# Encapsulación e Ocultación

A encapsulación consiste en ocultar a implementación da clase, facendo como si fora unha caixa negra que únicamente expón a súa funcionalidade e se pode utilizar a través do seu ***interfaz*** (o conxunto de métodos utilizables dende fora da clase). Para conseguir isto é necesario poder protexer ou ocultar as partes da clase (métodos e atributos) que non queremos que sexan accesibles exteriormente.

De particular importancia é a ocultación dos atributos para evitar que o usuario da clase poda cambiar o estado do obxecto de xeito imprevisto e incontrolado. Para conseguilo se acude á ocultación dos atributos e métodos mediane os modificadores de acceso:

- **public** : O acceso sempre se permite
- **protected** : O acceso so se permite dende a propia clase ou dende unha clase derivada (herdada)
- **private**: O acceso so se permite dende a propia clase.

# Composición, Agregación e Herdanza

Unha das máis poderosas características da programación orientada a obxectos é probablemente a reutilización do código. Idealmente, unha vez que programamos unha clase e comprobamos o seu correcto funcionamento, dispoñemos dunha unidade de código útil cunha funcionalidade definida.

O modo máis simple de reutilización do código é simplemente o uso da clase en varias partes da aplicación ou en distintas aplicacións, pero tamén podemos crear obxectos desa clase dentro dunha clase nova como un atributo:

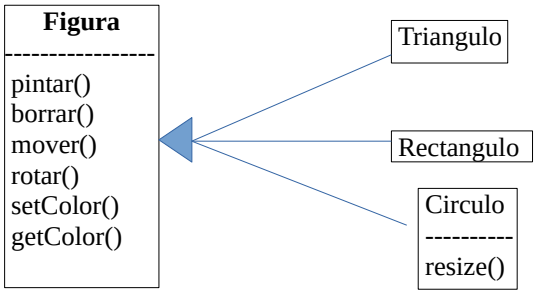

Deste xeito podemos dicir cun Vehículo **ten un** Motor. Este tipo de relación entre obxectos se coñece como ***asociación***, ***composición*** ou ***agregación***. A diferencia entre estas tres é sutil:

- ***Asociación***: Simplemente un obxecto conten outro pero non a través dunha relación forte, realmente son obxectos independentes. Por exemplo, un **Profesor** da clases a **Estudantes**
- ***Agregación***: Os obxectos teñen existencia independente. Que un dos obxectos deixe de existir non implica que o outro teña que deixar de existir. Un obxecto **Motor** pode ter existencia de xeito independente a que exista un **Vehículo**.
- ***Composición***: Os obxectos teñen unha relación dependente. Si o obxecto principal deixa de existir, tamén deixa de existir o obxecto relacionado. Por exemplo unha **Liña** dunha factura non ten sentido si a **Factura** non existe.

A relación de herdanza é máis íntima que as anteriores: A partir dunha clase é posible derivar novas clases “fillas” (*clase derivada* ou *subclase*) que herdan as características (atributos) e funcionalidade (métodos) da clase pai (*superclase*).

Calquera cambio que se faga na clase pai (superclase) afectará a todas as clases derivadas (subclases). Este tipo de relación pode describirse como **“é un”** en contraposición ao **“ten un”** utilizado na composición e agregación, deste xeito podemos dicir que un **Oso** é un **Mamífero**.

Con esta simple afirmación estamos establecendo as características básicas que teñen os *Osos*, que son comúns a todos os *Mamíferos*. Na clase *Mamífero* se describirían as características e métodos comúns a todos os *Mamíferos*, que serían automaticamente herdadas pola clase *Oso*. Na clase *Oso* definiríamos as características e métodos que distinguen aos *Osos* dos demais *Mamíferos*, e variaríamos si fora necesario o comportamento xeral dos mamíferos para adaptalo ás particularidades dos osos. Vexamos outro exemplo:




Tanto os Triángulos como os Círculos e os Rectángulos son Figuras. Polo tanto herdan todas as capacidades definidas na Figura. Os Círculos ademais poden cambiar de o seu tamaño.

# Sobrecarga, Sobreposición e Polimorfismo

Dentro dunha mesma clase poden definirse varios xeitos de realizar a mesma acción. Outro modo de definilo é dicir que dentro dunha mesma clase unha mesma acción pode "significar" cousas distintas. Esta característica se coñece como ***sobrecarga***.

A sobrecarga consiste en definir varios métodos co mesmo identificador, pero con argumentos distintos en número ou en tipo.

**Exemplo:**

```
class DivideEnDos {
        public int divide(int num) {
                return num/2;
        }

        public String[] divide(String txt) {
                String[] mitades=new String[2];  // Defino un array de 2 String
                int sz=txt.length();

                mitades[0]=txt.substring(0,sz/2);
                mitades[1]=txt.substring(sz/2);
                return mitades;
        }
}

public class ExemploSobrecarga {
        public static void main(String[] args) {
                DivideEnDos d=new DivideEnDos();
                String texto="Texto para dividir en dos";
                String[] dt;

                System.out.println("18 dividido en 2 é "+d.divide(18));
                System.out.println(""+texto+"' dividido en 2 é ");
                dt=d.divide(texto);
                System.out.println(dt[0]);
                System.out.println(dt[1]);
        }
}
```

A ***sobreposición*** consiste en variar o comportamento dun método definido na superclase.

O ***polimorfismo*** garante que se chame ao método da clase apropiada en todo momento.

**Exemplo:**

```
class Animal {
   public void talk() {
            System.out.println("Os animais non falan");
   }
}

class Can extends Animal {
   public void talk() {
            System.out.println("Guau! Guau!");
   }
}

class Gato extends Animal {
   public void talk() {
            System.out.println("Miau! Miau!");
   }
}

public class ExemploSobreposicionEPolimorfismo {
   // En 'a' se pode almacenar calqueira Animal. E os Can e Gato, SON Animal (extends...)
   public static void testPolimorfismo(Animal a) {
            // O importante e que aínda que a é de tipo Animal, o comportamento do método talk e distinto unhas veces de outras...
            a.talk(); // O polimorfismo permite tratar o Can, Gatos e Animal apropiadamente determiñando que método  chamar
   }

   public static void main(String[] args) {
            Animal a=new Animal();
            Gato g=new Gato();
            Can c=new Can();

            a.talk();  // Visualiza "Os animais non falan"
            c.talk();  // Visualiza "Guau! Guau!", xa que variei o comportamento do método talk de Animal (Sobreposicion)
            g.talk();  // Visualiza "Miau! Miau!", xa que variei o comportamento do método talk de Animal (Sobreposicion)
            testPolimorfismo(a);          // Visualiza "Os animais non falan"
            testPolimorfismo(c);          // Visualiza "Guau! Guau!"
            testPolimorfismo(g);          // Visualiza "Miau! Miau!"
   }
}
```

# Tratamento de Erros: Excepcións

O modo tradicional de tratar os erros nas aplicacións é o retorno por parte das funcións de valores especiais indicando o erro producido. Esta práctica ten varios inconvenientes:

- O valor de retorno habitualmente se utiliza para outra misión e pode ser que non dispoñamos de valores non válidos que podan indicar un erro.
- O verificar o valor devolto para comprobar si se trata de un erro é opcional, incómodo e a miúdo se deixa sen facer.
- Este modo de traballo modifica demasiado o algoritmo e a secuencia de traballo impoñendo numerosas sentencias condicionais (if) que fan o código longo e farragoso.

As linguaxes modernas proporcionan outra aproximación ao tratamento de erros: As ***excepcións***.

Unha excepción é unha notificación dun erro. Cando un programa lanza unha excepción interrompe a execución do método actual pasando o control ao código do método chamador deseñado para tratar o erro. Si ese código non existe, se lanzará a excepción ao seu método chamador anterior... e así sucesivamente ata chegar ao propio sistema. Nese caso o programa finaliza notificando o erro. Si en algún método tratamos o erro, a execución do programa continúa normalmente a partir dese punto.


Para deseñar os código de tratamento de erros, se utiliza o bloque **try {} catch(argumento) {} finally {}** ou similares (o sistema pode variar lixeiramente segundo a linguaxe de programación).

No bloque **try {}** pechamos un bloque de instrucións do que sabemos que é posible que "lance" algún erro (excepción) e no bloque **catch(argumento) {}** pechamos o código de tratamento do erro concreto que capturamos no argumento. O bloque **finally {}** permite especificar un conxunto de instrucións que se execute ao finalizar a función, se produza o erro ou non. Cando no bloque **try {}** se produza unha excepción, a execución saltará ao bloque **catch()** que corresponde ao erro producido, continuando a execución a partir de ahí e unha vez rematada a función levando a cabo o indicado en **finally {}** si se especifica. Un bloque try{} ten que levar asociado obligatoriamente como mínimo un bloque catch() **ou** un bloque finally{}

**Exemplo en Python:**

```
#/usr/bin/python
# -*- coding: utf-8 -*-

def squareRoot(x):
    if (x<0):
      raise Exception("Numero Negativo")
    error=0.000001
    b=x
    a=x/b
    while(b > a+error):
        b=(a+b)/2
        a=x/b
    return b

def ec1g(b,c):
    if (b==0):
        if (c==0):
            error="Infinitas Soluciones"
        else:
            error="Sin Soluciones"
        raise Exception(error)
    return -c/b

def ec2g(a,b,c):
    if (a==0):
        return [ec1g(b,c)]
    radicando=b*b-4*a*c
    if (radicando<0):
        raise Exception("Sin Soluciones Reais")
    sqr=squareRoot(radicando);
    sol1=(-b+sqr)/2*a
    sol2=(-b-sqr)/2*a
    return [sol1,sol2]
```

```
#--------------------   Comezo do Programa

tx2=float(input("Termo de x^2: "))
tx=float(input("Termo de x: "))
i=float(input("Termo independente: "));
try:
        print "As solucions son x=: "+str(ec2g(tx2,tx,i))
except Exception as err:
        print err.args[0]
```

## 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....");
   }
}
```