro de unha clase representa o propio obxecto. Se utiliza habitualmente para:

- Diferenciar entre identificadores si se chaman igual que un atributo antepoñendo ***this*** ao nome do atributo.
- Retornar o propio obxecto
- Chamar dende un construtor a outro construtor sobrecargado

1.6 - Composición e Agregación

A composición fai referencia a que un obxecto pode estar composto de outros. Por exemplo, unha *Factura* está composta de *LiñasFactura*. Ou un *Vehículo* está composto dun *Motor.* Existen dous tipos de composición:

- ***Composición débil*** ou *agregación*: Os compoñentes poden ser compartidos por outras clases e reutilizados. *A destrución do obxecto non implica a destrución do compoñente.* **(Empresa composta de Clientes)**
- ***Composición forte*** ou *composición*: Os compoñentes so teñen sentido como parte da clase e non son compartidos con outras clases. *A destrución do obxecto implica a destrución do compoñente*. **(Factura composta de LiñasFactura)**

> No caso da agregación, os compoñentes habitualmente se crean fora da clase e son engadidos mediante un método mentres que no caso da composición os compoñentes se crean dentro da propia clase (normalmente no construtor).

2. A Herdanza

Mediante a herdanza é posible desenvolver novas clases cun mínimo de codificación e permitindo características avanzadas na programación orientada a obxectos como é o ***polimorfismo***.

A ***herdanza*** consiste en definir unha nova clase partindo da definición de outra, herdando todos os seus atributos e métodos

Podemos ver a herdanza como o concepto "é un". Por exemplo, podemos decir que un *Camión*, "é un vehículo", xa que dispón de todas as características e funcionalidades de un *Vehículo*. A clase *Camión* debería herdar da clase *Vehículo*, e polo tanto ter todas as características e funcionalidades dos vehículos.
En cambio, un *Vehículo* non ten por que ser un *Camión* (que pode ter as súas propias características e métodos), por exemplo unha *Bicicleta* é un *Vehículo*, pero non é un *Camión*.

Do mesmo xeito que ***this*** é unha variable reservada polo sistema que sempre fai referencia a clase actual, a variable reservada ***super*** fai referencia a clase pai da clase actual.

2.1 - Herdanza simple e múltiple

Podemos distinguir entre linguaxes de herdanza simple e herdanza múltiple.

A herdanza simple indica que unha clase unicamente pode ter unha clase pai.

Por exemplo unha clase non ten a posibilidade de ser o mesmo tempo unha *Persoa* non podería ser ao mesmo tempo un *Músico* e un *Traballador*, cousa que si é posible nas linguaxes de herdanza múltiple.

Nas linguaxes de herdanza simple pode mitigarse o problema mediante o uso de ***interfaces*** ou facendo uso de composición, utilizando un atributo que aporte a funcionalidade e atributos necesarios.

As linguaxes orientadas a obxecto tamén poden ser de ***raíz única*** ou de ***raíz múltiple***.

Nas linguaxes de raíz única todas as clases descenden de unha única clase base denominada clase raíz. Todas as clases que podamos definir teñen os métodos e atributos definidos na clase raíz.

2.2 – O proceso de construción do obxecto

Para crear un obxecto normalmente facemos uso do operador ***new*** mediante o que chamamos a un método especial da clase denominado **construtor**:

```
Rectangle r=new Rectangle(10,12.5);
```

A definición anterior crearía na memoria RAM un obxecto *Rectangle* de lados 10 e 12.5 e almacenaría a súa posición de memoria na variable ***r***, capaz de almacenar unha referencia de un obxecto *Rectangle*.
Ese obxecto existiría durante toda a vida do programa ou ata que o eliminamos de xeito explícito cun comando da linguaxe (***dispose***) salvo que se dispoña de algún mecanismo de ***recolección de lixo*** encargado de detectar e eliminar de xeito automático os obxectos que xa non teñen utilidade para o algoritmo.
No momento en que un obxecto é eliminado, moitas linguaxes invocan a un método especial da clase denominado ***destrutor***. A misión do destrutor é realizar as tarefas de limpeza que se estimen oportunas segundo a función do obxecto da clase (pechar conexións de rede, conexións a bases de datos, liberar memoria... etc.)

Algunhas linguaxes orientadas a obxectos como C++ poden crear obxectos estáticos definindo unha variable. Por exemplo, con ***Rectangle r(10,12,5);*** se crearía un obxecto ***Rectangle*** de lados 10 e 12.5 identificado pola variable r. Cando a variable r remate a súa vida (porque a execución sae do seu ámbito) o obxecto se elimina.

Os ***construtores*** son os métodos encargados de construír e inicializar os atributos do obxecto. Na maior parte das linguaxes orientadas a obxectos os métodos construtores se chaman igual que a clase e non especifican tipo de retorno, xa que unicamente poden retornar o novo obxecto que se está a crear.

Todas as clases precisan dun método construtor. Si non codificamos o método construtor cando deseñamos a clase, se utiliza un método *sin argumentos* que inicializa os atributos a 0 denominado ***construtor por defecto***.

O proceso de construción dun obxecto é o seguinte:

1. Se crea a parte do obxecto correspondente á clase pai chamando ao construtor por defecto da clase pai
2. Se leva a cabo o algoritmo especificado no método construtor.

Para crear un obxecto antes é necesario crear os obxectos que lle anteceden na árbore de herdanza (o pai, o abó ... etc) o que se fai mediante a execución dos seus construtores.

Si a clase pai non ten un construtor sen argumentos será necesario invocalo de xeito explícito na primeira instrución do algoritmo do construtor empregando a palabra reservada ***super***.

2.2 - Superposición e Polimorfismo

Unha clase terá sempre todos os atributos e métodos que ten a súa clase pai. Sen embargo, non sempre queremos que os métodos se comporten igual.
Si consideramos que un ***Vehículo*** ten que ser capaz de ***acelerar*** (para o que ten un método ***acelerar***) e creamos as clases herdadas ***Camión*** e ***Bicicleta***, polo simple feito de ser ambas un ***Vehículo*** terán o método ***acelerar*** definido na clase ***Vehículo***. O problema é que o proceso (o algoritmo necesario) para acelerar un camión e unha Bicicleta non é o mesmo....

> A ***superposición*** consiste na realización nunha clase de unha versión alterada dun método da clase pai.

Por outra banda, como tanto o *Camión* como a *Bicicleta* son obxectos *Vehículo*, podemos facer isto sen ningún problema:

```
Vehiculo c=new Camión();
Vehiculo b=new Bicicleta();
```

E logo podemos dicirlle a ambos obxectos que aceleren....

```
c.acelera();
b.acelera();
```

Si nos fixamos na declaración anterior, tanto ***c*** como ***b*** son variables que almacenan referencias a un obxecto ***Vehículo***. ¿ Como aceleraría o obxecto ***c*** ? ¿ Como un Camión, ou como un Vehículo ?. ¿ E o obxecto ***c*** ?

> O ***polimorfismo*** fai referencia a que chamadas iguais poden ter efectos distintos que dependen do tipo exacto do obxecto sobre o que se invoque, e polo tanto garante que a chamada anterior executa o código correcto e apropiado ao tipo de Vehículo que se trate.

O polimorfismo nos permite despreocuparnos do tipo exacto dos obxectos e preocuparnos máis da funcionalidade necesaria. Non nos interesa nin importa como se acelera unha Bicicleta ou un Camión, o que nos interesa é simplemente que queremos que acelere. E como ambos son *Vehículo*, sabemos que aceleran.

3 - Clases Abstractas

Cando deseñamos a herdanza, é común que no deseño dunha clase estean moi claras as funcionalidades e atributos necesarios pero que se trate dunha clase demasiado xeral para poder codificar os algoritmos dos métodos.
Por exemplo no caso anterior non está claro como podemos deseñar o algoritmo para acelerar un *Vehiculo*, xa que a clase é tan xenérica que non é posible deseñar un algoritmo que acelere calquera *Vehiculo*. E certo que poderíamos crear un método que acelere a "maioría" de vehículos e logo utilizando a sobreposición crear os métodos apropiados para acelerar outros obxectos *Vehículo* que o precisen, pero iso ten varios problemas:

1. Poderíamos crear obxectos *Vehículo*..... que realmente non corresponden con ningún vehículo concreto
2. Cando deseñamos unha nova clase, por exemplo ***Barco*** e posible que se nos esqueza sobrepoñer o método ***acelera*** e o sistema non detectará ningún problema. O resultado é que o sistema intentaría acelerar os obxectos Barco do mesmo xeito que calquera outro *Vehículo*.

Para solucionar estes problemas podemos declarar a clase *Vehículo* como **clase abstracta**.

> Unha ***clase abstracta*** é unha clase que non permite crear obxectos (instancias) da clase, e que pode non ter codificada a funcionalidade de todos os seus métodos. Basta con indicar que os métodos sen codificar e que clase é abstracta.

Unha clase abstracta ten perfectamente definida toda a súa funcionalidade, pero non ten por que estar completamente implantada.
No exemplo anterior, a clase ***Vehículo*** e o método ***acelera*** serían abstractos. Isto impediría poder crear obxectos Vehículo xenéricos e obrigaría a sobrepoñer o método ***acelera*** en calquera clase herdada que non sexa abstracta, como *Camión*, *Barco* ou *Bicicleta*.

4 – Interfaces e tipos xenéricos

> Unha ***interface*** é un sistema que sirve para que un usuario poda interactuar co sistema. Deste xeito podemos falar de *interfaces de entrada*, como o teclado, *interfaces de saída* como a pantalla, *interfaces textuais* como as dos programas de consola, *interfaces gráficas* como a dos programas en contornas gráficas... etc. Todos eles serven para que o usuario poda acceder as funcionalidades ofrecidas polo sistema.
>
> En programación orientada a obxectos, unha *interface* consiste na ***definición do conxunto de funcionalidades*** ofrecidas por unha clase. Todas as clases teñen unha *interface* que define o xeito no que o usuario pode acceder a funcionalidade dos seus obxectos, pero tamén é posible definir unha interface sen necesidade de codificar os algoritmos que a implantan co obxecto de garantir que unha clase incorpora esa funcionalidade.

Unha interface en programación orientada a obxectos define un conxunto de funcionalidades que deben crear os obxectos que a implanten, O conxunto de métodos que compoñen a interface se denomina **API** da interface.

Cando dicimos que unha clase implanta unha interface queremos dicir que a clase dispón de todos os métodos definidos na interface e polo tanto nos ofrece toda a funcionalidade definida na interface.
De xeito similar a herdanza, podemos tratar os obxectos de clases que implanten unha interface como obxectos do "tipo" da interface e facer uso de polimorfismo. Por exemplo a seguinte interface Java define a funcionalidade necesaria para realizar debuxos xeométricos:

```
public interface Canvas {
        public void point(int y,int x,Color color) throws OutOfCanvasException;
        public void line(int y,int x,int y1,int x1,Color color) throws OutOfCanvasException;
        public void circle(int y, int x,int r, Color color) throws OutOfCanvasException;
        public void text(int y,int x,String text,Color color) throws OutOfCanvasException;
}
```

A partir do interface anterior poderíamos crear clases coa completa funcionalidade de un Canvas que tiveran versións dos métodos capaces de facer o traballo sobre distintos dispositivos de saída:

```
public class TextCanvas implements Canvas {
  /* Implementariamos todos os métodos definidos en Canvas de
     Xeito que a saída se amosa na consola de texto
  */
}
```

```
public class GraphicCanvas implements Canvas {
  /* Implementariamos todos os métodos definidos en Canvas
     de xeito que a saída se amosa na contorna gráfica.
  */
}
```

Esto permite crear obxectos de tipo *Canvas* capaces de visualizar a saída nunha pantalla de texto (como a consola) ou nunha pantalla gráfica (como unha fiestra do GUI):

```
Canvas pintor_console=new TextCanvas();
Canvas pintor_gui=new GraphicCanvas();

pintor_console.circle(2,10,123,Color.BLUE);   // Dibuxa un circulo na consola
pintor_gui.circle(2,10,123,Color.BLUE);   // Dibuxa un circulo nunha pantalla gráfica
```

As interfaces nos permiten definir unha funcionalidade sen necesidade de programar os métodos. Para que a aplicación funcione precisaremos de obxectos que teñan esa funcionalidade definida. Isto nos permite unha vez definida a funcionalidade necesaria empregar un obxecto para facer a aplicación incluso sin necesidade de crear unha clase que a implante. Un ***driver*** non é mais que a implantación da funcionalidade dun interface.

4.4.1 - Tipos xenéricos

Cando deseñamos unha clase, e mais comunmente cando deseñamos unha clase abstracta ou unha interface nos podemos atopar con que sería interesante poder dispoñer de toda a funcionalidade definida facendo uso de distintos tipos de datos. Por exemplo a seguinte interface nos proporciona un API para almacenar e recuperar obxectos de tipo String asociados a unha clave:

```
public interface StorageInterface {
        void almacena( String elemento, String clave);
        String recupera(String clave);
}
```

```
public class Storage implements StorageInterface {
        void almacena( String elemento, String clave) {
          /** Algoritmo **/
        }
        String recupera(String clave) {
          /** Algoritmo **/
        }
}
```

Si queremos almacenar obxectos *Cliente*, esta interface non é útil, e teríamos que definir outra incluso si o algoritmo para almacenar un String fora exactamente o mesmo que para gardar un *Cliente* ou calquera outro tipo de datos. Teríamos que facer unha implantación distinta da interface para cada tipo de datos que queiramos almacenar.

Os tipos xenéricos nos permiten indicar un "***tipo xenérico de datos***" que substitúe o compilador polo tipo apropiado antes de traducir o código fonte.

Por exemplo, o código anterior en Java quedaría algo así:

```
public interface Storage<K,V> {
        void almacena( V elemento, K clave);
        V recupera(K clave);
}
```

```
public class Storage implements StorageInterface<K,V> {
        void almacena(V elemento, K clave) {
          /** Algoritmo **/
        }

        V recupera(K clave) {
          /** Algoritmo **/
        }
}
```

Isto permite crear un obxecto capaz de almacenar e recuperar obxectos Cliente polo número de cliente: ***Storage<Integer,Cliente> data=new Storage<>();***. O sistema substituiría a ***K*** por ***Integer*** e a ***V*** por ***Cliente*** sen necesidade de crear unha clase cos tipos específicos.

4.4.2 - Interfaces funcionais

Si definimos unha interface cun único método, realmente non precisaríamos indicar explicitamente o nome do método cando facemos uso da funcionalidade, porque so temos unha.

> Una interface que define un único método se denomina ***interface funcional*** e nos permite crear obxectos e invocar o método dun xeito abreviado denominado *expresión lambda*.

4.4.3 – Clases anónimas e expresións Lambda

> Unha ***clase anónima*** é unha clase que carece de nome e se define e instancia no mesmo lugar. E moi útil para a creación de obxectos que implanten interfaces simples ou clases abstractas breves.
>
> Unha ***expresión lambda*** ou función anónima é un método do que non especificamos o nome. Se utilizan normalmente para crear obxectos que implantan interfaces funcionais, xa que como unicamente teñen un método non e necesario especificar o nome de xeito explícito.

```
public interface Visible {
   public void show(String str);
}
```

```
public class Exemplo {
  private String description;

  Public Exemplo(String d) {
        description=d;
  }

   /* Amosa a información do obxecto no obxecto Visible v)
   public void amosaDescricion(Visible v) {
        v.show(description);
   }
}
```

```
Exemplo e=new Exemplo("Hola Mundo");
// Para utilizar o método amosaDescricion debemos crear e pasar como argumento un obxecto de tipo Visible
// Visible é unha interfaz. Non podemos instanciar obxectos Visible, necesitamos definir unha clase que implemente Visible
//
// 1.- Visualización con clase anónima.
e.amosaDescricion(new Visible() {
        public void show(String str) {
                System.out.println(str);
        }
});

// 2.- Visualización con expresión Lambda
e.amosaDescricion((String str)->{System.out.println(str);});
```

5 - Introspección

> A **introspección** é a capacidade de un obxecto de facilitar información da estrutura da súa propia clase: o seu tipo, métodos, atributos, clase pai, ... etc.

A introspección é unha característica que se emprega para a implantación do ***polimorfismo***