El llenguatge PHP (II): Paradigma orientat a objectes

Introducció als conceptes de classe, objecte i herència

La programació orientada a objectes (POO) permet escriure programes amb més facilitat sobre escenaris del món real, ja que seguint aquest paradigma, els desenvolupadors defineixen literalment coses (els "**objectes**") -a partir de plantilles (les "**classes**")- on els estableixen algunes característiques bàsiques (les "**propietats**") i els demanen que facin coses (executar "**mètodes**").

Per exemple: pensem en un objecte de tipus "gos": hi ha molts gossos al món, però només un animal que és de la *classe* "gos". Com a tal, tenim un plantilla a partir de la qual estan fets tots els gossos (per molt diferent que siguin entre ells): tots tenen quatre potes, el nas humit, una determinada raça, un determinat color de pèl, no els agraden els gats...i borden. Així doncs, a partir d'aquesta classe "gos" podríem crear (o en paraules tècniques, "instanciar") un gos real anomenat per exemple "Poppy" que tingués assignats valors concrets en les propietats definides a la classe (és a dir, ser d'una raça determinada, tenir un color de pèl determinat, etc) per tal d'adquirir així la seva pròpia "individualitat" i, com no, que pogués realitzar les tasques ("mètodes") definides també a la classe "gos".

Segurament pensareu que propietats com "nombre de potes", "color de pèl" o "raça" no estan intrínsecament relacionades amb la condició de ser un gos sinó més aviat en la condició d'"animal". Per tant, seria més ordenat definir una *classe* "animal" que contingués totes aquestes propietats més "genèriques" (i els mètodes genèrics que necessitéssim també, com ara "córrer", "menjar", "dormir", etc) i reservar només les propietats i mètodes més intrínsecament relacionats amb el fet de ser gos a la classe "gos", la qual, això sí, hauria llavors d'*heretar* de la classe "animal" totes aquelles propietats i mètodes definits en ella i que, pel fet de ser "gos", també tenen. Fent-ho així, no costaria gaire ampliar l'esquema amb una altra classe "gat" que també heretés de la classe ja feta "animal" i que només incorporés dins seu les propietats i mètodes específics del fet de ser gat. L'**herència** es refereix sovint a una subclassificació: "gos" i "gat" serien sengles subclasses d'"animal".

> NOTA: Alguns llenguatges, com el C++, us permeten heretar de més d'una classe, que es coneix com a herència múltiple. Aquesta tècnica permetria tenir, per exemple, una classe "ocell" i una classe "cavall" i llavors poder crear una classe "cavall volador" que heretés tant d'ocell com de cavall (per obtenir animals com Pegàs). PHP no permet fer-ho perquè generalment fa que els programes siguin molt confusos, i és bastant rar fins i tot en C++.

> NOTA: El que sí permet PHP és heretar heretar tantes vegades com es vulgui. Per exemple, refinant l'esquema que hem descrit als paràgrafs anteriors, la classe "gos" podria heretar en realitat de la classe "carnívor", que contindria gats, gossos, óssos, etc. Els carnívors podrien heretar de la classe "mamífer", que conté tots els mamífers, que al seu torn podrien heretar de la classe "vertebrats", mantenint tots els animals amb una columna vertebral, etc: com més amunt es puja en la jerarquia, més vagues es tornaran les classes perquè cada classe hereta les funcions i variables de la seva classe pare, a més d'afegir les seves pròpies.

Implementació en PHP

A continuació es mostra el codi necessari per definir una classe "animal" que, de moment, no té cap propietat i només consta d'un mètode (que en realitat no és res més que una funció que els objectes d'aquesta classe podran executar...ja veurem què significa la paraula *public* que hi apareix al principi de la seva definició i que encara no hem vist). Com es pot veure, una classe es defineix mitjançant la paraula clau *class*, així

```
<?php
class animal {
   public function move(float $vel=50) : void {
      print "I move at $vel speed\n";
   }
}
?>
```

> NOTA: Els noms de les classes segueixen les mateixes regles que els de les variables a excepció del símbol $ al principi

Si creéssim un objecte de classe "animal", podríem invocar al mètode *move()* per tal de veure per pantalla el missatge corresponent però ara per ara, només definint la classe no tenim res "viu", només una mera definició (tenim l'equivalent a haver definit una funció però sense haver-la cridat encara) i per tant, no veurem que passi res en executar el codi anterior. Aviat ho farem.

Abans, però, crearem una altra classe, la classe "dog", que heretarà de la classe "animal". Això vol dir que "dog" tindrà també el mètode *move()* -tot i que, com es pot veure al codi, PHP permet redifinir mètodes heretats per tal que es comportin diferent en les subclasses- a més un mètode propi, *bark()*. L'herència s'implementa mitjançant la paraula clau *extends*, així

```
<?php
class animal {
   public function move(float $vel=50) : void {
      print "I move at $vel speed\n";
   }
}
class dog extends animal {
   public function bark()  : void {
      print "Bup, bup!\n";
   }
  public function move(float $vel=50) : void {
      $vel=$vel*2;
      print "I move at $vel speed because I'm a dog!\n";
   }
}
?>
```

Ha arribat el moment de crear els nostres primers gossos "reals" (és a dir, instanciar un o més objectes concrets a partir de la classe "gos"). Això es pot fer amb la paraula clau *new*, la qual generarà, a partir de la classe indicada, un objecte que podrem manipular a partir de llavors en el nostre codi de forma similar (tot i que amb diferències importants que anirem veient) a una variable (de fet, el nom de l'objecte creat ha de començar igualment amb "**$**"). Veiem-ho en un exemple (on es pot veure que per poder finalment executar un mètode pertanyent a un determinat objecte cal utilitzar la notació "*$nomObjecte->nomMètode(...)*"

> NOTA IMPORTANT: Les definicions de les classes "animal" i "dog" indicades a l'exemple anterior s´ha suposat que s'han traslladat a un arxiu "classes.php" per tal de no fer el codi més llarg del necessari

```
<?php
require("classes.php");
$poppy = new dog;
$poppy->bark();
$poppy->move();
?>
```

En executar l'script anterior, hauria d'aparèixer per pantalla, ara sí, les frases *"Bup, bup!"* i *"I move at 100 speed because I'm a dog!"* Si comentéssim, no obstant, la (re)definició del mètode *move()* en la classe "dog" i tornéssim a executar veuríem, en canvi, que les frases serien "Bup, bup!" i "I move at 50 speed"

Veiem ara com s'implementen les **propietats** dins de les classes. N'afegirem una a la classe "animal", així:

```
class animal {
   public string $name;
   public function move(float $vel=50) : void {
      print "I move at $vel speed\n";
   }
}
```

Com es pot veure, afegir una propietat en una classe consisteix simplement en declarar-hi al seu interior una variable d'un cert tipus (i afegint-hi un modificador de visibilitat -en aquest cas, *public*- que de seguida veurem què significa). Un cop afegida, tots els objectes creats a partir de la classe "animal" o de totes les subclasses que n'heredin (com la classe "dog") tindran aquesta propietat.

> NOTA: Indicar el tipus en una propietat és opcional...en l'exemple anterior es podria haver escrit simplement *public $Name;* Però llavors estaríem perdent el control del tipus en els possibles valors que podria adquirir, cosa que no desitjarem En tot cas, els tipus més comuns són *string, int, float, bool, array* i qualsevol classe reconeguda
> NOTA: És curiós saber que en el cas de definir variables "estàndard", no és possible fer-ho indicant el seu tipus (!?)

Veiem-ho: al següent exemple assignem valors concrets de la propietat "Name" a sengles objectes concrets de tipus "dog" (observeu la notació "*$nomObjecte->nomPropietat*" , on aquí el nom de la propietat no va precedit per cap símbol "$", atenció!); com es pot veure, cada objecte té el seu conjunt de propierats propi independents de la resta d'objectes de la mateixa classe:

```
<?php
require("classes.php");
$poppy = new dog;
$penny = new dog;
$poppy->name="Poppy";
$penny->name="Penny";
echo "Els meus gossos són $poppy->name i $penny->name";
?>
```

Cal saber que dins dels mètodes implementats per un objecte es pot tenir accés a qualsevol de les seves propietats mitjançant l'expressió "*$this->nomPropietat*" La paraula "this" sempre apuntarà a l'objecte particular que en aquell moment estigui executant aquesta expressió. Per exemple, si ara fem...:

```
class animal {
   public string $name;
   public function move(float $vel=50) : void {
      print "$this->name moves at $vel speed\n";
   }
}
```

> NOTA: També es podria haver indicat *$this->name* entre claus (així, *{$this->name}*) per assegurar-se que no hi ha errors a l'hora d'interpretar tota l'expressió

...un codi com el següent mostrarà la frase "Poppy moves at 50 speed" (suposant que la classe "dog" no sobreescriu la definició del mètode *move()*):

```
<?php
require("classes.php");
$poppy = new dog;
$poppy->name="Poppy";
$poppy.move();
?>
```

Existeixen un conjunt de paraules clau (com la ja vista *public*, entre altres que ara veurem) que es poden indicar no només al principi de la declaració d'una propietat o la d'un mètode, sinò també al principi de la declaració de la classe com a tal, que modifiquen el comportament de la propietat/mètode/classe a la què afecten. Concretament, són:

* *public* : La propietat/mètode es pot usar en qualsevol lloc de l'script
* *private* : La propietat/mètode només es pot usar dins de l'objecte on pertany
* *protected* : Com *private* però podent-se usar també en objectes descendents de la classe on pertany

I, opcionalment afegint-hi als anteriors, també tenim aquestes altres:
* *final* : La propietat/mètode no pot ser sobreescrita en subclasses
* *abstract* : El mètode/classe no pot ser usat directament: cal heredar-lo abans

A continuació es veurà la seva utilitat. Per exemple, un "problema" de les propietats públiques és que poden ser manipulades (és a dir, llegides/escrites) des de qualsevol lloc de l'script, cosa que no sol ser gaire segur. Per evitar això, les propietats es poden establir de tipus "privat", amb la qual cosa només els mètodes propis de la pròpia classe podran manipular-les.

> NOTA: La mateixa lògica existeix amb els mètodes públics: poden ser cridats des de qualsevol lloc de l'script; si volem que només puguin ser cridats des d'altres mètodes que estiguin definits a la mateixa classe i prou, cal fer-los privats

Per exemple, en la següent definició la propietat *$name* no podrà ser accedida d'enlloc excepte d'algun mètode pertanyent a la propia classe "animal" (com és el cas de *setName()*, el qual sí que és públic perquè volem que es pugui cridar des de qualsevol lloc). Amb això el que estem aconseguint, en el fons, és "forçar" a accedir a *$name* de forma controlada només a través del mètode *setName()*.

```
class animal {
   private string $nom;
   // ...
  public function setName(string $newName) : void {
     $this->nom=$newName;
  }
}
```

En el codi anterior, el mètode "setName()" és el que se'n diu un "**setter**"; és a dir, un mètode públic que permet establir "des de fora" el valor a una propietat privada concreta d'un objecte (per cadascuna n'hi haurà un "setter" diferent). Igualment, existeix el concepte de "**getter**" : un mètode públic que no té paràmetres i que serveix per retornar el valor d'una propietat privada concreta (per cadascuna n'hi haurà un "getter" diferent) i, per tant, permet obtenir-los/llegir-los/mostrar-los "des de fora". A continuació mostrem una ampliació del codi anterior afegint-hi el getter que hem anomenat "getName()" i un exemple del seu ús:

```
<?php
class animal {
  private string $nom;
  public function setName(string $newName) : void {
     $this->nom=$newName;
  }
  public function getName() : string {
     return $this->nom;
  }
}
$animal1=new animal;
$animal1->setName("Pepito");
echo $animal1->getName() . "\n";
?>
```

> NOTA: Existeix una altra forma d'implementar "getters" i "setters" que és mitjançant el mètodes predefinits que l'intèrpret PHP incorpora automàticament a qualsevol classe i que són *__get($nomPropietat);* i *__set($nomPropietat,valorPropietat);* Si els implementem d'aquesta manera, l'accés, tan de lectura com de modificació, a la propietat en qüestió formalment es podrà realitzar "des de fora" igual que si fos pública. Un exemple, variació del codi anterior:

```
<?php
class animal {
  private string $nom;
  public function __set($nom, string $newName) : void {
     $this->nom=$newName;
  }
  public function __get ($nom) : string {
     return $this->nom;
  }
}
$animal1=new animal;
$animal1->nom="Pepito";
echo $animal1->nom . "\n";
?>
```

Veiem ara un exemple d'ús d'un mètode *protected* (recordem, una variable/mètode amb aquesta característica és accessible només dins de l'objecte on pertany, ja sigui perquè està definit a la classe del propi objecte o perquè ha sigut herdat d'alguna classe superior, tant és). En el codi següent, quan es crida al mètode *bark()* -que és públic, així que en principi no hi hauria d'haver cap problema-, es produeix un error. Aquest error és degut a que en el codi de *bark()* , pertanyent a la classe "dog", s'utilitza el "getter" *getName()*, que és privat de la classe "animal" (i, per tant, només el podrien utilitzar mètodes de la pròpia classe "animal" i prou).

```
<?php
   class animal {
      public string $nom;
      private function getName() : string {
         return $this->Name;
      }
   }

   class dog extends animal {
      public function bark() : void {
         print "'Woof', says {$this->getName()} \n";
      }
   }

   $poppy = new dog;
   $poppy->nom = "Poppy";
   $poppy->bark();
?>
```

La solució és passar el mètode *getName()* de *private* a *protected*. Si ho feu i proveu el codi anterior, llavors sí que funcionarà perfectament perquè el que hem aconseguit és fer accessible *getName()* als objectes de tipus "animal" però també a tots (i només!)els que n'heredin d'aquesta classe, com són els de tipus "dog".

Veiem ara un exemple d'ús d'un mètode *final* (recordem, una variable/mètode amb aquesta característica no pot ser sobreescrita per una subclasse). Això permet evitar que altres programades puguin usar el codi propi fora dels límits que hom havia inicialment pensat. Per exemple, en executar el codi següent es dispararà un error perquè s'ha volgut sobreescriure el mètode *move()* a la classe "dog" quan s'havia definit com a "final" en la classe superior "animal".

> NOTA: El missatge d'error obtingut és similar a "*Cannot override final method animal::move()* ..." . Els dobles dos punts ("::") que hi apareixen són un símbol que permeten accedir a propietats o mètodes estàtics, constants o sobreescrits d'una classe (més sobre això més endavant)
> NOTA: Si es vol ser més radical, la paraula final es pot assignar a la declaració d'una classe sencera. D'aquesta forma, aquesta no podrà ser heredable (és a dir, no es podran crear subclasses a partir d'ella)

```
<?php
class animal {
   final public function move() : void {
      print("Fiu fiu!!\n");
   }
}
class dog extends animal {
  public function move() : void {
      print("Mec mec!!\n");
   }
}
$poppy= new dog;
$poppy->move();
?>
```

Que una propietat o mètode siguin declarats en una classe com a *static* significa que podran utilitzar-se en el codi del nostre script de forma directa (és a dir, sense haver d'instanciar primer cap objecte concret per poder llegir/escriure la propietat i/o cridar el mètode en qüestió). És fàcil veure que els **mètodes estàtics** el que permeten és fer-los servir com a funcions "estàndard"; les **propietats estàtiques** el que permeten, al seu torn, és tenir només un valor assignat (el valor de la propietat associada a la classe en sí, el qual compartiran llavors tots els objectes que siguin instanciats a partir d'ella) en contra del que seria l'habitual, que seria tenir un valor diferent per cada objecte diferent.

Al codi següent veiem un exemple d'ús de propietat estàtica, on s'ha utilitzat també la paraula clau *self::*, la qual serveix per referir-se sempre a la classe de l'objecte actual (¡no pas a l'objecte en sí, que seria amb *this*!!). Concretament, el que fa aquest codi és anar actualitzant una propietat estàtica cada cop que es creïi un nou objecte a partir de la classe en qüestió, la qual funcionarà precisament com a (únic) comptador dels objectes creats; és per això, doncs, que mostra els valors 1, 2, 3, corresponents als IDs dels tres objectes "empleat" creats, i 4, corresponent al valor actual de la propietat estàtica, que representa el proper ID que s'assignarà al pròxim objecte "empleat" que es creï (gràcies al seu ús dins del contructor...en parlarem aviat).

```
<?php
   class empleat {
      static public int $NextID = 1;
      public int $ID;

      public function __construct() : void {
         $this->ID = self::$NextID;
         self::$NextID++;
      }
   }

   $bob = new empleat;
   $jan = new empleat;
   $simon = new empleat;
   print $bob->ID . "\n";
   print $jan->ID . "\n";
   print $simon->ID . "\n";
   print empleat::$NextID . "\n";
?>
```

> NOTA: Recumint: un membre estàtic (propietat o mètode) pertany a la classe, no a les instàncies (objectes); és a dir, totes les instàncies comparteixen la mateixa còpia del membre estàtic. A partir d'aquí, per accedir als membres estàtics s'utilitza l'operador "::" en lloc de l'operador "->" i cal saber que "self" es refereix a la classe actual mentre que "this" es refereix a l'objecte actual; eś a dir, en la definició de la classe caldrà indicar *$this->membre* per referir-nos a un membre no estàtic i *self::$membre* per un d'estàtic.

A més de *self::* també podem disposar de la paraula clau *parent::* , la qual ens serveix per referir-nos a la classe "mare" de la classe de l'objecte actual i poder accedir així a les propietats i mètodes estàtics que aquesta pogués tenir.

Anar heredant de classe en classe és un mecanisme molt poderós per construir funcionalitat en els nostres scripts però sovint és fàcil perdre's en la cadena d'herència i molt sovint haurem de contestar a la pregunta: ¿com es pot **saber de quina classe és un objecte determinat**? Per saber la resposta podem fer servir la sentència *instanceof*, que pot ser utilitzada com un operador: retornarà *true* si l'objecte indicat a la seva esquerra és de la classe (o d'alguna de les classes superiors heredades!) indicada a la seva dreta. Per exemple, si tenim *$poppy = new dog;* tant *if ($poppy instanceof animal) { }* com *if ($poppy instanceof dog) { }* haurien de retornar *true*.

Si només es vol saber si un objecte determinat és descendent d'una classe però sense ser d'ella, es pot usar *is_subclass_of($nomObjecte, "nomClasse");*, la qual retorna *true* si l'objecte indicat al primer paràmetre (el qual també podria ser el nom d'una subclasse) és descendent de la classe indicada al segon paràmetre, i *false* si no (que inclou el cas en què l'objecte és de la classe indicada)

Existeixen unes quantes funcions relacionades més que són interessants: *class_exists("nomClasse");* retorna *true* si la classe indicada ha sigut declarada, *get_class($nomObjecte);* retorna el nom de la classe de l'objecte indicat, i *get_declared_classes();* retorna un array de totes les classes de les quals podríem crear-ne objectes.

Les propietats d'un objecte poden ser recorregudes igual que els elements d'un array, amb el bucle *foreach* (sempre que siguin accessibles). Veiem-ho en un exemple (on es pot comprovar com la propietat *$Password* és ignorada en el recorregut del bucle ja que és privada de la classe; en aquest sentit, fixeu-vos, en canvi, que la propietat *$FavouriteColour*, tot i no tenir assignat cap valor en l'objecte "person1" com li passa a *$Password*, sí que apareix al **llistat de propietats recorregudes** -amb valor *null*-).

```
<?php
class person {
  public string $FirstName;
  public string $LastName;
  private string $Password;
  public int $Age;
  public string $FavouriteColour;
}
$person1=new person;
$person1->FirstName = "Bill";
$person1->LastName = "Murphy";
$person1->Age = 29;
foreach($person1 as $var => $value) {
   echo "$var is $value\n";
}
?>
```

Si canviem, però, el codi anterior per a què el bucle estigui definit dins de la propia classe, llavors *$Password* sí que apareixerà al llistat de propiertats recorregudes (també hem aprofitat, com es pot veure, per declarar diverses propietats del mateix tipus en una sola línia per mostrar que això és possible fer-ho també):

```
<?php
class person {
  public string $FirstName, $LastName, $Password, $FavouriteColour;
  public int $Age;

  public function outputVars() : void {
     foreach($this as $var => $value) {
       echo "$var is $value\n";
     }
  }
}
$person1=new person;
$person1->FirstName = "Bill";
$person1->LastName = "Murphy";
$person1->Age = 29;
$person1->outputVars();
?>
```

Als exemples anteriors, en crear un objecte a partir de la classe "person" hem hagut tot seguit d'indicar una per una les seves propietats. Afortunadament, existeix una manera més ràpida i còmoda de, entre altres tasques, inicialitzar les propietats d'un objecte recentment creat: mitjançant l'ús d'un **constructor**. Un constructor és un mètode d'una classe que té la particularitat de cridar-se automàticament en el moment de crear un objecte d'aquesta classe. Per tant, el podem aprofitar, com diem, per "omplir" les propietats d'aquest nou objecte, entre altres coses que volguem. Per fer que un mètode d'una classe sigui el seu constructor (només en pot haver un per classe...si n'hi hagués més només es faria servir el darrer escrit en el codi) només cal que el seu nom sigui específicament aquest: *__construct()*. Veiem-ne un exemple:

```php
<?php
class person {
  private string $FirstName;
  private string $LastName;
  private string $Password;
  private int $Age;
  private string $FavouriteColour;

  public function outputVars() : void {
    foreach($this as $var => $value) {
      echo "$var is $value\n";
    }
  }

 public function __construct(string $par1,string $par2, string $par3, int $par4, string $par5) : void {
   print("Creating person $par1\n");
   $this->FirstName=$par1;
   $this->LastName=$par2;
   $this->Password=$par3;
   $this->Age=$par4;
   $this->FavouriteColour=$par5;
 }
}

$person1=new person("Bill", "Murphy", "1234", 29, "");
$person1->outputVars();
?>
```

Com es pot veure al codi anterior, gràcies a l'existència del constructor, ara podem passar els valors amb què volem inicialitzar les propietats d'un objecte simplement indicant-los com a paràmetres un rera l'altre en el moment de la creació de l'objecte: una manera molt més ràpida i còmoda de fer-ho que l'anterior, doncs. Fixeu-vos també que ara podem declarar totes les propietats de tipus *private*, ja que no caldrà establir el seu valor des de fora de la classe perquè ja se n'ocupa el constructor de fer-ho.

> NOTA: El constructor ha de ser marcat sempre com un mètode públic, ja que si no no podríem crear objectes d'una classe fora de la classe mateixa, cosa que no té gaire sentit
> NOTA: En el cas de crear un objecte a partir d'una subclasse que heredi d'una classe "mare", cal saber com se sel·lecciona el constructor a fer servir, ja que ambdues classes (la "mare" i la filla, o més si n'hi hagués una "àvia", etc) podrien incorporar un constructor propi cadascuna. El criteri és el següent: PHP mira si hi ha un constructor en la classe de l'objecte a crear: si existeix, el fa servir, però si no, va a buscar si a la classe immediatament superior (la "mare") n'hi ha, de constructor: si sí, el fa servir, però si no, va a buscar llavors si a la classe immediatament superior (l'"àvia") n´hi ha, de constructor, i així fins arribar al final de la jerarquia.
> NOTA: Existeix la possibilitat de reutilizar codi d'un constructor de la classe "mare" en una subclasse. La idea és que potser el constructor de la subclasse, que serà el que serà cridat a la pràctica, només hagi d'ampliar amb codi extra el que ja fa el constructor superior. Copiar-pegar seria una solució però és molt més elegant utilitzar per això la sentència *parent::__construct();* la qual significa "més a la classe 'mare' i allà executa el seu constructor"

També existeix el concepte de destructor, que representa un mètode que és automàticament cridat just abans de què un objecte de la classe corresponent sigui destruit. Per a què un mètode sigui un destructor només cal que sigui públic, no accepti cap paràmetre i s'anomeni *__destruct()*. Però, ¿quan es destrueix un objecte? Doncs es fa automàticament quan l'script on van ser creats finalitza (gràcies al recolector de brossa automàtic de PHP) però també es pot fer manualment amb la mateixa funció que vam aprendre per destruir variables, mitjançant *unset($nomObjecte);* Per tant, executant aquesta funció cridarem automàticament al destructor que hi hagi definit a la classe de l'objecte en qüestió, i tot seguit l'objecte desapareixerà.

Els objectes, a l'igual que les variables simples i els arrays, poden ser passats també com a paràmetres d'una funció. No obstant, cal tenir en compte que, a diferència de les variables simples i els arrays, els objectes es passen "**per referència**" i no "**per valor**". Que una variable/array es passi com a paràmetre "per valor" en una funció significa que el codi intern de la funció en qüestió tracta aquest paràmetre com una variable local còpia de la variable original; per tant, els canvis que es produeixin en el

valor d'aquesta variable local no afectaran al valor de la variable original indicada en la crida a la funció. En canvi, que un objecte es passi com a paràmetre "per referència" en una funció significa que el codi intern de la funció tracta aquest paràmetre com un únic objecte, tant fora com dins de la funció en qüestió; per tant, els canvis que es produeixin en les propietats de l'objecte dins del codi intern de la funció afectaran a l'objecte en sí en qualsevol lloc del nostre script. En altres paraules: quan es passa un objecte com a paràmetre d'una funció, qualsevol canvi que se li faci es reflectirà fora de la funció. Per comprovar-ho, podem executar el codi següent, on es pot veure com el valor de la propietat "FirstName" de l'objecte passat com a paràmetre a la funció "canviarnom()" canvia de forma permanent en l'objecte original un cop aquesta funció ja ha finalitzat:

```
<?php
class person {
  public string $FirstName;

  public function __construct(string $param1) : void {
    $this->FirstName=$param1;
  }
}

function canviarnom(person $unapersona, string $nom) : void {
  $unapersona->FirstName=$nom;
}

$person1=new person("Bill");
echo $person1->FirstName."\n";
canviarnom($person1,"Pepe");
echo $person1->FirstName."\n";
?>
```

No obstant, a vegades ens convindrà treballar només sobre còpies d'objectes sense que afecti a l'estat de l'objecte original. Per aconseguir això, hem de fer servir la paraula clau *clone* , la qual retorna una còpia completa de l'objecte indicat darrera seu. En concret, si tornem a executar el mateix codi anterior només que substituint la línia *canviarnom($person1,"Pepe");* per *canviarnom(**clone** $person1,"Pepe");* -o, de forma alternativa, ja que és el mateix però més explícitament, per les línies *$person2=clone $person1; canviarnom($person2,"Pepe");* - veurem que el valor de la propietat de l'objecte *$person1* no canvia

> NOTA: En realitat, hi ha una diferència entre escriure *canviarnom(clone $person1,"Pepe");* i *$person2=clone $person1; canviarnom($person2,"Pepe");* : en el primer cas el clon de *$person1* es destrueix automàticament en acabar-se l'execució de la funció *canviarnom()* -invocant-se per tant al seu destructor, si n'hi hagués-, mentre que en el segon cas no perquè el clon s'ha creat directament en el codi principal.

Internament, el que fa la sentència *clone* és és copiar el valor de totes les propietats de l'objecte original (a més dels seus mètodes) a un nou objecte en memòria mitjançant la funció interna *__clone()*. La gràcies és que podem sobreescriure aquesta funció interna per una de pròpia per realitzar tasques extra quan es realitza la clonació (es pot pensar que és com si fos un constructor específic per objectes copiats).Així, podríem tenir, per exemple, un mètode de la classe tal com el següent, el qual s'executaria quan un objecte d'aquesta classe es clonés (el que fa concretament és que l'objecte clonat tingui el mateix valor de la propietat "Name" que l'original però seguit de la cadena "++"; així, un clon d'un objecte amb aquesta propietat igual a "Pepe" tindrà llavors aquesta propietat amb el valor de "Pepe++", i un clon d'aquest darrer en tindrà el valor de "Pepe++++", etc.

```
public function __clone() : void {
   $this->Name .= '++';
}
```

En relació a aquest tema dels clons, cal saber també que els operadors "==" i "===" evaluen coses diferents quan es compara un objecte amb un altre. Concretament, el primer comprova que els valors de les propietats dels dos objectes siguin els mateixos mentre que el segon comprova que l'objecte en sí (en memòria) sigui el mateix (és a dir, que no siguin clons). Per tant, un codi com el següent mostrarà la frase "Els dos objectes són iguals" però no pas "Els dos objectes són un").

```php
<?php
   class foo { }
   $wom = new foo();
   $bat = clone $wom;
   if ($wom == $bat) { echo "Els dos objectes són iguals\n";}
   if ($wom === $bat){ echo "Els dos objectes són un\n";}
?>
```

NOTA: L'operador "===" no només serveix per comparar objectes; si s'utilitza amb valors simples és equivalent a "==" però amb l'avantatge que si la variable a comparar val *null* o *false* no es retornen falsos positius quan es compara amb una cadena buida o el valor 0. En aquest sentit també són interessants tenir en compte -tot i que amb l'operador "===" no serien necessàries, les funcions *is_null($nomVariable);* o *is_bool($nomVariable);* perquè *isset($nomVariable)* dona fals negatiu si la variable indicada val *false*. Tot això es pot veure en el següent codi d'exemple:

```php
<?php
$x = 0;
$y = null;

// Is $x null?
if($x == null){
    print('Oops! $x is 0, not null!');
}

// Is $y null?
if(is_null($y)){
    print('Great, but could be faster.');
}

if($y === null){
    print('Perfect!');
}

// Does the string abc contain the character a?
if(strpos('abc', 'a')){
    // GOTCHA!  strpos returns 0, indicating it wishes to return the position of the first character.
    // But PHP interpretes 0 as false, so we never reach this print statement!
    print('Found it!');
}

//Solution: use !== (the opposite of ===) to see if strpos() returns 0, or boolean false.
if(strpos('abc', 'a') !== false){
    print('Found it for real this time!');
}
?>
```

Ja vam veure que les **constants** a PHP les podíem definir amb la sentència *define()*; No obstant, si volem definir una propietat d'una classe que sigui constant, aquesta sentència no ens serveix perquè aquesta sentència s'interpreta sempre en temps d'execució de l'script (és a dir, "al vol"), no pas en temps de la compilació prèvia del codi que es realitza abans de la interpretació pròpiament dita (i que és quan es reconeixen les diferents classes que es faran servir durant l'execució i els seus membres). Per tant, si fem servir *define()* per declarar una propietat constant, en el moment de voler utilitzar-la no estarà reconeguda (perquè no s'haurà "compilat") i obtindrem un error.

És per això que per definir una propietat constant hem de fer servir una sintaxi alternativa: afegir la paraula clau *const* Cal saber que les constants definides així (les quals, com passava amb *define()*, no tenen cap símbol "$" al començament del seu nom) seran sempre públiques i estàtiques. Així doncs, dins del codi propi de la classe, ens podrem referir a una constant que hi pertanyi fent servir la sintaxi *self::nomConstant* , com qualsevol altre membre estàtic. Fora del codi propi de la classe, podran ser utiltizades amb la sintaxi *nomClasse::nomConstant* A continuació es veu un exemple d'ús:

```php
<?php
class unaclasse {
  const abc=1;   //Les constants no tenen tipus!!!
  public static function veure() : void {
      echo "La constant vista des de dins:". self::abc ."\n";
  }
}
unaclasse::veure();
echo "La constant vista des de fora:". unaclasse::abc ."\n";
//La línia següent donaria un error perquè les constants no poden canviar de valor
//unaclasse::abc=2;
?>
```

Respecte el dubte de quan utilitzar *define()* o *const*, en principi, a excepció del cas concret de voler definir propietats constants, que ja hem dit que ho hem de fer amb *const*, en general és millor usar *define()* perquè és més flexible (com a valor -i com a nom!- admet expressions, es pot escriure dins d'un bloc *if*, etc)

Conceptes avançats de la POO: interfícies i "traits"

El llenguatge PHP no permet, tal com ja hem comentat, l'herència múltiple. No obstant, podem "sortejar" aquesta limitació mitjançant l'ús de les **interfícies**. Una interfície és una classe abstracta (és a dir, una classe on tots els seus mètodes són abstractes, el que significa que no es poden usar directament sinó que cal heredar-los en una subclasse primer) però on no s'hi poden definir propietats i, sobre tot, on només està definit el prototip dels seus mètodes (és a dir, l'"esquelet" on s'indica el seu nom, el nombre i tipus de paràmetres que rebrà i el tipus de valor de retorn) però no pas el seu codi intern; codi intern que, en canvi, haurà de ser implementat obligatòriament a la subclasse corresponent.

> NOTA: Com que, de fet, les interfícies no són pròpiament classes, i per tant, no hi ha un procés d'herència pròpiament dit quan una subclasse implementa una interfície, a partir d'ara anomenarem "classes" al que estem mencionant en aquests paràgrafs anteriors com a "subclasses".

Una interfície actua, doncs, com una "llista" de mètodes que una determinada classe que vulgui implementar-la ha d'"omplir de contingut". És a dir, en fer que una classe implementi una determinada interfície (o més, en això no hi ha cap problema!), el que estem dient és: "aquesta classe és capaç de fer tot el que la interfície *x* diu que hauria de fer"; en essència, doncs, utilitzar interfícies és una manera de crear contractes amb les nostres classes: una classe que implementi una interfície ha d'incorporar obligatòriament mètodes concrets que siguin públics per a tots i cadascun dels mètodes abstractes definits a la interfície (perquè, en cas contrari, no funcionaran)

Es veu clar, doncs, que si fem que una classe implementi diverses interfícies (això sí que es pot fer, repetim!), podem aconseguir que aquesta classe ofereixi funcionalitats diverses, cadascuna provinent (si més no la seva definició) d'una interfície diferent.

Definir una interfície és molt similar a definir una classe: consisteix bàsicament en utilitzar la paraula clau *interface* (en lloc de *class*) i de no definir cap cos (amb *{...}*) pels mètodes declarats al seu interior. A partir d'aquí, per fer que una classe qualsevol implementi una determinada interfície, caldrà indicar a la definició d'aquesta classe la paraula *implements*, així:

> NOTA: Els mètodes d'una interfície han de ser públics per pura definició, així que sense indicar-ho ja ho són per defecte. És per això que no s'indica cap modificador d'accés en la seva definició dins d'una interfície (ni tampoc *abstract* o *static*)

```
<?php
interface ivaixell {
   function navegar() : void;
   function amarrar() : void;
}
interface iavio {
   function despegar() : void;
   function aterrar() : void;
}
class hidroavio implements ivaixell,iavio {
   public function navegar() : void { echo "Sóc un hidroavió i navego\n"; }
   public function amarrar() : void { echo "Sóc un hidroavió i amarro\n"; }
   public function despegar() : void { echo "Sóc un hidroavió i despego\n"; }
   public function aterrar() : void { echo "Sóc un hidroavió i aterro\n"; }
}
class avio implements iavio {
   public function despegar() : void { echo "Sóc un avio i despego\n"; }
   public function aterrar() : void { echo "Sóc un avio i aterro\n"; }
}
$hidro1 = new hidroavio();
$avio1 = new avio();
print_r(get_class_methods($hidro1));
print_r(get_class_methods($avio1));
$hidro1->despegar();
$avio1->despegar();
?>
```

Tal com es pot observar, al codi anterior s'ha fet servir la funció *get_class_methods($nomObjecte)*; , la qual retorna un array els elements del qual tenen com a valor els noms dels mètodes (públics) que té l'objecte (o classe) indicat. Una altra funció interessant que convé conéixer és *get_declared_interfaces();* , la qual retorna un array de totes les interfícies disponibles que podríem implementar en les nostres classes.

Les interfícies també es poden definir com un tipus de paràmetre que pot admetre una funció. I això és interessant perquè, d'aquesta manera, qualsevol objecte d'una classe que implementi la interfície indicada es podrà indicar com a valor concret d'aquell paràmetre de la funció. És a dir, no cal que en una funció l'objecte passat com a paràmetre sigui d'una classe concreta (com hem vist fins ara a exemples anteriors) sinò que pot ser suficient, si així s'indica en la definició de la funció, amb què implementi una determinada interfície, fet que fa que l'abast dels tipus d'objecte que pugui rebre aquella funció com a paràmetre sigui molt major. Veiem-ho amb un exemple:

```
<?php
interface ivaixell {
   function navegar() : void;
}
class barca implements ivaixell {
   public function navegar() : void { echo "Sóc una barca i navego poc a poc\n"; }
}
class trasatlantic implements ivaixell {
   public function navegar() : void { echo "Sóc un trasatlàntic i navego molt ràpid\n"; }
}
function dirhola(ivaixell $unainterf) : void {
    $unainterf->navegar();
}
$barca1 = new barca();
$trasa1 = new trasatlantic();
dirhola($barca1);
dirhola($trasa1);
?>
```

> NOTA: Si es vol que alguns dels mètodes declarats en una interfície tingui un codi predefinit, llavors no podrem fer ús d'interfícies (ja que aquestes no permeten definir el cos de cap mètode) sinó d'una classe abstracta. Les classes abstractes (que s'identifiquen per la presència de la paraula *abstract* davant de la paraula *class*) es poden utilitzar com a interfícies però també s'hi poden afegir mètodes totalment funcionals. Aleshores, ¿per què no utilitzar classes abstractes tot el temps? Perquè és més senzill implementar múltiples interfícies dins d'una classe que crear una cadena d'herència llarga i incòmoda.

Una altra forma de "sortejar" en PHP la limitació que una classe no pugui heretar membres de vàries classes "mare" a la vegada (l'anomenada herència múltiple) és mitjançant l'ús dels anomenats "**traits**". Els "traits" permeten reutilitzar lliurement conjunts de mètodes (per definició, públics, òbviament) en diferents classes independents entre sí.

Definir un "trait" és molt similar a definir una classe: consisteix bàsicament en utilitzar la paraula clau *trait* (en lloc de *class*). A partir d'aquí, per fer que una classe qualsevol pugui fer ús dels mètodes definits en un determinat "trait", cal que al principi del seu codi intern hi aparegui la sentència *use nomTrait,unAltre,...;*, així:

```
<?php
trait Hello {
   function sayHello() : string {
      return("Hello");
   }
}
trait World {
   function sayWorld() : string {
      return("World");
   }
}
```

```
class MyWorld {
   use Hello, World;
}
$world = new MyWorld();
echo $world->sayHello() . " " . $world->sayWorld() . "\n";
?>
```

Com es pot veure, al codi anterior s'han definit dos "traits", el primer (anomenat "Hello") només sap mostrar la paraula "Hello" i el segon (anomenat "World") només sap mostra la paraula "World". A la classe s'han aplicat els dos "traits" de forma que els objectes generats a partir d'ella siguin capaços de mostrar ambdues paraules mitjançant sengles mètodes obtinguts de cada "trait", com efectivament passa.

NOTA: Es poden crear "traits" a partir de la unió de diversos altres "traits"; això és molt útil per poder incloure dins d'una classe només un "trait" "gran" en lloc de diversos petits. A continuació es mostra com fer això modificant el codi anterior:

```
<?php
trait HelloWorld{ use Hello,World;  }
trait Hello {  function sayHello() : string { return("Hello");  }  }
trait World {  function sayWorld() : string { return("World");  }  }
class MyWorld {  use HelloWorld;  }
$world = new MyWorld();
echo $world->sayHello() . " " . $world->sayWorld() . "\n";
?>
```

Els "traits" es poden entendre com un mecanisme que permet, a mode de "puzzle", "copiar-pegar" definicions predefinides de mètodes dins de les classes que es vulguin crear en un moment donat. És important, recordar, però, que un "trait" no pot hereder, ni pot implementar interfícies, ni es pot instanciar: el seu únic objectiu és servir com a suport a les classes, no suplantar-les. El que sí que poden fer és, ja que els seus mètodes "s'injecten" a l'interior de la classes que les utilitzi com si fossin seus, accedir a qualsevol propietat o mètode *private* o *protected* d'aquella classe en qüestió.

Podria ser possible que algun mètode pertanyent a un "trait" s'anomenés igual que un mètode propi de la classe que fes servir aquest "trait". En aquest cas cal saber que els mètodes definits en un "trait" tenen preferència respecte els mètodes heredats de la classe "mare" però no sobre els mètodes definits a la pròpia classe. A continuació es mostra un exemple, on tenim una classe "Greeting" derivada de "Base", i ambdues classes tenen un mètode anomenat "sayHello()" però amb implementacions diferents. A més, hem inclòs el "trait" "Hello" a la classe "Greeting", el qual ofereix dos mètodes, "say()" i "sayBase()", el primer dels quals crida "sayHello()", mètode que existeix en ambdues classes així com en el "trait" però que podem veure a la sortida del programa que l'invocat és, tal com hem dit al principi d'aquest paràgraf, el de la classe filla.

NOTA: Si necessitéssim fer referència al mètode des de la classe pare, ho podríem fer utilitzant la paraula clau *parent::* tal com es mostra al mètode "sayBase()".

```
<?php
trait Hello {
   function sayHello() : string { return "Trait Hello\n";  }
   function say() : void { echo $this->sayHello();  }
   function sayBase() : void { echo parent::sayHello();  }
}
class Base {
   function sayHello() : string { return "Base Hello\n";  }
}

class Greeting extends Base {
   use Hello;
   function sayHello() : string { return "Hello\n";  }
}

$g =  new Greeting();
$g->say();         // Hello
$g->sayBase(); // Base Hello
?>
```

També podria ser possible que mètodes pertanyents a diferents "traits" tinguin el mateix nom i siguin utilitzats per una mateixa classe. En aquest cas, és necessari triar quin d'aquests mètodes es vol utilitzar mitjançant la paraula clau *insteadof*, així:

```
<?php
trait Game { function play() : void { echo "Mee mee!\n"; } }
trait Music { function play() : void { echo "Pii pii!\n"; } }
class Player {
   use Game, Music { Music::play insteadof Game; }
}
$player = new Player();
$player->play(); //Pii pii
?>
```

En l'exemple anterior hem hagut de triar un mètode sobre l'altre, però si encara volem tenir accés al mètode no triat, el que hem de fer és assignar-li un "àlies", amb la paraula clau *as*, així (en aquest codi, variació de l'anterior, seguim utilitzant el mètode "play" del "trait" "Music" però hem assignat un àlias (anomenat "gameplay") al mètode "play" del "trait" "Game" per tal de poder-lo utilitzar sense conflicte:

```
<?php
trait Game { function play() : void { echo "Mee mee!\n"; } }
trait Music { function play() : void { echo "Pii pii!\n"; } }
class Player {
   use Game, Music {
      Music::play insteadof Game;
      Game::play as gameplay;
   }
}
$player = new Player();
$player->play(); //Pii pii
$player->gameplay(); //Mee mee
?>
```

> NOTA: Per obtenir un array de tots els noms dels "traits" que pot implementar una determinada classe, es pot fer servir el mètode *$obj->getTraitNames()* on *$obj* és l'objecte obtingut en haver executat prèviament la sentència *$obj= new ReflectionClass("nomClasseAInvestigar");* Per més informació sobre el constructor *ReflectionClass* i els seus possibles múltiples usos, consulteu https://www.php.net/manual/es/class.reflectionclass.php o https://diego.com.es/reflection-en-php

Excepcions

En lloc de simplement aturar l'script quan ocorre un error fatal i irrecuperable (fent ús de funcions com *die()* o similars), l'intèrpre PHP disposa d'un altre mecanisme més sofisticat que permet processar de forma personalitzada els possibles errors d'execució detectats: la gestió d'excepcions. Una excepció és un objecte creat per l'intèrpret PHP automàticament quan aquest detecta un error que conté, en forma de propietats i mètodes específics, tota la informació necessària per avaluar el tipus d'error concret detectat. La gràcia de fer servir el mecanisme d'excepcions és que un cop es detecta l'error corresponent (o en altres paraules, "es captura l'excepció"), es podran executar unes determinades línies de codi "ad-hoc" per tal de pressumiblement, "amortiguar" l'efecte de l'error i redirigir llavors el fluxe d'execució de l'script cap a altres parts de codi controlades, fent així que el nostre script sigui més robust.

Per a què es pugui capturar una excepció, el codi susceptible de generar l'error ha d'estar escrit dins d'un bloc *try { ... }* , a continuació del qual haurà d'escriure's el/s codi/s "amortiguador"/s corresponent/s, dins d'un (o més) bloc/s *catch(tipusExcepcio $nomObjecte) { ... }* (cada bloc *catch* es correspondrà a un determinat tipus d'excepció detectada -l'intèrpret incorpora ja molts tipus predefinits però es poden crear d'altres a partir de l'herència-, de fet, l'objecte indicat entre parèntesi representa l'excepció del tipus indicat, les propietats i mètodes del qual es podran fer servir llavors dins del bloc de codi en qüestió; d'altra banda, si no existís el bloc *catch* pertinent, llavors sí que s'interromprà l'execució de l'script amb un error del tipus "Uncaught excepction..."). Opcionalment, després dels blocs *catch* es pot indicar un bloc *finally { ... }* , el cual conté codi que sempre s'executarà després del bloc *try*, s'hagi capturat alguna excepció o no.

D'altra banda, existeix la possibilitat, dins del codi *try* , de provocar manualment la creació d'una excepció (i, per tant, la seva corresponent "captura" en el bloc *catch* adient) amb la sentència específica *throw new Exception("Missatge");* on "Exception" és el constructor d'una excepció genèrica (però podríem utilitzar-ne algun altre constructor més específic per tipus concrets d'excepcions) i el valor del seu únic paràmetre serà assignat, dins del bloc *catch* corresponent, com a valor del mètode *getMessage()* de l'objecte-excepció en qüestió, que estarà disponible en aquell bloc. La utilitat de la sentència *throw* és forçar a codi on un possible error per defecte no emetria cap excepció a que l'emeti (i, per tant, poder redireccionar així el flux d'execució del programa).

> NOTA: Per saber la llista dels diferents tipus específics d'excepcions disponibles per defecte al llenguatge PHP, així com la llista dels seus diferents mètodes i propietats (tant heredats com propis) es pot consultar la documentació oficial a https://www.php.net/manual/en/class.exception.php i també a https://www.php.net/manual/en/spl.exceptions.php

Veiem-ne un exemple: en el següent codi primer es defineix el codi d'una funció que serveix per invertir el valor introduit per paràmetre sempre i quan aquest no sigui 0 (perquè 1/0 és una indeterminació matemàtica); en aquest cas, però, provocarà una excepció (genèrica). A continuació, comença l'script pròpiament dit, que consisteix en l'execució de la funció anteriorment definida varis cops, tots ells dins d'un codi *try*, fins que en una determinada invocació es genera l'excepció, donant pas llavors a l'execució del codi definit en -l'únic- bloc *catch* que existeix. L'important aquí és veure que l'execució de l'script no es para sinó que, un cop interpretat el bloc *catch*, el fluxe segueix de forma normal, i és per això que es mostra en pantalla la frase final "Hello world".

```
<?php
function inverse(int $x) : float {
    if($x==0) {    throw new Exception("Division by zero.");    }
    return(1/$x);
}
try {
    echo(inverse(5)."\n");
    echo(inverse(0)."\n");
} catch(Exception $e){
    echo("Caught exception: " . $e->getMessage() ."\n");
}
echo("Hello World\n");  //Continuo l'execució gràcies a què he aconsegut "capturar" l'excepció amb èxit
?>
```

Tal com hem dit, és possible tenir més d'un bloc *catch* definit per tal de poder detectar diferents tipus d'excepcions més enllà de la genèrica. Per exemple, al següent codi, variant de l'anterior, hem establert dos blocs *catch*, un específicament pensat per l'excepció generada quan es divideix un nombre entre 0 -l'anomenada *DivisionByZeroException*, tal com es pot confirmar a la documentació oficial mencionada en la nota anterior- i l'altra per qualsevol altre tipus d'excepció. En aquest sentit, cal vigilar amb l'ordre en què s'escriuen els blocs *catch* per tal de que siguin sempre de més específics a menys; això és perquè sempre s'executa només el codi d'un sol bloc *catch*: el bloc que sigui el primer que l'intèrpret PHP trobi on coincideixi el tipus de l'excepció indicat com a paràmetre de *catch* amb el tipus de l'excepció real gestionada en aquell moment: per tant, si posem el tipus genèric com a paràmetre del primer *catch*, sempre serà aquest el bloc que s'executarà i no pas els eventuals següents blocs *catch* que hi poguessin haver escrits després.

```
<?php
function inverse(int $x) : float {   return(1/$x);    }
try {
    echo(inverse(5)."\n");
    echo(inverse(0)."\n");
} catch(DivisionByZeroError $e){
    echo("Caught division by zero exception: " . $e->getMessage() . "\n");
} catch(Exception $e) {
    echo("Generic exception: " . $e->getMessage() ."\n");
}
echo("Hello World\n");
?>
```

NOTA: Noteu com apareix el missatge de "Caught division by zero exception"; si invertíssim els blocs *catch*, no obstant, llavors apareixeria el missatge "Generic exception", tal com s'ha comentat al paràgraf anterior

NOTA: Noteu també que en el codi anterior no hi ha cap sentència *throw* Això és perquè, tal com es pot veure, en realitat no cal: en el moment de voler calcular una divisió per 0, a les versions modernes de l'intèrpret PHP ja n'hi ha prou per a què generi l'excepció pertinent de forma automàtica, sense que haguem de fer-ho nosaltres. Això, no obstant, només serà cert en el cas de fer servir excepcions predeterminades del llenguatge, però no per excepcions definides per nosaltres, tal com veurem en el proper paràgraf

Les excepcions, tal com mostra la documentació, són classes que poden heredar-se per tal de crear excepcions pròpies, més enllà de les que proporcioni el llenguatge per defecte. Així, podrem definir excepcions personalitzades, sobreescrivint el cos (o valors) dels mètodes (o propietats) heredats, o creant-ne directament de nous. Per exemple, al següent codi definim un nou tipus d'excepció personalitzada anomenada "MiDBZError" que en lloc de declarar nous mètodes o altres elements, simplement assigna un valor predeterminat per a la propiertat predefinida *$message* (el valor de la qual sempre s'assimila al valor que s'hag indicat com a primer paràmetre en el constructor de l'excepció; si no s'indiqués cap valor per aquest primer paràmetre, llavors el missatge mostrat pel mètode *getMessage();* seria llavors precisament el valor establert per aquesta propietat *$message*). Es pot executar el codi següent per comprovar efectivament què és el que passa.

```
<?php
class MiDBZError extends DivisionByZeroError {
  protected $message="Cuidao!";
}
function inverse(int $x) : float {
    if($x==0) { throw new MiDBZError();  }
    return(1/$x);
}
try {
    echo(inverse(5)."\n");
    echo(inverse(0)."\n");
} catch(MiDBZError $e){
    echo($e->getMessage()."\n");
}
echo("Hello World\n");
?>
```

NOTA: Podeu veure més exemples d'ús d'excepcions a https://www.w3schools.com/php/php_exception.asp