Herdanza

Sumario

Introdución

Nos puntos anteriores mencionamos varias veces o concepto de herdanza. En Java, unha clase pode derivar doutra e, polo tanto, herdar os seus atributos e os seus métodos.

A unha clase que deriva doutra chámaselle **subclase**. Tamén se lle chama clase derivada, clase estendida ou clase filla. A clase dende a que deriva a subclase chámaselle **superclase**. Tamén se lle chama clase base ou clase pai/nai.

Exceptuando a clase Object, que non ten superclase, en Java toda clase ten unha única superclase directa. A isto chámaselle **herdanza simple**. Se non se especifica unha superclase explicitamente para unha clase determinada derivará da clase Object.

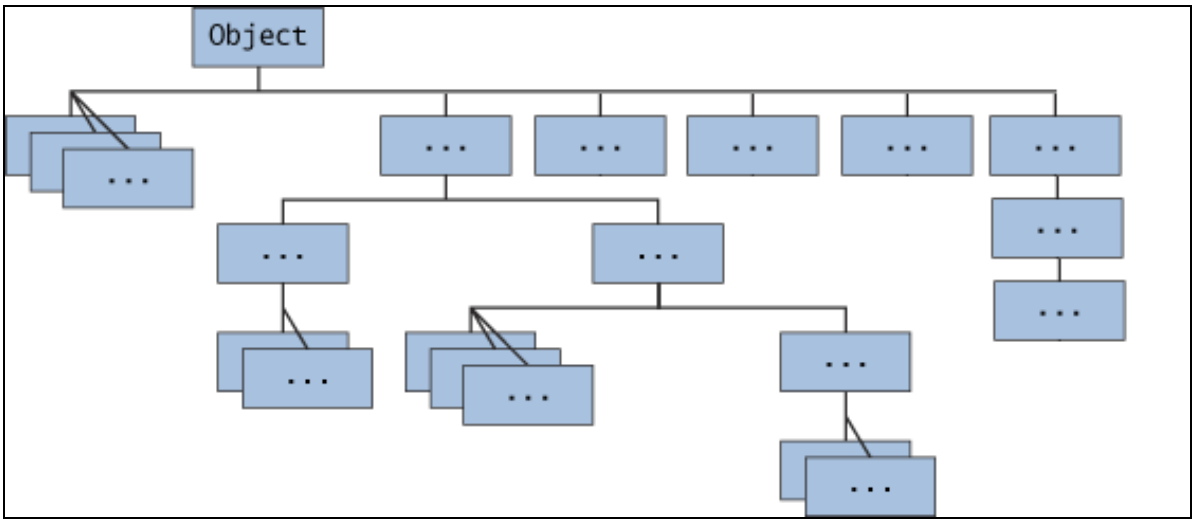
As clases poden derivarse a partir doutras que, á súa vez, derivan doutras clases e así sucesivamente, pero todas as clases en Java derivan, en último caso, da clase Object que é a clase no nivel máis alto da xerarquía.

A idea de herdanza é simple pero moi potente e característica da programación orientada a obxectos: Cando se queira crear unha clase nova e exista outra clase que xa inclúa código que queremos usar podemos derivar desa clase. Este é un xeito de reutilizar os atributos e métodos dunha clase existente sen ter que escribila (e depurala) un mesmo de novo, dende cero.

Unha subclase herda todos os atributos e métodos da superclase pero no os construtores. Non entanto, o construtor da superclase pode invocarse dende a subclase.

A xerarquía de clases en Java

A clase Object, definida no paquete java.lang, define e implementa comportamento común a todas as clases, incluídas as que ti escribes (recorda que o comportamento dunha clase impleméntase a través dos seus métodos). Na API de Java hai moitas clases que implementan distintos comportamentos. Moitas derivan directamente de Object e outras non, formando o que se coñece como unha **xerarquía de clases**.



Na parte máis alta da xerarquía, Object é a clase máis xeral de todas. As clases próximas á parte baixa da xerarquía proporcionan un comportamento máis especializado. Canto máis abaixo, maior especialización.

Exemplo de herdanza

Creamos a clase Empregado cos seus datos, e queremos crear a clase Secretaria, que terá todos os datos de Empregado e uns datos específicos. Pode dicirse que Secretaria é un empregado con características adicionais:

```
public class Empregado {
  String nome;
  Date inicioTraballo;
  Date dataNacemento;
  String titulacion;
 }

 public class Secretaria {
  String nome;
  Date inicioTraballo;
  Date dataNacemento;
  String titulacion;
  String telefonoMobil;
 }
```

Ao definir Secretaria duplícanse os datos que xa tiñamos incluídos en Empregado! Se usamos a herdanza solucionamos este problema. É dicir, se logo de crear unha clase se necesita unha versión máis especializada dela utilízase a herdanza. Para elo úsase a palabra reservada extends, tal e como segue:

```
public class Secretaria extends Empregado {
   String telefonoMobil;
 }
```

A clase Secretaria é unha subclase de Empregado e herda dela todos os métodos e características (atributos), excepto o construtor. A herdanza é especialmente interesante e potente se os métodos na clase Empregado fosen complexos e levara moito tempo escribilos e depuralos, xa que nos aforraría moitísimo traballo.

Que se pode facer nunha subclase

Unha subclase herda todos os atributos e métodos **public** e **protected** da superclase, independentemente de se está ou non no mesmo paquete. Na subclase pódense usar os atributos e métodos tal e como se herdan, reemprazalos, ocultalos ou definir novos métodos e atributos particulares desa subclase específica. En concreto, nunha subclase podemos facer o seguinte:

- ◊ Os atributos herdados pódense usar directamente, como se fosen atributos propios.
- ◊ Pódese declarar un atributo na subclase co mesmo nome que o da superclase. A isto chámaselle **ocultar un atributo** e non é recomendable, como veremos máis adiante.
- ◊ Pódense declarar novos atributos na subclase que non están na superclase.
- ◊ Os métodos herdados tamén se poden usar directamente.
- ◊ Pódese escribir un novo método na subclase que teña a mesma sinatura que un da superclase. A isto chámaselle **sobreescribir**

un método.

- ◊ Pódese escribir un método static na subclase que teña a mesma sinatura que un da superclase. A isto chámaselle **ocultar un método**.
- ◊ Pódense declarar novos métodos na subclase que non estean na superclase.
- ◊ Pódese escribir un construtor para a subclase que invoque ao construtor da superclase, implicitamente ou usando a palabra reservada super.

Nas seguintes seccións desta unidade afondaremos en como facer todo isto e veremos para que vale.

Atributos e métodos privados da superclase

Unha subclase non herda os atributos e métodos privados da súa clase pai/nai. Non entanto, se a superclase ten métodos public ou protected para acceder aos seus atributos privados a subclase pode usalos.

O operador instanceof

Xa sabemos que un obxecto é do tipo de dato da clase dende o que se instanciou. Por exemplo, se escribimos:

```
public Secretaria unhaSecretaria = new Secretaria();
```

Entón unhaSecretaria é do tipo Secretaria.

A clase Secretaria deriva directamente de Empregado e indirectamente da clase Object. Polo tanto, unha Secretaria é un Empregado e tamén un Object. Isto quere dicir que a clase Secretaria pode usarse en calquera lugar onde se poda usar a clase Empregado ou a clase Object. A isto chámaselle **casting** de obxectos.

Á inversa non é necesariamente certo: un Empregado pode ser unha Secretaria ou pode ser, por exemplo, un Xefe. Analogamente, un Object pode ser un Empregado ou unha Secretaria, pero non necesariamente.

O casting proporciona a posibilidade de usar un obxecto dun tipo en lugar doutro, sempre que se faga entre obxectos que o permitan segundo a herdanza.

Isto pode crear confusión á hora de saber con que tipo de obxecto estamos a traballar nun programa. O operador instanceof permite comprobar o tipo dun obxecto particular como vemos no seguinte exemplo:

```
public void metodo (Empregado e) {
  if (e instanceof Secretaria) {
    ...
  } else if (e instanceof Xefe) {
    ...
  } else {
      // empregado normal
  }
 }
```

Redefinición de métodos

Cando creamos unha subclase quereremos, en moito casos, herdar o seu comportamento (métodos) tal e como están, na superclase. Noutras situacións quereremos modificar ese comportamento. Por exemplo, podemos ter unha superclase chamada Coche cun método xeral que sirva para todos os coches que se chame acelera(), pero unha subclase Ferrari seguramente sobrescriba o método acelera().

Para modificar o comportamento dunha superclase basta con escribir un método na subclase co mesmo nome que o método que queiramos modificar da superclase. Asemade, deberá ter o mesmo tipo de retorno e a mesma lista de parámetros que o método da superclase.

Se modificamos métodos de instancia falamos de sobrescritura. Se modificamos métodos static (de clase) falamos de ocultación.

Sobrescritura

A sobrescritura de métodos permite a unha clase modificar parte do comportamento que herda dunha superclase. Un método nunha subclase coa mesma sinatura (nome, número e tipo de parámetros), e que devolva o mesmo tipo de dato que un método na superclase, **sobrescribe** o método da superclase.

Cando sobrescribimos un método podemos usar unha etiqueta (anotación) especial, @Override, que lle indica ao compilador que estamos tentando sobrescribir un método da superclase. Se por algunha razón o compilador detecta que o método non existe na superclase devolverá un erro.

Vexamos un exemplo con código:

```
public class Animal {
   public void come() {
      System.out.println("Método come da superclase Animal: un animal pode comer de todo.");
   }
 }

 class Cabalo extends Animal {
   public void come() {
       System.out.println("Método comer da subclase Cabalo: un cabalo come herba.");
   }
 }
```

Ocultación

Se unha subclase define un método static coa mesma sinatura que un método static da superclase, o método da subclase **oculta** ao método da superclase.

A distinción entre ocultación e sobrescritura de métodos ten implicacións importantes. Cando sobrescribimos un método, o método que se invoca é o da subclase. Cando ocultamos un método, o método que se invoca pode ser da subclase ou da superclase xa que é static. Vexámolo cun exemplo con dúas clases. A primeira é a clase Animal, que contén un método static e un método de instancia:

```
class Animal {
        public static void probaMetodoStatic() {
            System.out.println("Método static da clase Animal.");
        }
        public void probaMetodoInstancia() {
            System.out.println("Método de instancia na clase Animal.");
        }
    }
```

A segunda clase é a clase Gato, subclase de Animal:

```
class Gato extends Animal {
         // Ocultación
        public static void probaMetodoStatic() {
            System.out.println("O método static da subclase Gato.");
        }
        // Sobreescritura
        public void probaMetodoInstancia() {
            System.out.println("O método de instancia na subclase Gato.");
        }
  }

  class Principal {
        public static void main(String[] args) {
            Gato unGato = new Gato();
            Animal unAnimal = new Animal();
            unGato.probaMetodoInstancia();
            unGato.probaMetodoStatic();
            Animal.probaMetodoStatic()
        }
    }
```

A clase Gato sobrescribe o método de instancia da clase Animal e oculta o método static de Animal.

A saída do programa é

```
O método de instancia na subclase Gato.
O método static da subclase Gato.
Método static da clase Animal.
```

A seguinte táboa resume que acontece cando se define un método coa mesma sinatura que un método na superclase.

	Método de instancia da superclase	Método static da superclase
Método de instancia da subclase	Sobrescribe	Erro de compilación
Método static da subclase	Erro de compilación	Oculta

Modificadores de acceso

Cando se redefine un método hai que ter en conta que o método da subclase non pode ser menos accesible que o da superclase. Por exemplo, un método protected na superclase pode converterse en public na subclase pero non en private.

Redefinición de atributos

Dentro dunha clase, un atributo que teña o mesmo nome que un atributo da superclase oculta o atributo da superclase, incluso se os seus tipos son distintos. Dentro da sublcase non se pode acceder ao atributo da superclase directamente polo seu nome, senón que se terá que usar a palabra reservada **super**, que veremos na próxima sección. En xeral, non é recomendable ocultar atributos xa que fan o código difícil de ler.

A palabra reservada super

Acceso aos atributos e métodos da superclase

Se un método sobrescribe outro método da superclase pódese invocar ao método sobrescrito usando a palabra reservada super. Tamén se pode usar super para acceder a atributos ocultos (aínda que o uso de atributos ocultos non se aconsella). Vexámolo cun exemplo. Temos unha clase chamada Superclase:

```
public class Superclase {
        public void metodoImprimir() {
            System.out.println("Metodo imprimir da Superclase");
        }
    }
```

E unha subclase chamada Subclase que sobrescribe o método metodoImprimir():

```
public class Subclase extends Superclase {
        public void metodoImprimir { //Sobrescribe metodoImprimir da Superclase
            super.metodoImprimir();
            System.out.println("Metodo imprimir da Subclase");
        }
        public static void main(String[] args) {

        Subclase s = new Subclase();
        s.metodoImprimir();
        }
    }
```

Dentro de Subclase, metodoImprimir() refírese ao método declarado en Subclase, que sobrescribe o da Superclase. Para referirnos ao metodoImprimir() herdado da Superclase, Subclase ten que usar a palabra super, tal e como vemos no exemplo anterior. A compilación e execución de Subclase imprime o seguinte:

```
   Metodo Imprimir da Superclase.
   Metodo Imprimir da Subclase.
```

Con esta técnica podemos ampliar a funcionalidade dun método herdado.

Acceso ao construtor da superclase

O seguinte exemplo amosa como se invoca a un construtor dunha superclase coa palabra super(). Para elo creamos unha clase Estudante cun atributo nome, e unha subclase EstudanteBecario cun atributo salario.

```
class Estudante {
    private String nome;
    public String leNome() {
      return nome;
    }
```

```
    public void escribeNome(String nome) {
      this.nome = nome;
    }
    public Estudante (String nome) {
      this.nome = nome;
    }
}

class EstudanteBecario extends Estudante  {
    private int salario;
    public EstudanteBecario (String s,int i){
        super(s);
        salario = i;
     }
    public int leSalario(){
        return salario ;
      }
    public void escribeSalario(int cartos){
        salario = cartos;
     }
}

class EstudanteDemo {
    public static void main(String []args) {
      EstudanteBecario e = new EstudanteBecario("Manuel", 6000);
      System.out.println(e.leSalario() + " " + e.leNome());
    }
}
```

Se nos fixamos, o construtor de EstudanteBecario ten unha instrución super(s) que o que fai é invocar ao construtor da superclase, neste caso, Estudante. A saída do programa será:

```
6000 Manuel
```

A invocación dun construtor dunha superclase debe facerse na primeira liña do construtor da subclase. A sintaxe para chamar a un construtor dunha superclase é:

```
super();
    --ou--
    super(lista parámetros);
```

Con super() chámase ao contrutor da superclase sen argumentos. Con super(lista parámetros) chámase ao construtor da superclase cunha lista de parámetros.

Se un construtor non invoca explicitamente a un construtor da superclase o compilador Java automaticamente insire unha chamada ao construtor sen argumentos da superclase. Se a superclase non ten un construtor sen argumentos o compilador dará un erro. A clase Object da API de Java ten un construtor sen argumentos, así que se Object é a única superclase non haberá problema. Polo tanto, irán executándose os construtores de todas as superclases dentro da xerarquía até chegar á clase máis xeral, a clase Object. A isto chámaselle **cadea de construtores** e hai que telo en conta cando hai unha liña longa de descendencia.

A clase Object de Java

A clase Object, do paquete java.lang, está na raíz da xerarquía de clases, é dicir, na parte máis alta. Toda clase en Java deriva, directa ou indirectamente, da clase Object. Cada clase que se use ou se escriba nun programa herda os métodos de instancia da clase Object. Non é obrigatorio usar estes métodos pero se se usan pódense sobrescribir con código específico para as nosas necesidades. Nas seguintes seccións imos ver algúns métodos interesantes da clase Object:

- `public boolean equals(Object obj)`

  Indica se un obxecto é igual a este.

- `protected void finalize() throws Throwable`

  Invócao o recolector de lixo cando determina que non existen referencias a un obxecto.

- `public final Class getClass()`

  Devolve a clase á que pertence un obxecto en tempo de execución.

- `public int hashCode()`

  Devolve o valor do código hash do obxecto.

- `public String toString()`

  Devolve a representación en formato cadea dun obxecto.

O método equals()

Ao falar dos tipos primitivos, vimos que o operador == permite saber se os valores das variables que se comparan son iguais ou non. Isto non serve para comparar obxectos xa que o operador == retorna true se as **referencias** de ambos obxectos son iguais, pero non se os contidos dos obxectos son iguais. Se queremos comparar dous obxectos polo seu contido hai que sobrescribir o método equals ou facer un método novo. Por exemplo, dous libros que teñen o mesmo ISBN son o mesmo obxecto no noso programa, aínda que teñan referncias distintas:

```
class Libro {
        public String ISBN ;
        public boolean iguais (Libro l) {
            return (this.ISBN == l.ISBN);
        }
        public Libro(String ISBN) {
          this.ISBN = ISBN;
        }
  public static void main (String []arg) {
    Libro primeiroLibro  = new Libro("0201914670");
    Libro segundoLibro = new Libro("0201914670");
    if (primeiroLibro.iguais(segundoLibro)) {
      System.out.println("Os obxectos son iguais");
    }
    else {
      System.out.println("Os obxectos non son iguais");
    }
    if (primeiroLibro.equals(segundoLibro)) {
      System.out.println("Con equals: os obxectos son iguais");
    }
    else {
      System.out.println("Con equals: os obxectos non son iguais");
    }
   }
 }
```

O método finalize()

O sistema chama a este método automaticamente cando un obxecto non ten referencias a el. Con todo, nun determinado momento pode interesarnos invocalo explicitamente para liberar recursos. Por exemplo, na E/S con ficheiros, se non pechamos os descritores dos mesmos podemos invocar a finalize() para que os peche automaticamente.

O método getClass()

Este método non se pode sobrescribir. Devolve un obxecto de tipo Class co que se pode obter información sobre a clase dun obxecto como, por exemplo, o seu nome con getSimpleName(), a súa superclase con getSuperclass(), etc. Por exemplo, o seguinte método devolve o nome da clase dun obxecto:

```
void imprimeNomeClase(Object obj) {
    System.out.println("O nome da clase é: " + obj.getClass().getSimpleName());
 }
```

A clase Class ten máis de 50 métodos distintos.

O método hashCode()

Este método devolve a dirección de memoria en hexadecimal dun obxecto. Dous obxectos serán iguais se teñen o mesmo código hash.

O método toString()

Devolve unha representación en formato cadea dun obxecto. É moi útil para depurar programas. Pódese usar directamente con System.out.println() para amosar información dun obxecto.

Clases e métodos finais

O palabra reservada **final** utilízase para impedir que unha clase se poida modificar. Ao declarar unha clase como final estamos indicando que se atopa ao final da xerarquía de clases e iso impide que poida herdarse. Por exemplo, a clase java.lang.String, que proporciona a API de Java, é unha clase final e, polo tanto, non se pode modificar.

Os métodos tamén poden definirse como final. Os métodos coa palabra reservada final non se poden sobrescribir ou redefinir. Declarar un método como final pode ser útil cando a implementación dese método non se deba cambiar porque, ao facelo, por exemplo, ponse en perigo a consistencia da clase. Os métodos static e private son final automaticamente. Vexamos un exemplo:

```
class Xadrez {
   enum Xogador { BRANCAS, NEGRAS }
   final Xogador dimePrimeiroXogador() {
       return Xogador.BRANCAS;
   }
   public static void main(String []arg) {
      Xadrez unXadrez = new Xadrez();
      System.out.println(unXadrez.dimePrimeiroXogador());
   }
 }
```

No caso das variables, se se declaran como final convértense en constantes.

Clases e métodos abstractos

A palabra reservada **abstract** úsase para declarar métodos ou clases abstractas. Un método abstracto é un método que se declara e non ten código, é dicir, non se proporciona a implementación del. Por exemplo:

```
abstract void moverA(double deltaX, double deltaY);
```

Se unha clase ten algún método declarado como abstract, a propia clase ten que ser declarada como abstract. Con todo, unha clase abstracta pode ter métodos non abstractos, tal e como vemos no seguinte exemplo:

```
abstract class Persoa {
   // Declaración de atributos
  String nome;
   // Declaración de métodos abstractos
  abstract void listaSalario();
   // Declaración de métodos non abstractos
  public String dimeNome() {
    return nome;
  }
 }
```

Se creamos unha subclase a partir dunha clase abstract a subclase ten que implementar os métodos abstractos da superclase. Se non o fai tamén terá que declarase como abstracta.

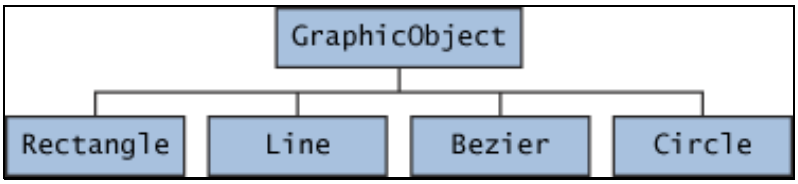
Unha clase abstracta non se pode instanciar. Por outra parte, non se pode declarar unha clase como abstract e final ao mesmo tempo xa que se estaría obrigando e pohibindo a herdanza desa clase.

As clases abstractas proporcionan un mecanismo moi potente para facilitar o deseño e programación orientada a obxectos, xa que podemos deseñar aplicacións que conteñan unha serie de clases abstractas e codificalas sen entrar na definicición dos detalles do código dos métodos. A aplicación

queda desta maneira ben estruturada, definida e consistente (podemos compilala). A partir deste punto de partida, resulta moito máis sinxela a fase de implementación que pode levarse a cabo en paralelo por diversos programadores, coñecendo cada un dos obxectos que te que modificar e as clases relacionadas que pode empregar.

Un exemplo de clase abstracta

Nunha aplicación de debuxo podemos traballar con círculos, rectángulos, liñas, curvas Bezier e moitos outros obxectos gráficos. Estes obxectos teñen certos atributos (posición, orientación, cor da liña, cor de recheo, etc.) e métodos en común (moverA, xirar, redimensionar, debuxar, etc.). Algúns destes métodos e atributos son iguais para todos os obxectos gráficos. Por exemplo: posición, cor de recheo e moverA. Con todo, outros poden requirir implementacións distintas segundo o tipo de obxecto gráfico do que esteamos falando. Por exemplo: redimensionar ou xirar (non é o mesmo xirar unha liña que un círculo). Todos os obxectos gráficos teñen que poder xirarse pero cada método impleméntase dun xeito distinto. Polo tanto, podemos aproveitar as similutedes e declarar todos os obxectos gráficos para que herden da mesma superclase abstracta (por exemplo, ObxectoGrafico), tal e como vemos na figura:



Declaramos unha clase abstracta GraphicObject, para proporcionar atributos e métodos que comparten todas as subclases (posición, moverA, etc.). GraphicObject declara tamén métodos abstractos, como debuxar ou redimensionar, que precisan implementarse en todas as subclases, pero **todas as subclases teñen que implementalos dun xeito distinto**. A clase GraphicObject podería ser similar á seguinte:

```
abstract class GraphicObject {
        int x, y;
        ...
        void moverA(int newX, int newY) {
            ...
        }
        abstract void debuxar();
        abstract void redimensionar();
    }
```

Cada subclase de GraphicObject, como Circle e Rectangle, debe proporcionar implementacións para os métodos debuxar e redimensionar:

```
class Circle extends GraphicObject {
        void debuxar() {
            ...
        }
        void redimensionar() {
            ...
        }
    }
    class Rectangle extends GraphicObject {
        void debuxar() {
            ...
        }
        void redimensionar() {
            ...
        }
    }
```