Clases e Obxectos

Cando resolvemos un problema mediante orientación a obxecto o miramos como un conxunto de obxectos actuando entre eles. Eses obxectos teñen unhas características determinadas (atributos) e son capaces de levar a cabo unhas accións concretas (métodos). No enunciado, podemos comprobar que os substantivos corresponden normalmente con obxectos ou atributos, e os verbos con métodos.

Una vez que temos localizados os obxectos que interveñen no noso problema, debemos xeneralizalos describindo os atributos e métodos dun xeito xeral, deseñando Clases ás que pertencen os obxectos que aparecen no problema.

Os valores particulares dos atributos dun obxecto nun instante determinado se denominan ***estado do obxecto***.

Unha Clase é unha definición, e un obxecto é un elemento concreto que pertence a unha clase, e polo tanto compre coa definición descrita na Clase.

EXEMPLO

Desexamos automatizar a nosa vivenda, de modo que podamos controlar mediante un programa que desenvolveremos en Java a temperatura, humidade, a iluminación e as persianas das habitacións. Para poder facer isto dispoñemos dun aparello de aire acondicionado controlable por software, de xeito que podemos seleccionar o grao de humidade e a temperatura. Tamén dispoñemos de persianas "intelixentes" que podemos subir, baixar ou poñer a unha porcentaxe de apertura indicada. En canto a iluminación, as lámpadas da casa se poden regular de luminosidade 0 (apagadas) a 100 (luminosidade máxima).

O que queremos é un programa que nos permita xestionar e saber o estado actual do aire acondicionado, as persianas da Salón, Habitación Principal e Habitación de Invitados, e as lámpadas do Salón, Habitación Principal, Habitación de Invitados e Cociña.

Análise:

Neste exemplo, vemos que temos obxectos concretos como a persiana do Salón ou a persiana da Habitación de Invitados e temos clases como a Persiana as que pertencen eses obxectos. Podemos distinguir as seguintes clases, atributos e métodos examinando os substantivos e verbos:

- temperatura, humidade son atributos do AireAcondicionado
- a iluminación é un atributo das Lámpadas
- a "porcentaxe de apertura" (posición) é un atributo das Persianas
- O AireAcondicionado se lle pode poñer a temperatura e humidade que se desexe, eso son métodos do AireAcondicionado
- As lámpadas se poden poñer a unha iluminación concreta. Eso e un método das Lámpadas
- As persianas se poden regular en altura. Eso e un método das Persianas. Tamén se deben poder "subir" completamente e "baixar" completamente.

Atributos e Métodos

Os atributos das clases en Java se representan mediante a definición de variables, e os métodos mediante a definición de funcións.

EXEMPLO

```
class AireAcondicionado {          // Definición da clase
   int humidade;                   // Atributo - Porcentaxe de humidade (0-100)
   int temperatura;                // Atributo - Tempeartura de (15 a 40)

   int setTemperatura(int temperatura) { // Método
          ... algoritmo do método...
   }

   int setHumidade(int humidade) { // Método
         ...algoritmo do método....
   }
}
```

```
class Lampada {            // Definición da clase
   int luminosidade;       // Atributo - De 0 (apagada) a 100 (máximo)

   int setLuminosidade(int luminosidade) {  // Método
          ... algoritmo do método...
   }
}
```

```
class Persiana {           // Definición da clase
   int posicion;           // Atributo - De 0 (pechada) a 100 (completamente aberta)

   int setPosicion(int pos) { // Método
          ... algoritmo do método...
   }

   int abrir() { // Método
          return setPosicion(100);
   }

   int pechar() { // Método
          return setPosicion(0);
   }
}
```

Unha vez deseñadas as clases (non necesitan estar totalmente escritas, o importante e ter claro de que métodos teñen que dispoñer e en menor medida, os atributos) poderíamos realizar o programa, creando os obxectos concretos que interveñen no problema. No noso caso serían as Lámpadas do Salón, Habitación Principal, Habitación de Invitados e Cociña, as Persianas do Salón, Habitación Principal e Habitación de Invitados, e o Aire Acondicionado. A creación de obxectos concretos se denomina ***instanciación*** e en Java se realiza co operador **new**.

En Java, cando creamos un novo obxecto con se reserva unha zona libre na memoria, se crean ahí as estructuras necesarias e se devolve a súa dirección. O que nos gardamos na variable sempre será a dirección onde se atopa realmente a información que define o obxecto.

```
Lampada l_salon=new Lampada();

Lampada l_habitacion_principal=new Lampada();
```

Construtores e Destrutores: this

A creación dun novo obxecto supón a creación das estruturas en memoria necesarias para gardalo estado do obxecto, e en colocar os valores iniciais nos atributos. O encargado de facer isto é un método "especial" da clase denominado ***construtor.***

O construtor é o encargado de realizar as tarefas necesarias para obter un obxecto válido (poñer os valores iniciais nos atributos, e calquera tarefa adicional que sexa necesaria). Si nos cando deseñamos a clase non incluímos un, Java xera un *construtor sen argumentos* denominado ***construtor por defecto***. No momento en que nos escribamos un construtor na clase, Java xa non o proporciona.

Os construtores son métodos que se chaman exactamente igual que a clase e para os que non se indica valor de retorno (retornan sempre a dirección de memoria onde se creou o obxecto).

Dentro dos obxectos, se dispón sempre dunha variable reservada denominada **this** que almacena sempre a dirección do propio obxecto. Habitualmente se utiliza esta variable para acceder a atributos e métodos da clase en caso de conflitos de nomes.

EXEMPLO

A clase AireAcondicionado anterior, podemos deseñala cun construtor que estableza uns valores por defecto nos atributos....

```
class AireAcondicionado {          // Definición da clase
   int humidade;                   // Atributo - Porcentaxe de humidade (0-100)
   int temperatura;                // Atributo - Tempeartura de (15 a 40)

  AireAcondicionado(int humidade,int temperatura) {  // Construtor
          this.humidade=humidade;          // Necesitamos this, para distinguir entre o atributo e o parámetro
          this.temperatura=temperatura;
  }

   int setTemperatura(int temperatura) { // Método
          ... algoritmo do método...
   }

   int setHumidade(int humidade) { // Método
         ...algoritmo do método....
   }
}
```

En algunhas linguaxes orientadas a obxecto o tempo de vida dos obxectos está plenamente establecido, se sabe exactamente o momento en que un obxecto deixa de existir. En Java, non é así. Java é unha linguaxe de memoria xestionada, na que un proceso en segundo plano elimina os obxectos non utilizados cando o considere oportuno, o que se chama un ***garbage collector***.

Do mesmo xeito que dispoñemos dun método especial encargado de construír obxectos da clase, existe un método que se executa sempre no instante en que o obxecto deixa de existir, un ***destrutor.*** Os destrutores normalmente realizan tarefas de "limpeza" necesarias cando o obxecto desaparece, como eliminación de ficheiros temporais, peche de conexións... etc. En Java ese método se chama ***finalize()*** e normalmente non se aconsella o seu uso xa que debido a memoria xestionada non se pode saber cando o obxecto é liberado ou si vai ser liberado. A efectos prácticos podemos facernos a idea de que en Java NON temos destrutores.

Ocultación

Un dos principios da orientación a obxecto é que os obxectos deben comportarse como “caixas negras” que ofrecen unha funcionalidade a través dos seus métodos garantida baixo calquera circunstancia, sen importar o que se poda facer fora da clase (*encapsulación*).

Para conseguir iso é necesario poder restrinxir o acceso a atributos e métodos impedindo que se poñan valores erróneos nos atributos ou se utilicen métodos de formas non axeitadas.

EXEMPLO

As Persianas admiten posicións dende 0 (baixada) a 100 (completamente aberta). Si non restrinximos o acceso aos atributos un programa podería facer o seguinte:

```
Persiana p_salon=new Persiana();
p_salon.posicion=589;                // Isto podería provocar o mal funcionamento da Persiana, está fora de rango
```

Para evitar isto podemos “protexer” o acceso aos atributos e métodos impedindo si o desexamos a súa manipulación directa dende puntos específicos do programa. Os graos de protección son:

- **public** – Acceso permitido
- **friendly** – Acceso permitido so dende clases do mesmo “package”. E o valor por defecto si non se especifica nada.
- **protected** – Acceso permitido so dende a propia clase, dendo o mesmo “package” ou clases descendentes (que “heredan” de esta)
- **private -** Acceso so permitido dende a propia clase

Como norma xeral, o mellor e que os atributos sexan **private**, e que os métodos public sexan únicamente os que proporcionan funcionalidade a aplicación.

Copia e comparación

Cando creamos un obxecto recibimos a súa dirección de memoria. Cando traballamos con obxectos estamos traballando realmente con referencias. Esto ten certas consecuencias:

EXEMPLO

```
Persiana p_salon=new Persiana();
Persiana p_habitacion;
p_habitacion=p_salon;                // A referencia do obxecto se copia a p_habitacion. Apuntan ao mesmo obxecto !!!

p_habitacion.setPosicion(100);       // Abre a persiana p_salon !!!
```

Como podemos observar o operador de asignación non “ten o mesmo efecto” que cos tipos primitivos, non “copia” o obxecto dunha variable a outra. En realidade o funcionamento é o mesmo, o que ocorre é que a variable é unha referencia ao obxecto Persiana que se crea con new, e iso é o que se copia.

Si realmente desexamos crear unha “copia”, o que se fai normalmente e deseñar un construtor que recibe como argumento un obxecto da propia clase, e que se encarga de copiar os atributos

relevantes para que consideremos os dous obxectos “iguais”. Ese tipo de construtor se denomina ***construtor de copia***.

EXEMPLO

```
class Persiana {            // Definición da clase
   int posicion;            // Atributo - De 0 (pechada) a 100 (completamente aberta)

  Persiana(Persiana p) { // Constructor de Copia
        this.posicion=p.posicion;
  }

   int setPosicion(int pos) { // Método
        ... algoritmo do método...
   }

   int abrir() { // Método
        return setPosicion(100);
   }

   int pechar() { // Método
        return setPosicion(0);
   }
}
```

```
Persiana p_salon=new Persiana();
Persiana p_habitacion;
p_habitacion=new Persiana(p_salon); // p_habitación referencia a un NOVO obxecto cos mesmos valores que p_salon

p_habitacion.setPosicion(100);      // Abre a persiana p_habitación, non p_salon
```

Un problema similar se da cando queremos determinar si dous obxectos da mesma clase “son iguais”. O primeiro problema é que non existe un modo “estándar” de decidir cando dous obxectos se consideran iguais, é algo subxectivo. O segundo problema é que a comparación co operador de comparación (==) non funciona, xa que compararemos as direccións. Este operador determinaría cando dous obxectos **son o mesmo**, xa que conteñen a mesma dirección.

Para solucionar este problema se recorre a un método que recibindo un obxecto da mesma clase decida si se pode considerar igual ao actual ou non (normalmente comparando os atributos).

Escribir un método que se encargue de iso é perfectamente válido, pero a clase Object de Java nos proporciona un método denominado ***equals*** que se encarga preferentemente de esta cuestión. Debemos sobrepoñer o método equals para dotar as nosas clases da capacidad de decidir cando dous obxectos son iguais ou non.

equals ten a seguinte forma:

public Object equals(Object obj)

Herdanza, Sobreposición e Polimorfismo - super

A herdanza é unha das características máis potentes da programación orientada a obxecto, e consiste en crear novas clases a partindo dos atributos e métodos de outra. Tamén se denomina "especialización", xa que a partir dunha clase máis xeral (Lampada) podemos deseñar clases máis concretas (LampadaColor). Podemos dicir que unha LampadaColor ***é unha*** Lampada

EXEMPLO

```
class LampadaColor extends Lampada {          // Definición da clase heredando de Lampada
   private int cor;                           // Atributo – cor da luz en RGB

  public setColor(int cor) {                  // Método para cambiar a cor da Lámpada
          this.cor=cor;
  }
}
```

```
LampadaColor lc=new LampadaColor();
lc.setColor(145);                             // Poñemos color
lc.setLuminosidade(50);                       // Poñemos luminosidade
```

Con esta definición LampadaColor herda os atributos (*luminosidade*) e os métodos (*setLuminosidade*) da súa clase base Lampada. Únicamente temos que definir as variacións sobre a clase base.

É posible que aínda que na nova clase dispoñemos dos mesmos métodos que ten a clase base, o comportamento dos mesmos deba ser algo diferente. Cando é así, e posible escribir na clase derivada unha nova versión do método sobrepoñendo o comportamento do método orixinal. Isto se coñece co nome de ***sobreposición.***

EXEMPLO

Supoñamos que na clase LampadaColor cando poñemos a luminosidade necesitamos antes establecer a cor da lámpada á seleccionada no seu atributo.....

```
class LampadaColor extends Lampada {          // Definición da clase heredando de Lampada
   private int cor;                           // Atributo – cor da luz en RGB

  public setLuminosidade(int luminosidade) { // SOBREPOSICIÓN – sobreescribo o método declarado en Lampada
          establecerCor(cor);
          ... Algoritmo do método ....
  }

  public setColor(int cor) {                  // Método para cambiar a cor da Lámpada
          this.cor=cor;
  }
}
```

Java é capaz de determinar en tempo de execución de que clase é exactamente un obxecto e invocar ao método apropiado. E o que se coñece como ***polimorfismo***. Podemos dicir neste caso que o obxecto Lampada pode ter dúas formas: unha Lampada xenérica ou unha LampadaColor.

EXEMPLO

```
Lampada lamp=new LampadaColor();      // É perfectamente válido, xa que unha LampadaColor é unha Lampada
lamp.setColor(123);                   // ERROR – Lampada non ten un método setColor
lamp.setLuminosidade(40);             // Se comporta como unha LampadaColor, se chama ao método sobreposto
```

Nas clases herdadas dispoñemos dunha variable reservada que almacena a referencia ao obxecto pai denominada **super**. Podemos utilizar esta variable para facer referencia directa a métodos e atributos da clase base.

Cando creamos unha nova clase herdando de outra, para construír un obxecto da mesma primeiro é necesario construír as estruturas indicadas da clase base, iso se realiza por defecto executando o construtor por defecto da clase base de modo automático (o construtor sen argumentos). O que ocorre moitas veces é que xa temos deseñado un construtor na clase base, e que ese construtor ten argumentos, co que o intento de construción dos obxectos das clases derivadas fallan (ao non existir un construtor sen argumentos na clase base). Existen dúas solucións a este problema:

1. Crear un construtor sen argumentos na clase base
2. Poñer como primeira instrución no construtor da clase derivada unha chamada ao construtor da clase base empregando super: ***super(... lista argumentos ...);*.** Debe ser a primeira instrución, xa que antes de facer nada na clase derivada é necesario ter construídas as estruturas necesarias para a clase base.

Java é unha linguaxe de herdanza única (unha clase so pode herdar de outra, non de varias) e de raíz única (todas as clases herdan dunha clase base inicial denominada **Object**).

A clase Object proporciona algúns atributos e métodos que son herdados por TODAS as clases Java que se deseñen. O máis utilizados e sobrepostos e o método equals (para decidir cando dous obxectos son iguais).

Sobreposición de Equals

Si queremos escribir un método que decida cando un obxecto da nosa clase o consideramos igual a outro a mellor opción é sobrepoñer ***equals***. Este método se debe sobrepoñer do modo que se indica no seguinte exemplo para a clase *ClaseExemplo*:

```
public boolean equals(Object obj) {
        ClaseExemplo ce;

        if (obj==null) return false;                    // Si obj non apunta a ningún obxecto, non é igual a este
        if (obj==this) return true;             // Si obj é o mesmo obxecto, son iguais
        if (! (obj instanceof  ClaseExemplo))  return false;  // Si obj non é de ClaseExemplo, tampouco é igual a este
        // Xa sei que obj é de ClaseExemplo, pero para poder accedeer aos atributos e métodos definidos en
        // ClaseExemplo necesito unha variable de tipo ClaseExemplo, realizao un Casting e o almaceno
        ce=(ClaseExemplo) obj;
        ..... Comparación de Atributos para decidir si son iguais ou non ....
}
```

Neste exemplo podemos ver unha técnica chamada "***casting***". O "casting" consiste en "forzar" a interpretación de información de un tipo como si fora dun tipo distinto, e unicamente se pode facer si a información é "compatible" co tipo.

```
float f=123.5;
double x=(double) f;        // "Interpretamos" o float que ten f, como un double. E unha conversión válida
```

Métodos e atributos estáticos e finais

Como vimos ata agora os atributos se definen cando deseñamos a clase pero non existen ata que creamos un obxecto. Cada obxecto ten o seu propio estado (os seus propios atributos cos seus propios valores). Do mesmo modo, a funcionalidade pertence ao obxecto, non á clase. As clases son meras definicións de tipos.

No obstante existen situacións en que desexamos manter valores comúns para todos os obxectos da clase, ou desexamos poder utilizar un método sen necesidade de ter que crear un obxecto porque

proporcionan unha funcionalidade “xenérica” que non depende do obxecto (calcular unha raíz cadrada, por exemplo). Cando ocorre isto podemos definir os atributos ou os métodos coma ***static***

Si por outro lado non queremos que de unha clase se podan heredar novas clases, podemos declarar a clase como **final.** Nos atributos a declaración de final fai que o valor establecido ao construír o obxecto non se poda cambiar, e nun método, que non se poda sobrepoñer nunha clase heredada.

Xestión de Erros: Excepcións

Na programación clásica as condicións de erro se indicaban con valores de retorno das funcións. O programa debía controlar o valor de retorno para comprobar si se levaba a cabo correctamente, e tomar as medidas oportunas en caso contrario.

Como a comprobación dos posibles valores de retorno era vontade do programador moitos erros quedaban sen comprobar. Ademais o control de erros con este esquema supoñía un montón de sentencias condicionais pesadas de escribir e que enredaban o código.

As Excepcións son a solución ofrecida pola programación orientada a obxecto para este problema. No momento en que se detecta un erro, un programa pode lanzar un obxecto ***Exception***. Ese obxecto debe ser capturado e xestionado.

En Java podemos distinguir dúas “familias” de Exception:

- ***checked exception*** – Deben ser capturadas e tratadas obrigatoriamente, ou no seu defecto, indicar que o método as “relanza”.
- ***uncheked exceptions*** – Se poden ignorar, aínda que si se producen durante a execución do programa e non son capturadas provocarán que o programa finalice de forma incontrolada.

Preferiblemente sempre se deben utilizar *checked exceptions*. Para “capturar” e xestionar as exceptions se utilizan bloques try... catch... finally

```
try {
        .... instruccións que poden lanzar Exceptions... en este caso ClaseException, ClaseException2, ou outra...
} catch (ClaseException variable) {
        ... xestión do erro ClaseException. Os detalles do erro están en variable
} catch (ClaseException2 variable) {
        ... xestión do erro ClaseException. Os detalles do erro están en variable
}.....
} catch(Exception variable) {
        ... Aquí captura todas as Exception que non foron xestionadas con anterioridade ....
} finally {
       ... Estas instrucións se executarán existan erros ou non, incondicionalmente, ao final do método.....
}
```

tamén é válido un bloque try{} finally{} ou simplemente try{} catch() {}

```
try {
        .... instruccións que poden lanzar Exceptions... en este caso ClaseException, ClaseException2, ou outra...
} finally {
       ... Estas instrucións se executarán existan erros ou non, incondicionalmente, ao final do método.....
}
```

```
try {
        .... instruccións que poden lanzar Exceptions... en este caso ClaseException, ClaseException2, ou outra...
} catch(ClaseException e) {
       ... Estas instrucións se executarán existan erros ou non, incondicionalmente, ao final do método.....
}
```

Arrays

Cando desexamos almacenar un número relativamente grande de datos do mesmo tipo non é práctico o uso de variables independentes. Imaxinemos que queremos almacenar as idades dos 30 alumnos dunha clase... ¿utilizamos 30 variables distintas?. Aparte de pouco flexible (que pasa si mañá son 35?) non é manexable.

A solución pasa por utilizar unha estrutura de datos capaz de almacenar un conxunto de datos do mesmo tipo, e que permita seleccionar o dato que queremos utilizar mediante un ou varios índices: un ***Array.***

En Java os Array son obxectos con atributos como ***length*** que nos da o seu número de elementos e os seus índices comenzan en 0. Java tamén proporciona a clase ***Arrays*** que contén un conxunto de métodos estáticos útiles para a xestión (ordeacións, búsquedas, inicialización.... etc).

A definición dun array en Java é así:

```
Lampada[] l;  // Define unha variable capaz de almacenar a dirección dun Array de direccións a obxectos Lampada
l=new Lampada[20];  // Crea un Array de 20 elementos que son direccións a obxectos Lampada

// En lamp NON TEÑO 20 OBXECTOS Lampada. So teño 20 espacios para gardar direccións de obxectos Lampada
l[0]=new Lampada();        // En l[0] teño a dirección dun obxecto Lampada, pero en l[1], non
```

Para percorrer os arrays se utiliza un ou varios índices segundo o tipo de array, pero Java proporciona unha forma abreviada de for que nos facilita a tarefa:

```
for (Lampada l: aLamp) {
        System.out.println(l);        // l vai tomando todos os valores do array aLamp, de un en un...
}
```

Sería equivalente a:

```
for (int idx=0;idx<aLamp.length;idx++) {
        System.out.println(aLamp[idx]);
}
```

Cando definimos un Array de varias dimensións debemos ter en conta que en realidade estamos definindo “Arrays de Arrays”. Vexamos un exemplo:

```
String[][] str;      // str é capaz de almacenar unha referencia a un Array que almacena referencias a Arrays de String

// --> Caso A

str=new String[10] []; // en str teremos a referencia ao Array que almacena referencias a Arrays String que estamos creando (10 elementos)
str[0]=new String[4];  // en str[0] teremos a referencia de un Array con capacidade para referenciar a 4 Strings
str[1]=new String[8]; // en str[1] teremos a referencia de un Array con capacidade para referenciar a 8 Strings

// ---> Caso B) Tamén poderíamos facer....

str=new String[10][8]; // str será a referencia a un array de 10 elementos que referencian a Arrays de 8 elementos que referencian a Strings
```