Sobreposición, Sobrecarga e Polimorfismo

Como xa se explicou en "***Introducción á Programación Orientada a Obxectos II***":

- A ***Sobrecarga*** consiste na realización de distintas versións dun mesmo método, diferindo no número e/ou tipos dos argumentos.
- A ***Sobreposición*** consiste en escribir nunha clase heredada unha nova versión dun método xa existente na clase base.
- O ***Polimorfismo*** indica que un obxecto pode ser tratado como un elemento da clase pai, sen variar o seu comportamento (estaremos restrinxidos aos métodos e atributos definidos na clase pai, pero o comportamento será o especificado na propia clase).

Tratamento de Erros: Excepcións

Como xa se explicou en "***Introducción á Programación Orientada a Obxectos II***":

- As excepcións son un modo de xestionar os posibles erros dunha aplicación, "lanzando" o erro de xeito que deba ser capturado e xestionado.
- En Java existen dúas familias de excepcións: ***Checked*** e ***Unchecked***. As "checked" exceptions se verifica que son tratadas no momento da compilación, o que obliga a codificar un tratamento de erro. As "unchecked" exceptions se verifican en tempo de execución, polo que si non tratamos o erro o programa compila igualmente.
- Cando se lanza unha excepción temos dúas posibilidades: Ou a tratamos mediante un bloque try {} catch() {} ou a relanzamos na cabeceira do método. En Java utilizamos ***throws*** para indicar que é posible que o método lance unha excepción.

Exemplo en Java

```
class Ec2Grao {
private static final double error_sqrt_max=0.000001;
private double a;
private double b;
private double c;

// Constructor
public Ec2Grao(double tx2,double tx,double i) {
        a=tx2;
        b=tx;
        c=i;
}

// Método que soluciona a ecuación. Non trata as posibles excepcións, se non que as relanza
// Si quixera tratar as excepcións, poñería o código do método entre try {} catch(ArithmeticException e) {}
//      Lanzo un erro "xenérico" (Exception).... o polimorfismo permite capturar en main as excepcións apropiadas
public double[] soluciona() throws Exception {
        double[] solucions;
        double radicando;

        if (a==0) {
                solucions=new double[1];                // Unha solución, array de 1 elemento
                solucions[0]=ec1grao(b,c);              // Si produce unha excepción, a relanzo..... non a trato aquí
        } else {
                solucions=new double[2];                // dúas solucións, array de 2 elementos
                radicando=squareRoot(b*b-4*a*c);        // Si produce unha excepción a relanzo... non a trato aquí
                solucions[0]=(-b+radicando)/(2*a);
                solucions[1]=(-b-radicando)/(2*a);
        }
        return solucions;
```

```
}

// Métodos auxiliares ----- son privados, non forman parte do Interface da clase, ou sexa, son simples funcións
//
// Calcula unha raíz cadrada polo algoritmo babilónico. Si non é posible, lanza o erro
private double squareRoot(double radix) throws ArithmeticException {
        double b;
        double a;

        if (radix<0) throw new ArithmeticException("Raiz dun número negativo");
        b=radix;
        a=radix/b;
        while(b > a+error_sqrt_max) {
                b=(a+b)/2;
                a=radix/b;
        }
        return b;
}

   // Soluciona a ecuación de 1º grao, si é posible. Si non é posible lanza o erro
private double ec1grao(double b,double c) throws ArithmeticException {
        String str_error;
        if (b==0) {
                if (c==0) str_error="Infinitas Solucións";
                else      str_error="Sin Solucións";
                throw new ArithmeticException(str_error);
        }
        return -c/b;
}
}
public class ExampleExceptions {
//  Método Principal
//        Probar as seguintes execucións:
//                java ExampleException
//                java ExampleException test forzar fallo
//                java ExampleException 0    0 8
//                java ExampleException 0 0 0
//                java ExampleException 0 4 -12
//                java ExampleException 1 2 10
//                java ExampleException 2 5 2.25
public static void main(String args[]) {
  double[] s;
  Ec2Grao ec;

  try {
    if (args.length != 3)  { // Si non especificamos os parámetros producimos un erro
        throw new Exception("Uso: java ExampleException  cX²  cX  termoIn");
    }
    ec=new Ec2Grao(Double.parseDouble(args[0]),Double.parseDouble(args[1]),Double.parseDouble(args[2]));
    s=ec.soluciona();          // Aquí pode producirse unha Exception
    if (s.length == 1) {
        System.out.printf("E unha ecuación de 1º grao %sx+%s=0, con solución: %f\n",args[1],args[2],s[0]);
    } else {
        System.out.printf("As solucións da ecuación de 2º grao %sx²+%sx+%s=0 son %f e %f\n",args[0],args[1],args[2],s[0],s[1]);
    }
  } catch(ArithmeticException e) { // Gracias ao polimorfismo, aquí virán as ArithmeticException que lanzamos en Ec2Grao
        System.out.println("ERROR Aritmético: "+e.getMessage());
  } catch(NumberFormatException e) { // Esta excepcion pode producirse ao fallar a conversión de string a número
        System.out.println("Number conversion failure "+e.getMessage());
  } catch(Exception e) { // Outra Excepción calqueira que se poida producir.... como a que lanzo cos parámetros erróneos
        System.out.println(e.getMessage());
  } finally { // Execución obrigatoria ao final do método
        // Isto se executa ao final SEMPRE, aínda que se produza un erro (excepcion) antes
        System.out.println("O programa rematou!!!");
  }
    // Isto se executa sempre ANTES do finally, sempre que non se produza antes unha excepción
      System.out.println("Finalizando aplicación....");
 }
}
```

Arrays

Os arrays **son obxectos** que conteñen un conxunto de elementos do mesmo tipo aos que podemos acceder mediante un ou varios índices. En Java, o primeiro elemento ten como índice 0.

Os arrays cun so índice se chaman tamén *vectores*, e a os de dous índices se lles coñece como *matrices*.

En Java, as variables capaces de almacenar arrays se definen do seguinte xeito:

tipo[] identificador;

Por exemplo:

int[] edades;

ContaCorrente[] contas;

O primeiro caso declara unha variable capaz de gardar a referencia a un array de 1 índice que almacenará valores **int**. No segundo caso se define unha variable capaz de gardar a referencia a un array de 1 índice que será capaz de almacenar referencias a obxectos de tipo *ContaCorrente*.

Para crear os obxectos array, coma sempre, se utiliza o operador ***new***:

edades=new int[30]; *// Crea un Array de 30 int e almacena a referencia en edades*

// Crea un Array capaz de almacenar 30 ContaCorrente e almacena a referencia en contas
contas=new ContaCorrente[30];

No segundo caso será necesario instanciar os obxectos ContaCorrente que desexe almacenar no array. O primeiro caso, como int é un tipo primitivo e non unha clase, xa podemos almacenar os números enteiros que queiramos nos distintos elementos.

Por comodidade, tamén se pode instanciar un array na propia declaración:

// Array capaz de almacenar 4 ContaCorrente, inicialmente apuntan a null (non apuntan a ningunha ContaCorrente)
ContaCorrente[] contas={ null, null, null, null }
// Isto é equivalente o anterior
ContaCorrente[] contas=new ContaCorrente[4];

// Array capaz de almacenar 4 ContaCorrente, inicialmente o elemento 1 apunta a un obxecto ContaCorrente
// o resto apuntan a null (non apuntan a ningunha ContaCorrente)
ContaCorrente[] contas={ null, new ContaCorrente(), null, null }

Para realizar tarefas comúns cos Array, a libraría de clases de Java suministra unha clase con métodos estáticos chamada Arrays. Mediante esta clase se poden facer ordeacións, buscas, inicializacións, mezclas..... etc. Para máis información consultar a API da clase.

Clases e Referencias

Como xa se explicou anteriormente, en Java cando definimos unha variable para almacenar un obxecto, o que creamos é un lugar onde podemos almacenar a referencia a un obxecto, que debemos crear posteriormente:

ContaCorrente cc; *// NON temos un obxecto ContaCorrente, so un sitio onde gardar unha referencia*

cc=new ContaCorrente(); *// CREAMOS un obxecto ContaCorrente e almacenamos a súa referencia en cc*

Este xeito de traballo ten como consecuencias importantes:

1. Non se poden comparar obxectos cos operadores de comparación (==, >, < ... etc), xa que o que almacenamos nas variables é unha referencia. O seguinte código se compila e executa correctamente, pero non funciona como parece....

```
ContaCorrente c1=new ContaCorrente();
ContaCorrente c2=new ContaCorrente();
if (c1 == c2) {
        System.out.println("Son iguais");
}
```

   A comparación c1 == c2 é certa si c1 almacena a mesma referencia que c2, e decir, si c1 e c2 *son o mesmo obxecto*. Non determina si c1 "é igual" que c2. Debemos ter en conta ademais que o concepto de que dous obxectos "son iguais" é subxectivo. ¿Cando son iguais dous Vehículos? ¿Si coincide a marca? ¿Si coincide a marca e o modelo? ¿O ano de fabricación?..... ***Para comparar dous obxectos, necesitamos métodos de comparación***.

2. Non se pode "copiar" un obxecto mediante o operador de asignación (=).

```
ContaCorrente c1=new ContaCorrente();
ContaCorrente c2=new ContaCorrente();
c1=c2;
```

   Con c1=c2 non obtemos en c1 unha "copia" de c2. Simplemente c1 apuntará ao mesmo obxecto que c2, xa que copiamos a referencia (que é o que almacenan tanto c1 coma c2). Calquera modificación do obxecto apuntado por c2 altera o obxecto apuntado por c1, xa que son o mesmo. ***Para copiar un obxecto, necesitamos métodos de copia***

3. Debemos ser conscientes do efecto de pasar unha variable que referencia a un obxecto como parámetro a un método:

```
class Banco {
   private ContaCorrente c;

   public Banco(ContaCorrente novacc) {
       c=novacc;  // Aqui copia a referencia almacenada en novacc en c NON O OBXECTO
   }
}

class App {
   public static void main(String[] apps) {
     ContaCorrente cc=new ContaCorrente();
     Banco b=new Banco(cc);

     cc.setCliente("Pepito"); // Se modifica o atributo privado de Banco !!!!!!
   }
}
```

   Para evitar romper a protección do atributo ***c*** sería necesario *facer unha copia* de **novacc** no constructor de Banco, e gardar a copia no atributo. E para facer a copia, necesitamos métodos de copia.

Equals e Constructores de Copia

Vistas as consecuencias de tratar os obxectos como referencias na sección anterior, vexamos as solucións adoptadas:

Respecto á comparación, a clase Object ten un método deseñado para determiñar si dous obxectos son iguais, denominado ***equals***. Este obxecto fai unha comparación xenérica, devolvendo ***true*** si os obxectos son iguais, e ***false*** si non o son.

public boolean equals(Object obj);

Si leemos a documentación no API de Object, veremos que además das conclusións obvias:

- Si temos unha referencia x non nula, x.equals(null) debe retornar false
- A implementación por defecto de Object devolve true únicamente si as dúas referencias son as mesmas (facendo o mesmo que x==y).

Algunhas clases de Java teñen sobreposto este método para que se comporte do xeito apropiado, como String. Nas nosas clases será necesario sobrepoñelo do xeito apropiado.

De modo xeral a implementación de equals debería:

- Comprobar que o parámetro non é nulo. Si é nulo, retornar false
- Si o parámetro é o mesmo obxecto (a mesma referencia) que o actual, retornar true
- Comprobar que o parámetro e da mesma clase que estamos tratando, se non, retornar false
- Realizar a comparación para determiñar si os dous obxectos se poden considerar “iguais”. Esto é subxectivo e depende do tipo de obxectos que se comparen.

Exemplo:

Equals que determina si dúas contas bancarias son iguais

```
class ContaBancaria {
        private String titular;
        private String num_cc;

        // Constructor
        public ContaBancaria(String n_cliente,String cc) {
            // Non importa que titular e n_cliente apunten ao Mesmo obxecto String, xa que non se poden cambiar
            titular=n_cliente;
            num_cc=cc;
        }

        // Determina si dúas ContaBancaria son iguais. Imos supoñer que son iguais si os seus num_cc coinciden.
        // Sobrepoñemos o método equals de Object. Obj realmente é da clase ContaBancaria, que tamén é un Object
        // (polimorfismo)
        public boolean Object(Object obj) {
                if (obj==null) return false;
                if (this==obj) return true; // this almacena a referencia ao obxecto actual
                if (! (obj  instanceof ContaBancaria)) return false;
                // Quero traballar con obj como ContaBancaria, para poder usar seus atributos. (Casting)
                // O podo facer porque obj REALMENTE é un obxecto ContaBancaria, si non fora así lanzaría
                // unha unchecked exception
                ContaBancaria cb_obj=(ContaBancaria)  obj;
                return  num_cc.equals(cb_obj.num_cc);   // Iguais si os seus num_cc son iguais
        }
}
```

A **copia de obxectos** pode facerse de varios xmodos, sendo o máis recomendado actualmente o uso de **constructores de copia**. Un constructor de copia é un constructor que recibe como argumento un obxecto da propia clase e copia os seus atributos, creando así un obxecto igual que o recibido como argumento:

```
class ContaBancaria {
        private String titular;
        private String num_cc;

        // Constructor
        public ContaBancaria(String n_cliente,String cc) {
            // Non importa que titular e n_cliente apunten ao Mesmo obxecto String, xa que non se poden cambiar
            titular=n_cliente;
            num_cc=cc;
        }

        // Constructor de copia
        public ContaBancaria(ContaBancaria cb) {
                titular=new String(cb.titular);  // A clase String xa ten un constructor de copia
                num_cc=new String(cb.num_cc);
        }

        // Determina si dúas ContaBancaria son iguais. Imos supoñer que son iguais si os seus num_cc coinciden.
        // Sobrepoñemos o método equals de Object. Obj realmente é da clase ContaBancaria, que tamén é un Object
        // (polimorfismo)
        public boolean Object(Object obj) {
                if (obj==null) return false;
                if (this==obj) return true; // this almacena a referencia ao obxecto actual
                if (! (obj  instanceof ContaBancaria)) return false;
                // Quero traballar con obj como ContaBancaria, para poder usar seus atributos. (Casting)
                // O podo facer porque obj REALMENTE é un obxecto ContaBancaria, si non fora así lanzaría
                // unha unchecked exception
                ContaBancaria cb_obj=(ContaBancaria)  obj;
                return  num_cc.equals(cb_obj.num_cc);   // Iguais si os seus num_cc son iguais
        }
}

class App {
        public static void main(String[] args) {
                ContaBancaria cb1=new ContaBancaria("María","445455445454");
                ContaBancaria cb2=new ContaBancaria(cb1);  // Crea unha copia
        }
}
```

Normas de Estilo

Cando se codifica en unha linguaxe de programación convén seguir certas normas se estilo que normalmente establece a empresa de desenvolvemento. Sen embargo, en Java existen unhas normas que sigue práticcamente todos os programas:

- Os nomes das clases comezan con maiúscula, **SEMPRE**
- Os nomes dos métodos e atributos comezan con minúscula **SEMPRE**
- As constantes normalmente se representan *en maiúsculas*
- Si un identificador (de método ou atributo) e un nome composto, comeza por minúscula e os demáis nomes en maiúscula (por exemplo: *nomeCliente*)