Unitat Formativa 4. Programació Orientada a Objectes. Fonaments

1. Introducció a la POO. Característiques dels llenguatges orientats a objectes

La programació orientada a objectes (POO) és un paradigma de programació molt conegut a la comunitat de desenvolupadors. La peça fonamental d'aquesta metodologia és la **classe**, que podem entendre com un tipus de dada complexe, que pot incloure internament tant diferents tipus de dades com les funcions que permeten treballar amb aquests valors interns.

Característiques principals de la POO

- **Encapsulació:** una classe pot contenir dades, de valor variable o constant, que reben el nom de **atributs**/propietats i funcions que reben el nom de **mètodes**/procediments, que permetran accedir i manegar als atributs.

- **Ocultació:** es poden definir tant atributs com mètodes com privats i protegits (inaccessibles des de les altres classes) o públics (accessibles per la resta de classes).

- **Herència:** una classe pot heretar els components (atributs i mètodes) d'una altre classe per aprofitar codi que ja està depurat i estalviar en desenvolupament i proves. Té altres avantatges que veurem més endavant, però aquesta es prou important.

- **Polimorfisme:** un mètode es pot implementar de diferent manera en funció del tipus i numero de paràmetres rebuts. També quan una classe hereta d'una altre es pot redefinir algun mètode, però aquest cas s'estudiarà específicament amb detall.

La POO potencia la reutilització del codi (augmentant el rendiment del desenvolupament i la fiabilitat dels programes) i la protecció de les dades (augmenta la seguretat al seu accès).

1.1. Concepte de classe. Sintaxi, estructura i components

Una classe és la plantilla a partir de la que es crearan els objectes, que compartiran les mateixes propietats i comportaments de la classe que instancien. Per dissenyar una classe es segueixen unes convencions:

- La primera lletra del nom de la classe en majúscula i la resta en minúscula.
- Els atributs o propietats són dades que podran ser de valor constant o variable.
- Els procediments o mètodes són operacions que ofereix la classe per manipular els atributs. Poden rebre paràmetres o no.
- Una classe pot tenir atributs que siguin objectes d'altres classes.

Per exemple, podem crear un tipus de dada que representi les característiques d'un usuari del nostre domini:

```
declare(strict_types=1);        // HO ESTABLIREM SEMPRE, NO HO INDICARÉ MÉS ALS EXEMPLES

class Guest {               //definim una classe per representar a un usuari convidat
        //components protegits, accessibles per a classes derivades
   protected string $email;
   protected string $alias;

// constructor, cridat per defecte al crear una nova instància u objecte, no pot retornar cap valor
//posar el valor per defecte dels paràmetres a null permet que es pugui crear un objecte "buit"
   public function __construct (string $email, string $alias) {
      $this->email = $email;
      $this->alias = $alias;
   }

        //components públics
   public function email(): string {
      return $this->email;
   }

   public function setEmail (string $email) {
      $this->email = $email;
   }

   public function alias(): string {
      return $this->alias;
   }

   public function setAlias(string $alias) {
      $this->alias = $alias;
   }
}
```

1.2. Instanciació d'objectes, ús del constructor

Un **objecte** és una instancia (variable) d'una determinada classe. Al crear un objecte basat en una classe, aquest rep una copia de la estructura de dades de la classe. La única diferencia entre 2 objectes de una mateixa classe és el valor dels seus atributs.

Al instanciar un objecte es reserva l'espai necessari en memòria, i la instància es realitzar la instrucció **new**. Al instanciar un objecte s'executa automàticament el procediment o mètode constructor (el mètode de nom __construct) de la classe afectada, que no podrà retornar cap valor. Per accedir als seus components farem servir l'operador '**->**'

Exemples:

```
$g = new Guest("jose@gmail.com", "Jose");
echo "L'usuari " . $g->alias() . " te el seguent correu electrònic: " . $g->email();

$g->setAlias("Joan");
$g->setEmail("joan@gmail.com");

echo "L'usuari " . $g->alias() . " te el seguent correu electrònic: " . $g->email();
```

Operador de referència this

Aquest operador és una referència al propi objecte, així en cas de coincidència entre el nom dels atributs locals de la classe i els paràmetres rebuts per un mètode, sempre es pot diferenciar entre ells fent servir l´operador this. Podem inclús retornar aquesta referència al propi objecte, si fos necessari, amb un **return $this;**

Paràmetres a la crida a mètodes

- Els tipus primitius com **int, double, char, boolean, float, short** i **byte** només es passen per valor (una còpia, cap mètode pot modificar el valor original). La classe **string** que s'utilitza per treballar amb cadenes de caràcters també funciona, excepcionalment, d'aquesta forma.

- Els objectes, i els vectors o arrays de tipus primitius o objectes es passen per referència, per tant els mètodes que els reben sí podran canviar el seu valor.

1.3. Mecanismes per controlar la visibilitat dels components de les classes

Els components d'una classe (mètodes i propietats) es defineixen amb uns **especificadors d'accés** per **encapsular** (protegir) les dades. També ens referim a aquesta encapsulació com **visibilitat** o la capacitat per poder accedir als membres d'una classe.

El procediment més comú per implementar la encapsulació és que els atributs siguin privats o protegits, per fer impossible accedir als mateixos sense fer servir els mètodes o procediments, que hauran de ser per força públics si es vol que altres objectes els puguin cridar.

Operadors per establir l'accés als components de la classe:

- **private**: els components privats (atributs i mètodes) només són accessibles per als components (mètodes) de la pròpia classe.
- **protected**: els components protegits són accessibles per la pròpia classe i les seves derivades.
- **public**: els components son accessibles per als components de la resta de les classes.
- **(friendly)**: modificador per defecte, no existeix explícitament, on components públics i protegits serien accessibles per la resta de les classes ubicades al mateix paquet (**package**).

	Components (atributs i mètodes)			
Accessibles per	**prívate**	**protected**	**public**	**(friendly)**
La pròpia classe	**Si**	**Si**	**Si**	**Si**
Altre classe (no subclasse), dins del package		**Si**	**Si**	**Si**
Altre classe (no subclasse), a un altre package			**Si**	
Subclasse, dins del package		**Si**	**Si**	**Si**
Subclasse, a un altre package		**Si**	**Si**	

Entenem com Package la carpeta on agrupem un conjunt de classes que amb relació entre elles.

1.4. Herència i Jerarquia de classes: superclasse i subclasse

El concepte de herència es molt important dins de la filosofia de la POO, és el mecanisme que permetrà la creació de nous tipus de classes basades en altres existents, aprofitant tot el que aquestes ja tenen i poden fer. És una forma de reutilitzar codi, que permet als programadors no haver de repetir funcionalitats desenvolupades.

Fem servir la paraula clau **extends** per implementa la herència entre classes. Per exemple:

```
public class Member extends Guest {  //Member és una nova classe que hereta
                                      // els components protegits i públics de la classe Persona
   protected int $id;
   protected string $name;
   protected string $paymentCode;

// constructor, cridat per defecte al crear una nova instància u objecte,
// cridant al constructor de la seva clase base a través de parent i l'operador d'ambit ::
   public function __construct ( string $email, string $alias, int $id, string $name,
                                 string $paymentCode) {
      parent::__construct($email, $alias);     //crida explícita al mètode constructor de Guest
      $this->id = $id;
      $this->name = $name;
      $this->paymentCode = $paymentCode;
   }
    //a partir de aquí caldria definir la llista de mètodes get–set per a cada nou atribut i resta
    //de mètodes necessaris per gestionar el funcionament dels objectes d'aquesta classe
}
```

A Guest se li considera una **superclasse** (perque la que fem servir com a model per crear un nou tipus de classe) i Member una **subclasse** (perque rep tots els components no privats de la superclasse).

Les subclasses poden fer servir els mètodes heretats de la superclasse o bé redefinir-los per adaptar-los a una nova funcionalitat (***sobrescriptura de métodes***), a més de poder crear els seus propis atributs i métodes. D'aquesta manera construïm **Jerarquies de Classes,** on Guest seria la classe arrel i de la que podrien heretar un conjunt de classes que a la vegada podrien servir com a superclasses d'altres classes amb un major grau d'especificació.

Constructors en l'herència. Operadors d'ambit *parent* i *::*

Així com l'operador **this** fa referència a la pròpia classe, tenim un operador **parent** que fa referència a la superclasse de la qual s'han heretat propietats. Sempre que sigui necessari cridar a un mètode de la superclasse de la que heretem (que a la pràctica equival a anul·lar-lo), ho farem explícitament posant-li al davant el prefix **parent::** amb ***l'operador d'àmbit* ::** (com queda reflectit a l'exemple del constructor, on enviem al constructor de la classe base els paràmetres necesaris).

1.5. Modificador *abstract*

Les **classes abstractes** representen un model tan genèric que no té sentit declarar cap objecte d'aquesta classe. És habitual que la classe arrel d'una estructura jeràrquica sigui tan genèrica que no serveixi instanciar cap objecte d'aquesta classe per obtenir un tipus de dada amb prou significació.

Les classes abstractes acostumen a tenir pocs atributs i els seus procediment solen restar buits (**mètodes abstractes**), que només es declaren i als que definim també com abstractes, obligant a que siguin redefinits per les subclasses. Però no és obligatori que tots els mètodes d'una classe abstracta siguin abstractes. A través de l'herència podrem definir classes derivades amb un grau de especificació suficient per tenir objectes amb significat complert.

Amb el modificador **abstract** declarem una classe com abstracta i impedim que es pugui crear un objecte d'aquesta classe.

Exemples:

```
abstract class Member extends Guest {

        public abstract function benefits():string;
        // mètode abstracte definit així per obligar a redefinir-lo a la classe que la hereti.
        // És un mètode sense cos acabat en ;
}
```

Fixeu-vos que ara ja no podem crear objectes de la classe Member. Però com és tan genèrica tampoc tindria sentit, en canvi les classes que hereten i son més específiques sí podran tenir objectes instanciats:

```
class SilverSubscriber extends Member {
   protected string $phone;
   protected string $facebook;
   protected string $instagram;

        // IMPORTANT: quedem obligats a definir els mètodes abstractes heretats de Member
   public function benefits(): string {
      return "RRSS:$this->facebook;$this->instagram";
   }
        //A continuació vindrien el constructor i els getters/setters adients
}
```

Per crear un objecte haurem de pasar als constructor tots els atributs propis i heretats de SilverSubscriber:

```
$silver = new SilverSubscriber("jorge@gmail.com", "dragonX", 101, "jorge",
                                "11223344556677889900",  "666555444", "faceID", "instaID");
```

1.6. Modificador *final*

Un altre cas el trobem quan tenim una classe tan específica que ja no té sentit que altra classe hereti d'aquesta, per lo que es pot considerar una classe final. Amb el modificador **final** establim que cap classe podrà heretar d'aquesta.

Aquest modificador s'aplica també als mètodes on establirà que ja no podran ser redefinits per les possibles subclasses i als atributs perquè no puguin ser modificats una vegada han rebut un valor (és com declarar-los atributs **amb valor constant**).

Exemples:

```
class GoldSubscriber extends SilverSubscriber {
        protected string $twitter;
        protected string $linkedin;

//final al mètode implica que ja no el podrà reescriure una classe que hereti de GoldSubscriber
        public final function forumActivity():array  {
                ...       //to do...
         }
}


// final a la classe implica que ja no es podrà heretar de DiamondSubscriber
final class DiamondSubscriber extends GoldSubscriber {
        protected array $professionalSites;

        //to do...
}
```

1.7. Incorporació i ús de conjunts i llibreries de classes.

Paquets (Packages) Les carpetes que guarden al seu interior un conjunt de classes es consideren contenidors de codi, tècnicament els denominem paquets (**packages**).

Quan una classe ubicada a un paquet necessita incorporar una classe definida a un altre paquet, amb la comanda **include** especificarem la ruta de l'arbre de directoris fins arribar al paquet on es troba la classe amb la que volem treballar. Dins d'un paquet es poden crear subpaquets per organitzar millor les classes.

```
include 'subscribers/Guest.php';          // incorporem la classe Guest
```

NOTA: Totes les classes ubicades a un mateix package ja són accesibles directament entre elles, no cal utilitzar **include** o de la seva família com **include_once, require, require_once.**

1.8. Sobreescriptura de mètodes

Per defecte s'hereten tots els components ***protected*** i ***public*** de la superclasse, però pot ser necessari redefinir algun mètode heretat mantenint la signatura original. A aquest mecanisme se l'anomena **sobreescriptura de mètodes**, i permet **redefinir** els mètodes heretats de la superclasse.

NOTA: Hem de tenir clara la diferència entre redefinir un mètode i **sobrecarregar-lo.** Sobrecarregar-lo aprofita el nom del mètode però fa servir diferent nombre i/o tipus d'arguments, i a llenguatges com PHP no està suportat, a d'altres com Java sí..

Exemple:

```
/*La classe GoldSubscriber que hereta de SilverSubscriber necessita redefinir el mètode benefits
per actualitzar els avantatges a nivel de autodifusió d'aquesta subscripció */
class GoldSubscriber extends SilverSubscriber {
        protected string $twitter;
        protected string $linkedin;

        //Sobreescrivim benfits per a una nova funcionalitat, però aprofitan la anterior
        public function benefits(): string {
                //cridem al mètode de la superclasse i afegim més xarxes
                return parent::benefits() . ";$this->twitter; $this->linkedin";
        }
}
```

1.9. Crides a mètodes estàtics

Trobem una excepció a la regla de la programació orientada a objectes que diu que per utilitzar un mètode o atribut es necessari crear un objecte de la classe a la que pertanyen. Els components **estàtics** es defineixen amb la paraula clau **static**, per poder-los utilitzar sense instanciar objectes.

Exemple:

```
class Operations {
        public static function factorial(int $n): float {
           $fact = 1;
           for (; $n > 0; $n--) {
              $fact = $fact * $n;
           }
           return $fact;
        }
}
print "<br><br>RESULT: " . Operations::factorial(6);          //cridem a un mètode estàtic
```

Els mètodes estàtics també es coneixen com ***genèrics***, per que el resultat de les seves operacions no depèn de les propietats de l'objecte instanciat, donarien el mateix resultat a qualsevol objecte. Una característica d'aquests mètodes és que només consumeixen la memòria necessària per definir-los.

Si definim com a estàtics els atributs d'una classe establim una funcionalitat diferent. Un atribut estàtic vol dir que és únic i comú per a tots els objectes de la classe (es com una variable global però limitada a l'àmbit dels objectes d'una mateixa classe). Només s'inicialitza amb el primer objecte definit de la classe a la que pertany, i tots els objectes podran accedir a aquest atribut i modificar-lo.

Un exemple ràpid d'entendre amb atributs estàtics seria el de implementar un comptador d'objectes d'una classe. Imaginem que el volem per a la classe exemple Persona:

```
abstract class Member extends Guest        {
        protected static int $members = 0;  //atribut  estàtic, només el posa a 0 el primer objecte

        public function __construct(string $email, string $alias, int $id, string $name, string
        $paymentCode) {
                parent::__construct($email, $alias);

                $this->id = $id;
                $this->name = $name;
                $this->paymentCode = $paymentCode;
                self::$members++;            // Adaptem el constructor afegint la següent línia
        }
```

```
        //per una gestió coherent hauré de fer alguna cossa quan l'objecte es destrueixi
        public function __destruct() {
                self::$members--;
        }

        // i finalment, proporcionarem un mètode per conèixer el numero de membres creatsi
        public function getMembers(): int {
                return self::$members;
        }
}
```

Podem provar la funcionalitat de forma simple de la seguent forma, on observarem com l'atribut s'actualitza a mida que es van creant i destruint membres:

```
$silver = new SilverSubscriber ("jorge@gmail.com", "dragonX", 101, "jorge", "11223344556677889900");

$gold = new GoldSubscriber("joan@gmail.com", "TigerZ", 102, "joan", "11223344556677889988");
echo "<br><br>- Num de subscribers online: " . $silver->getMembers();

$diamond = new DiamondSubscriber("josep@gmail.com", "BlackPanther", 103, "josep", "11223344556677889999");

echo "<br><br>- Num de subscribers online: " . $gold->getMembers();

$gold = null;
echo "<br><br>- Num de subscribers online: " . $diamond->getMembers();
```

1.10. Sobrecàrrega de mètodes: Polimorfisme

El polimorfisme és un mecanisme de la POO que permet sobrecarregar els mètodes de manera que amb un mateix nom podem implementar diferents variants del mateix, on l'únic que canviarà és el tipus i numero de paràmetres d'entrada utilitzats.

NOTA: alguns llenguatges com PHP no soporten el polimorfisme, encara així es mostra la seva implementació encara que aquest llenguatge no li dona suport.

Exemple amb un mètode anomenat **suma**. Al cridar-lo, si PHP suportés el polimosfisme, en temps real s'executará la instància adequada del mètode en funció dels paràmetres subministrats.

```
class Operations {

        public static function suma(float $x, float $y): float {
           return $x + $y;
        }

      //Aquest mètode i el següent donen error perquè a PHP no es pot repetir el nom
      //no suporta la sobrecàrrega de mètodes

        public static function suma(int $x, int $y): int {
           return $x + $y;
        }


        public static function suma(array $list): float {
           $result = 0;
           foreach ($list as $r) {
              $result += $r;
           }
           return $result;
        }
}
```

On volguem fer servir el mètode estàtic podem escriure aquesta línia i comprovar resultats:

```
print "<br><br>RESULT: " . Operations::suma(13.25, 7.50);
```

1.11. Herència múltiple, problemes derivats

L'Herència múltiple permet que una classe pugui derivar o heretar de més d'una superclasse, ampliant la varietat de atributs, mètodes i constructors de la nova subclasse. L'avantatge és un major aprofitament del codi, pero els problemes associats moltes vegades no compensen els efectes relacionats:

- Ens podríem trobar que les superclasses tenen components (atributs o mètodes) amb el mateix nom, provocant un conflicte o obligant a redefinir components (no es molt lògic).

- El concepte de "híbrid" que es produeix quan heretes de 2 o més classes de vegades resulta molt forçat i no s'aprofiten totes les característiques de totes les superclasses.

C++ és dels pocs llenguatges orientats a objectes que suporta aquest tipus d'herència, la majoria dels que es fan servir usualment no permeten més que l'herència simple de forma directa.

1.12. Definició i implementació d'interfícies

Una interfície ens permet definir una sèrie de comportaments o serveis, sense especificar com han de ser implementats. Les classes que hagin d'implementar una interfície hauran d'especificar internament com duran a terme els comportament o serveis definits. Per això es pot considerar que una interfície és ***com un contracte de serveis***.

Les classes que implementin una interfície podrien ser de diferents famílies o jerarquies, però totes resten compromeses a implementar els serveis de la interfície.

Exemples de definicions de interfícies:

Les interfícies es crearan igual que les classes, dins d'un determinat paquet, com un arxiu més amb extensió *.java* i el seu equivalent en bytecode *.class*. Per nombrar les interfícies, s'aconsella seguir el mateix criteri que amb les classes: la primera lletra en majúscula.

Si definim algun atribut a la interfície ha de ser final o **constant** i els **mètodes públics són per defecte abstractes** (no es poden implementar a dins) sense tenir que indicar-ho. Només es definieix el prototipus o signatura de la crida al servei que inclou la interfície.

```
interface Billable {                    //interfície per a qualsevol producte o servei facturable
        public function getCode ():string;
        public function getPrice ():float;
        public function getData ():string;
        public function getDetails ():string;
}
```

```
interface Transportable {              //interfície per a qualsevol producte transportable
        public function getCode ():string;
        public function getDimensions ():string;
        public function getVolum ():float;
        public function getWeight ():float;
        public function isFragile ():bool;
}
```

Exemple d'una classe que implementa una interfície:

Només cal afegir a la definició de la classe la paraula **implements** i el nom de la interfície (o interfícies separant-les per comes) a implementar.

```
class Rent implements Billable {

        //definició dels atributs, constructors i d'altres mètodes que siguin necessaris
        public function getCode ():string {
                // definició específica del mètode
        }

        public function getPrice ():float  {
                // definició específica del mètode
        }

        public function getData ():string {
                // definició específica del mètode
        }

        public function getDetails ():string {
                // definició específica del mètode
        }
}
```

Els comportaments específics es definiran a cada classe que implementi la interfície.

Una classe pot implementar múltiples interfícies, simplement haurà d'implementar els respectius mètodes de cadascuna de les interfícies implementades.

```
class Order implements Billable, Transportable {

        // Implementació específica dels mètodes d'ambdues interfícies

}
```

Una interfície també pot heretar d'altres interfícies, però amb la particularitat de que **les interfícies sí admeten la *herència múltiple*.** Les interfícies permeten "amagar" la limitació de no implementar la herència múltiple, encara que no és la seva finalitat.

Exemples de herència simple i múltiple entre interfícies:

```
interface ExtendedBillable extends Billable {
        public function getSurcharges ():float;
        public function getTaxes ():float;
}

interface TransportBillable extends ExtendedBillable, Transportable {
        //si es necessari afegiria els seus propis mètodes o atributs constants
}
```

La interfície ***TransportBillable*** hereta tots els components de les altres dues, i qulasevol classe que la implementi haurà de redefinir tots els mètodes heretats, i els propis si hi hagués.

Amb els exemples anteriors es mostra com podem aconseguir una herència múltiple orientada a serveis (i atributs constants). Podem considerar una interfície com un tipus de façana que defineix uns serveis o comportaments, encapsulant l'objecte de la classe que ofererirà realment els serveis.

No podrem crear instancies d'objectes a partir d'una interfície com fem amb les classes a través de **new**, però qualsevol objecte de qualsevol classe que implementa una interfície és compatible amb el tipus definit per la interfície.

L'avantatge de poder fer servir una interfície com un tipus de dada, és que podrem utilitzar objectes de diferents classes (que implementin la interfície) com a paràmetre de funcions que ofereixen un servei, obtenint un codi més genèric i amb moltes més possibilitats de reutilització i adaptació, perquè obtenim un codi molt menys acoblat.

Exemple amb una interfície per als Patrocinadors (empreses o usuaris del nostre negoci):

```
interface Partner {                    //interfície per a qualsevol patrocinador, empresa o particular
   public function name():string;
   public function contactData():string;
}
```

Les classes **GoldSubscriber** i una nova classe **Company,** que representa a una empresa que ens patrocinarà, implementaran els serveis d'aquesta interfície. La classe **DiamondSubscriber**, al heretar de GoldSubscriber, ja implementa per defecte la interfície Partner sense que ho tinguem que especificar, només haurà de redefinir, si fos necessari, la forma d'oferir els serveis.

```
class GoldSubscriber extends SilverSubscriber implements Partner {
        // Implementem el mètode que restava ja que name() ja estava definit
         public function contactData(): string {
              return "Phone:$this->phone;Email:$this->email;LinkedIn:$this->linkedin";
         }
}

final class DiamondSubscriber extends GoldSubscriber {

        public function contactData(): string {
                // Es sobreescriurà el per adaptar-lo a aquesta classe
        }
}

class Company implements Partner {
           protected int $id;
           protected string $name;
           protected string $address;
           protected string $email;
           protected string $phone;
           protected string $representative;
           protected string $sector;

           public function contactData(): string {
              return "Phone:$this->phone;Email:$this->email;Address:$this->address";
           }
        // A continuació caldrà definir el constructor i els getters/setters respectius
}
```

Per comprovar el funcionament crearem un petit programa:

```
$comp = new Company (301, "Bill SA", "Carrer 7", "bill@gmailcom", "666555444", "Victor", "Transport");

$diamond = new DiamondSubscriber ("josep@gmail.com", "BlackPanther", 103, "josep", "11223344556677889999", "666555222", "faceID", "instaID", "linkID", "tweetID", "myProfSite");


// declarem una funció de proves que rebi un objecte que implementi la interfície Partner
function printContactData (Partner $partner ):void {
   echo "<br><br>- Patrocinador: " . $partner->contactData();
}

// Finalment, des del mateix programa, cridem a la funció enviant qualsevol dels objectes que implementan la interfície. Encara que els objectes son de classes diferents, tenen un comportament comú establert per la interfície
printContactData($diamond);
printContactData($comp);
```

1.13. Aspectes relacionats amb el disseny de jerarquies de classes

Per aprofundir en les possibilitats de llenguatge veurem alguns aspectes:

- Ús de **instanceof** per comprovar a quina classe pertany un objecte.

  Tornant als exemples amb els usuaris que poden ser patrocinadors (Partners) de la nostra activitat, podríem tenir la necessitat de conèixer de quin tipus concret d'objecte es tracta, ja que rebem un objete que implementa Partner però això no ens indica de quin tipus d'objecte es tracta.

  Per exemple, una funció semblant a la de l'apartat anterior:

```
function printObjectType(Partner $partner): void {

   if ($partner instanceof DiamondSubscriber) {
      echo "<br><br>Es tracta d'un DiamondSubscriber. Aqui el seu website: ";
      echo implode($partner->getProfessionalSites());
   }

   if ($partner instanceof Company) {
      echo "<br><br>Es tracta d'una Company. Aqui el seu sector: ";
      echo $partner->getSector();
   }
}
```

  Observem que fem la pregunta a la variable $***partner*** per saber si es una instancia de ***DiamondSubscriber*** o de ***Company***, perquè volem accedir a un mètode exclusiu de cadascuna d'aquestes classes. Aquests mètodes no hi son a la interfície ***Partner*** i no els podem cridar sense estar-ne segurs de que es tracta de la classe adient perquè provocaríem un error.

  L'operador **instanceof** ens ajuda a resoldre aquest problema. Però com es pot comprovar genera un codi complexe i de difícil manteniment quan la varietat d'objectes rebuts que implementin *Partner* comenci a créixer, per tant cal utilitzar-lo amb un criteri clar o millor, intentar no fer servir sempre que sigui possible aquest operador.

- **Comparar objectes:** els operadors **==** i **===** permeten comparar referències a objectes.

  L'operador de comparació == compara 2 objectes, indicant que son iguals si els 2 objectes son instàncies de la mateixa classe i tenen els mateixos atributs amb valors coincidents.

  Exemple:

```
$comp = new Company(301, "Bill SA", "Carrer 7", "bill@gmailcom", "666555444", "Victor",
"Transport");
$comp1 = new Company(302, "Gill SA", "Carrer 9", "gill@gmailcom", "666555333", "John",
"Economist");
$comp2 = new Company(302, "Gill SA", "Carrer 9", "gill@gmailcom", "666555333", "John",
"Economist");

if ( $comp == $comp1 ) {
    echo "Son Objectes de la mateixa clase i amb idèntic contingut";
} else{
    echo "No son Objectes de la mateixa clase o tenen diferent contingut";   // < -- -
}

echo "<br><br>";
if ( $comp1 == $comp2 ) {
    echo "Son Objectes de la mateixa clase i amb idèntic contingut";          // < -- -
} else{
    echo "No son Objectes de la mateixa clase o tenen diferent contingut";
}
```

  L'operador d'identitat === també compara 2 objectes, però en aquests cas indica que son iguals només si els 2 objectes fan referència a la mateixa instància de la mateixa classe. Vol dir que només tenim un objecte però podem tenir diferents referències per accedir a ell.

  Exemple:

```
$refComp1 = $comp1;              //definim una segona referència al mateix objecte

if ($comp1 === $comp2) {
    echo "Les dues referències apunten al mateix Objecte";
} else {
    echo "Les dues referències NO apunten al mateix Objecte";                // < -- -
}

echo "<br><br>";
if ($comp1 === $refComp1) {
    echo "Les dues referències apunten al mateix Objecte";                   // < -- -
} else {
    echo "Les dues referències NO apunten al mateix Objecte";
}
```