Introdución a OOP: Obxectos e Clases

A programación orientada a obxectos é un método de análise de problemas e de creación de algoritmos. En lugar de pensar na solución aos problemas simplemente como unha serie de pasos a seguir se analiza buscando os elementos (*entidades*) que interveñen no problema (normalmente identificados polos nomes que aparecen na definición do problema) e as responsabilidades que esas entidades deben ser capaces de asumir (accións que deben ser capaces de levar a cabo, normalmente identificadas por verbos que aparecen na definición do problema).

O programa modélase indicando a secuencia de interaccións entre os diferentes obxectos intercambiando “mensaxes” (e dicir, chamando aos seus métodos).

Na **Programación Orientada a Obxectos (OOP)** encapsúlanse os datos (atributos) e comportamento (métodos) nunha unidade chamada **clase**, que é unha definición de tipo de datos a partir da que podemos crear (instanciar) obxectos que son os que realmente teñen a información e as funcionalidades.

Imaxina que estás organizando unha oficina. En lugar de pensar 'primeiro fago isto, despois aquelo', pensas: 'necesito un xefe, secretarias, contables... Cada un ten o seu traballo e comunícanse entre eles de xeito que o xefe para facer o seu traballo pode facer uso da secretaria e do contable. Así funciona a POO.

```
public class Persona {
    String nome;
    int idade;

    public Persona(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }

    public void saudar() {
        System.out.println("Ola, son " + nome + " e teño " + idade + " anos.");
    }
}
```

Este código define un tipo de datos que representa a unha persoa: a ***clase Persona***.

Esta clase define que os obxectos da clase persona teñen un nome (String) e unha idade (int). Son os atributos dos obxectos persona. O valor deses atributos nun instante concreto se denomina **estado do obxecto**.

Tamén podemos observar as capacidades dos obxectos Persoa, o que poden facer, neste caso “saudar”.

O método que vemos Persona, que se chama igual que a clase, recibe o nome de ***construtor***, e é un método especial que sirve para crear obxectos da clase (instanciar obxectos) facendo uso do operador ***new:***

Persona juan=new Persona(“Juan González”,27);

Esta instrución crea na memoria un obxecto Persona cos atributos nome e idade, e devolve a súa dirección ou ***referencia***, que se almacena na variable ***juan***. ***O estado*** da persoa referenciada pola variable juan, e {nome:“Juan González”,edade: 27 }

Unha vez creado o obxecto podemos facer uso da súa funcionalidade: ***juan.saudar();***

Os construtores. A variable *this*

Un **constructor** é un método especial usado para **inicializar** un novo obxecto da clase. En Java ten estas propiedades claves:

- O seu nome coincide exactamente co nome da clase.
- **Non** ten tipo de retorno (nin sequera `void`, xa que retorna o obxecto da clase construído).
- Pódese declarar con parámetros ou sen parámetros. Si non definimos ningún construtor o sistema proporciona un sin parámetros denominado *construtor por defecto*.
- Pódese sobrecargar. Isto quere decir que se poden crear diferentes construtores con distintos argumentos (en número ou tipo) para proporcionar distintos xeitos de crear o obxecto
- Si creamos un construtor, o sistema xa non prové do construtor sen parámetros (construtor por defecto). Si queremos ter un o temos que poñer de xeito explicito.

O constructor prepara o estado inicial do obxecto (asigna valores aos atributos, valida argumentos, etc.). A seguinte clase ten o construtor sobrecargado, proporcionando tres xeitos de crear obxectos (instancias) da clase. Un sin argumentos, outro con dous argumentos String e outro cun unico argumento String:

```
public class Persoa {
        String nome; // atributo de instancia
        int idade;

        // Constructor sen argumentos
        public Persoa() {
                // inicialización por defecto
                this.nome = "";
                this.idade = 0;
        }

        // Constructor sobrecargado
        public Persoa(String nome, int idade) {
                this.nome = nome; // 'this' refírese ao propio obxecto, para indicar que e o atributo
                this.idade = idade;
        }

        // Constructor sobrecargado
        public Persoa(String nome,) {
                this.nome = nome; // 'this' refírese ao propio obxecto, para indicar que e o atributo
                this.idade = 0;
        }

}
```

Isto nos permite crear obxectos persoa destas dúas maneiras:

1. Persoa p=new Persoa(); // A persoa terá “” como nome e 0 como idade
2. Persoa p=new Persoa(“Juan”,31); // A persoa terá “Juan” como nome e 31 como idade
3. Persoa p=new Persoa(“Luis”); // A persoa terá “Luis” como nome e 0 como idade

A variable this

Observando o exemplo anterior, vemos que nos construtores se antepón **this.** ao nome dos atributos. ***this*** é unha variable especial en java (e en outras linguaxes OOP. En Python se chama ***self***) que sempre almacena a referencia ao propio obxecto.

Neste caso, si nos fixamos ben, vemos que o construtor recibe os argumentos ***nome*** e ***idade*** que como todos os argumentos, son variables locais ao método. O problema e que fora do bloque do método temos definidos os atributos ***nome*** e ***idade***, dificultando a súa inicialización (poñer

nome=nome é absurdo, aínda que nós saibamos que o primeiro ***nome*** corresponde ao atributo e o segundo ao parámetro. Para evitar estas ambigüidades se antepón ***this***, que é o propio obxecto, deste xeito ***this.nome*** está claro que se refire ao atributo.

Tamén podemos utilizar this para abreviar os construtores sobrecargados e facilitar a súa modificación....

```
public class Persoa {
        String nome; // atributo de instancia
        int idade;

        // Constructor sen argumentos
        public Persoa() {
                this("Anonymous");      // Chamamos ao construtor cun unico argumento
        }

        // Constructor sobrecargado
        public Persoa(String nome, int idade) {
                this.nome = nome;
                this.idade = idade;
        }

        // Constructor sobrecargado
        public Persoa(String nome,) {
                this.nome = nome;
                this.idade = 0;
        }

}
```

A Encapsulación

A encapsulación e a práctica de agrupar datos e métodos que operan sobre eses datos nunha soa unidade (a clase). Co obxecto de garantir o correcto funcionamento da clase, se fai necesario ocultar e controlar o acceso tanto os atributos como a os métodos.

Como xa comentamos a orientación a obxectos e como si un xefe en lugar de facer él absolutamente todo o traballo contrata a unha secretaria a un contable e a un administrativo.
O xefe levará a cabo algunhas tarefas que lle son propias, pero o resto as encomendará a secretaria, ao administrativo ou ao contable.
A encapsulación garante que o xefe non pode interferir no traballo dos demais. O contable levará a cabo as actividades encomendadas polo seu xefe relacionadas coa contabilidade de xeito correcto e independente do que suceda nos demais departamentos ou do que faga o xefe.

Para lograr iso, e necesario regular o acceso aos métodos e atributos dos obxectos para que non podan ser manipulados dende fora e producir malos funcionamentos. Se dispón dos seguintes graos de protección:

- **default**: Si non se indica nada as clases, metodos e atributos son accesibles unicamente ***dende dentro do mesmo paquete***
- **public**: As clases, métodos e atributos públicos s**empre son accesibles** directamente
- **private**: As clases, métodos e atributos privados so son accesibles dende a propia clase. Cando opera sobre as clases, so se pode aplicar a clases internas (definidas dentro de outra clase)
- **protected**: As clases, métodos e atributos protected son accesibles **dende dentro do mesmo paquete ou dende unha clase herdada** (xa veremos a herdanza mais adiante)

E importante protexer as clases, atributos e métodos do xeito máis apropiado para garantir o bo comportamento dos obxectos sexan cales sexan as circunstancias.

Ademais tamén asegura que sexa posible modificar o xeito de traballar das clases sen afectar ao resto do programa sempre que se respete o API público (os métodos e atributos públicos)

Cando os atributos non son accesibles directamente se fai necesario o uso de métodos públicos na clase para poñer valores nos atributos e para obtelos. Deste xeito, se pode garantir un acceso controlado aos mesmos (por exemplo impedindo que se coloquen valores inválidos). Estes métodos se coñecen como ***getters*** e ***setters***.

A sobrecarga

A sobrecarga (overloading) permite definir varios métodos ou construtores co mesmo nome pero con diferentes parámetros, ofrecendo múltiples maneiras de executar unha mesma acción segundo as necesidades do programa.

Pódese aplicar tanto a métodos coma a construtores.

Dicímos que un método está **sobrecargado** cando na mesma clase existen **varias versións do mesmo nome de método**, pero:

- O **número de parámetros** é distinto, **ou**
- O **tipo dos parámetros** é distinto, **ou**
- A **orde dos parámetros** é distinta (a orde dos tipos, o nome dos parámetros non importa).

O tipo de retorno e irrelevante cando sobrecargamos os métodos. Simplemente os nomes dos métodos son iguais, pero os tipos dos parámetros varían en número ou orde.

Argumentos variables

```
// Método con argumentos variables. O método recibe o array numeros de tipo int[]
// cos argumentos que se pasan na chamada
public static void imprimirNumeros(int... numeros) {
      System.out.println("Número de elementos: " + numeros.length);
      for (int n : numeros)
         System.out.println(n);
}
```

Cando se chama a un método con `varargs`, Java crea automaticamente un array cos argumentos que lle pases, polo tanto, dentro do método podes tratar o parámetro coma un array normal.

Os test unitarios

A encapsulación permite que mediante test unitarios se garante o correcto funcionamento dos métodos das clases en calquera circunstancia. Un test unitario consiste nun conxunto de datos de proba coidadosamente elixidos que permite comprobar que os métodos públicos realmente fan o que deben. Si a encapsulación está correctamente deseñada, isto garante que os obxectos da clase sempre farán correctamente o seu traballo.

Static

A palabra reservada ***static*** indica que o elemento así definido ***pertence a clase no seu conxunto***, e non a cada obxecto en particular.

Son elementos ***compartidos entre todas as instancias da clase*** (obxectos), e ***existen de forma independente*** aos obxectos creados.

Habitualmente, **accedemos a eles antepoñendo o nome da clase**, sen necesidade de crear unha instancia.

Clases estáticas

As clases estáticas só se poden definir como **clases internas**, é dicir, dentro doutra clase.
Isto permite **crear obxectos da clase interna estática sen necesidade de instanciar a clase externa**, xa que **non dependen dun obxecto externo** para existir.

Atributos estáticos

Os atributos estáticos almacenan información **común a todos os obxectos dunha clase**.
En consecuencia, pódese dicir que representan **un estado global compartido**, accesible sen necesidade de crear instancias.

Métodos estáticos

Os métodos estáticos poden ser chamados **sen necesidade de crear ningunha instancia** da clase.
Dado que a súa existencia **non depende de ningún obxecto concreto**, **só poden acceder directamente a outros membros estáticos da clase**, como atributos ou métodos tamén declarados `static`.

Introdución a Herdanza: Sobreposición de Métodos

A herdanza é unha das chaves da programación orientada a obxectos. Partindo dunha definición dunha clase —xeralmente probada e funcional— pódese definir outra nova para ampliar e / ou modificar o seu comportamento e atributos.

Isto favorece a r***eutilización do código*** (unicamente temos que indicar as diferenzas no comportamento) e permite ***estender a funcionalidade*** sen comprometer outras partes do programa.

A clase orixinal se denomínase ***clase pai*** ou ***clase base***, mentres que a nova clase se coñece como ***clase filla*** ou ***clase derivada***.

`super` é unha palabra reservada en Java, similar a `this`.
Mentres que `this` almacena unha referencia ao **obxecto actual** da propia clase, `super` almacena unha referencia á **clase pai** ou á superclase.

Polo tanto:

- Se en `Can` facemos `super.facerSon()`, estamos chamando ao **método `facerSon()` definido na clase `Animal`**.
- Se facemos `super(nome)` dentro do constructor de `Can`, estamos chamando ao **constructor de `Animal`**.

```java
// Clase base ou clase pai
public class Animal {
    // Atributo común a todos os animais
    protected String nome;

    // Constructor
    public Animal(String nome) {
        this.nome = nome;
    }

    // Método común: emitir son xenérico
    public void facerSon() {
        System.out.println("O animal fai un son.");
    }

    // Método que amosa información do animal
    public void mostrarInfo() {
        System.out.println("Nome: " + nome);
    }
}
```

```java
// Clase derivada ou clase filla
public class Can extends Animal {
    private String raza;  // Novo atributo

    // Constructor da clase filla: chama ao constructor da clase pai con super()
    public Can(String nome, String raza) {
        super(nome); // Chamada ao constructor de Animal
        this.raza = raza;
    }

    // Sobrescritura (override) do método facerSon()
    @Override
    public void facerSon() {
        System.out.println("O can ladra: ¡guau guau!");
    }

    // Novo método específico da clase Can
    public void correr() {
        System.out.println(nome + " está a correr alegremente.");
    }

    // Sobrescritura do método mostrarInfo()
    @Override
    public void mostrarInfo() {
        System.out.println("Nome: " + nome + " | Raza: " + raza);
    }
}
```

```java
// Clase principal para probar a herdanza
public class ProbaHerdanza {
    public static void main(String[] args) {
        Animal a1 = new Animal("Bicho");
        a1.facerSon();      // Chamará ao método de Animal
        a1.mostrarInfo();

        System.out.println("------------------");

        Can c1 = new Can("Trosky", "Pastor alemán");
        c1.facerSon();      // Chamará ao método sobrescrito en Can
        c1.mostrarInfo();   // Mostra tamén a raza
        c1.correr();        // Método exclusivo da clase filla
    }
}
```

A clase Object

Java é unha linguaxe **orientada a obxectos de raíz única e herdanza simple**.

- En Java, **todas as clases herdan directamente ou indirectamente da clase `Object`**, que actúa como **raíz da xerarquía de clases**.

- `Object` é a superclase común de todas as clases, polo que **toda clase, explícita ou implícitamente, herda os seus métodos**, como `toString()`, `equals()` ou `hashCode()`.

- Que Java sexa de **herdanza simple** significa que **unha clase non pode ter máis dunha superclase directa**. Por exemplo, non se pode crear un obxecto que herde ao mesmo tempo dunha clase `Computadora` e dunha clase `Telefono`.
-

A clase `Object` proporciona métodos moi relevantes que herdan **todas as clases definidas en Java**, entre os máis importantes destacan:

Método	Descrición
`toString()`	Devolve unha representación en texto do obxecto. Normalmente convén sobrescribilo para amosar información útil.
`equals(Object obj)`	Compara se dous obxectos son iguais en contido. Por defecto compara referencias, pero pódese sobrescribir para comparar atributos.
`hashCode()`	Devolve un código hash asociado ao obxecto. Úsase en estruturas como `HashMap` ou `HashSet`
`getClass()`	Retorna un obxecto `Class` que describe a clase do obxecto en tempo de execución.
`wait()`, `notify()`, `notifyAll()`	Métodos para sincronización de fíos en programación concorrente.

As consecuencias das referencias: A comparación e a copia

En Java, **todas as clases crean obxectos que se manexan mediante referencias**. Isto significa que cando declaramos unha variable de tipo obxecto, non se almacena directamente o obxecto na memoria da variable, senón **un enderezo (referencia) que apunta ao obxecto real na memoria**.

Este comportamento ten importantes consecuencias cando comparamos obxectos ou cando facemos asignacións.

Comparación de obxectos

O operador `==` **compara o contido das variables.** Se as variables conteñen referencias a obxectos o que se está a comparar son as dúas referencias, que **so serán iguais si apuntan ao mesmo obxecto na memoria,** sin importar o contido dos atributos dos obxectos.

```
class Can {
    String nome;

    Can(String nome) {
        this.nome = nome;
    }
}

public class ProbaReferencias {
    public static void main(String[] args) {
        Can c1 = new Can("Trosky");
        Can c2 = new Can("Trosky");
        Can c3 = c1;

        System.out.println(c1 == c2); // false, diferentes obxectos
        System.out.println(c1 == c3); // true, mesma referencia
    }
}
```

O método `equals()` permite **establecer o noso propio criterio de igualdade**, non simplemente comparar a referencia. Por defecto, a implementación herdada de `Object` é equivalente a `==`, polo que **convén sobrescribir `equals()`** para clases propias.

```
class Can {
    String nome;

    Can(String nome) { this.nome = nome; }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof Can)) return false;
        Can outro = (Can) obj;
        return this.nome.equals(outro.nome);
    }
}

public class ProbaEquals {
    public static void main(String[] args) {
        Can c1 = new Can("Trosky");
        Can c2 = new Can("Trosky");

        System.out.println(c1.equals(c2)); // true, mesmo contido
    }
}
```

Sempre que se sobreescribe equals é unha boa práctica sobrescribir tamén hashCode(). Netbeans facilita moito a sobreposición destes métodos mediante un menú contextual.

Copia de obxectos

O problema da copia é similar ao da comparación. Si facemos unha asignación -- como `c3 = c1;` -- non estamos copiando o obxecto, **so se copia a referencia**. Polo tanto, `c1` e `c3` apuntan ao mesmo obxecto na memoria, e **non se crea ningunha copia real dos datos**.

Para solucionar este problema a mellor solución é o uso dun **construtor de copia**. Un construtor de copia é un construtor que recibe como argumento un obxecto da propia clase e **se encarga de copiar a información aos atributos do novo obxecto** que se está creando.

```
class Can {
    String nome;

    Can(String nome) {
        this.nome = nome;
    }

    // Constructor de copia
    Can(Can outro) {
        this.nome = outro.nome;
    }
}
```

Ao facer a copia, debemos ter en conta que **algúns atributos poden ser obxectos**, como `Array` ou listas, que tamén necesitan **copiarse para crear novos obxectos.** Podemos distinguir entre:

- **Shallow copy (copia superficial):** copia os atributos primitivos e referencias, **pero as referencias internas seguen apuntando aos mesmos obxectos**.
- **Deep copy (copia profunda):** copia **todos os atributos**, creando tamén novos obxectos para as referencias internas, asegurando que a copia é completamente independente do orixinal.

Os obxectos **inmutables**, como `String`, poden ser simplemente asignados usando `=` porque **o seu contido non pode cambiar**.
Isto significa que compartir a mesma instancia entre varias referencias **non representa un problema**, xa que ningunha referencia poderá modificar o obxecto orixinal.

```
class Perro {
   String nome;
   int[] idades;

   Perro(String nome, int[] idades) {
      this.nome = nome;
      this.idades = idades;
   }

   // Constructor de copia
   Perro(Perro outro) {
      this.nome = outro.nome;
      this.idades = outro.idades;
   }
}
```

Neste caso si facemos:

Perro p1=new Perro(“Toby”,{20,30,40});
Perro p2=new Perro(p1)

Non fará unha copia válida, xa que a modificación dunha idade en p2, implica que tamén se modificará p1 (idades de p2 apunta ao mesmo array de p1.
Deberíamos facer algo así:

```
// Constructor de copia profunda
Perro(Perro outro) {
          this.nome = outro.nome;
          this.idades = new int[outro.idades.length];          // Creamos un novo array
          System.arraycopy(outro.idades, 0, this.idades, 0, outro.idades.length);  // Copiamos os elementos
}
```