# modèles pour l'accès aux données : les ORM

# entités et données persistantes

- **les entités** : objets métier du domaine utilisés par l'application pour la réalisation de ses fonctionnalités
  - classes, objets, identifiants
  - associations, agrégation, héritage
- **données persistantes** : données stockées de façon permanente et sûre, généralement à l'aide d'un système dédié, au sein de l’infrastructure
  - tables, clés primaires (PK), clés étrangères (FK), table pivot
  - collection mongo, fichier xml

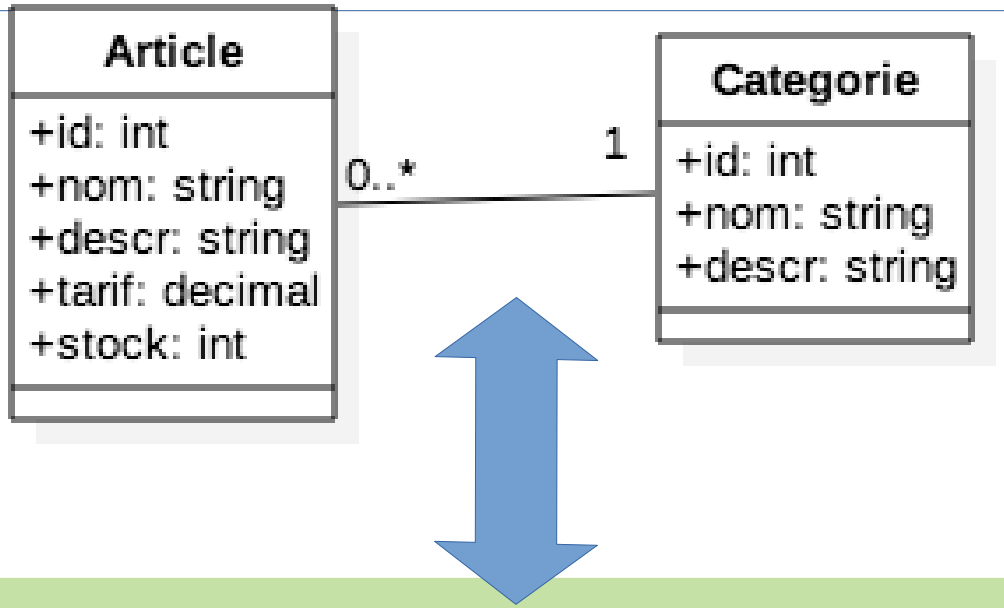
# le rôle d'un ORM

- **faire persister les entités, c'est à dire organiser** et **gérer** le lien

  **entités ↔ données persistantes**

- Utiliser des **objets** dans le code métier et les faire persister
  - créer des objets et les stocker dans la base
  - interroger la base et récupérer des objets
- **ORM (O**bject-**R**elational **M**apper) : librairie réalisant le lien entre les objets dans l'application et les données persistantes dans les tables de la base de données relationnelle
- Lien entre couche métier et infrastructure

# exemple




**Entités du domaine**

**article**

| id | nom | descr | tarif | stock | cat_id |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |

**categorie**

| id | nom | descr |
|---|---|---|
| | | |
| | | |

**base relationnelle**

# quelques ORMs

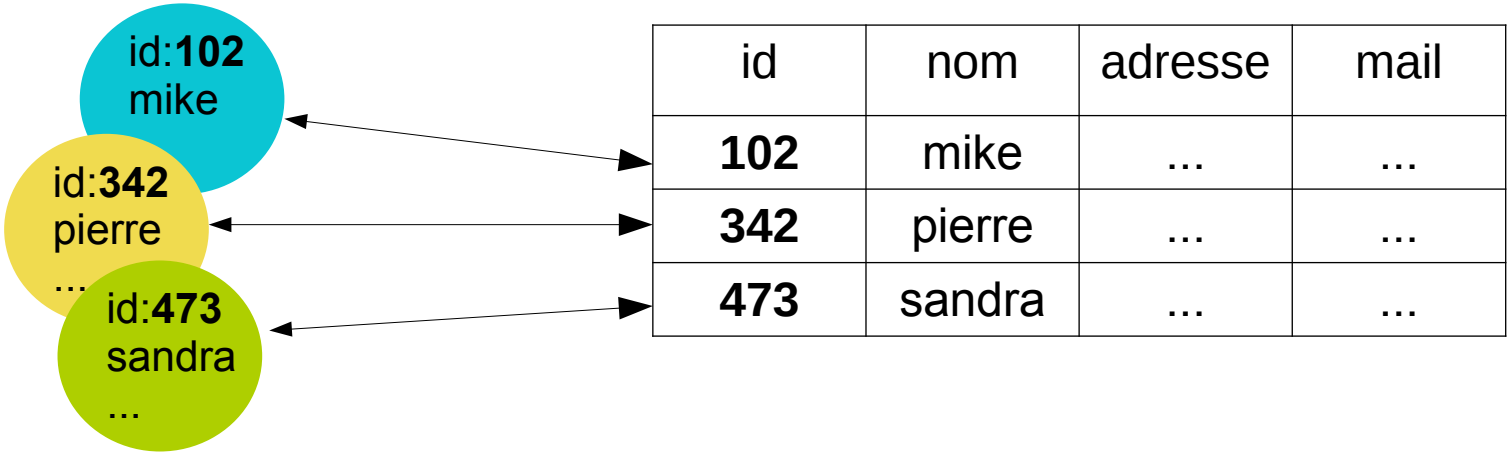
- java : JPA, hibernate, Java Data Object …
- python : SQLAlchemy, Django, Peewee …
- ruby : ActiveRecord, DataMapper, Sequel
- Node.js : TypeORM, Sequelize, Mongoose
- PHP : **Eloquent, Doctrine**, CakePHP ORM, RedBean, Atlas.Orm, CycleORM ...

# principes de base (I)

- un ORM propose des services ***génériques*** indépendants de la structure des données persistantes et des entités
- En se basant sur un modèle d’architecture structurant le lien métier-infrastructure

# principes de base (II)

- le lien entre entité et ligne de table est réalisé grâce à un identifiant d'objet unique pour chaque type d'objet, utilisé comme PK dans la table
  - Entier auto-incrémenté ou issu d’une séquence
  - Chaine de caractères générées aléatoirement (UUID)




| id | nom | adresse | mail |
|---|---|---|---|
| **102** | mike | ... | ... |
| **342** | pierre | ... | ... |
| **473** | sandra | ... | ... |

# principes de base (III)

- Un ORM se base sur un **modèle de conception** : principe **architectural** organisant les différentes classes et structurant le lien métier->infra
- **Deux modèles :**
- **Active Record :** Eloquent, RedBean, ActiveRecord
- **Repository** ou **DataMapper** : Hibernate, Doctrine

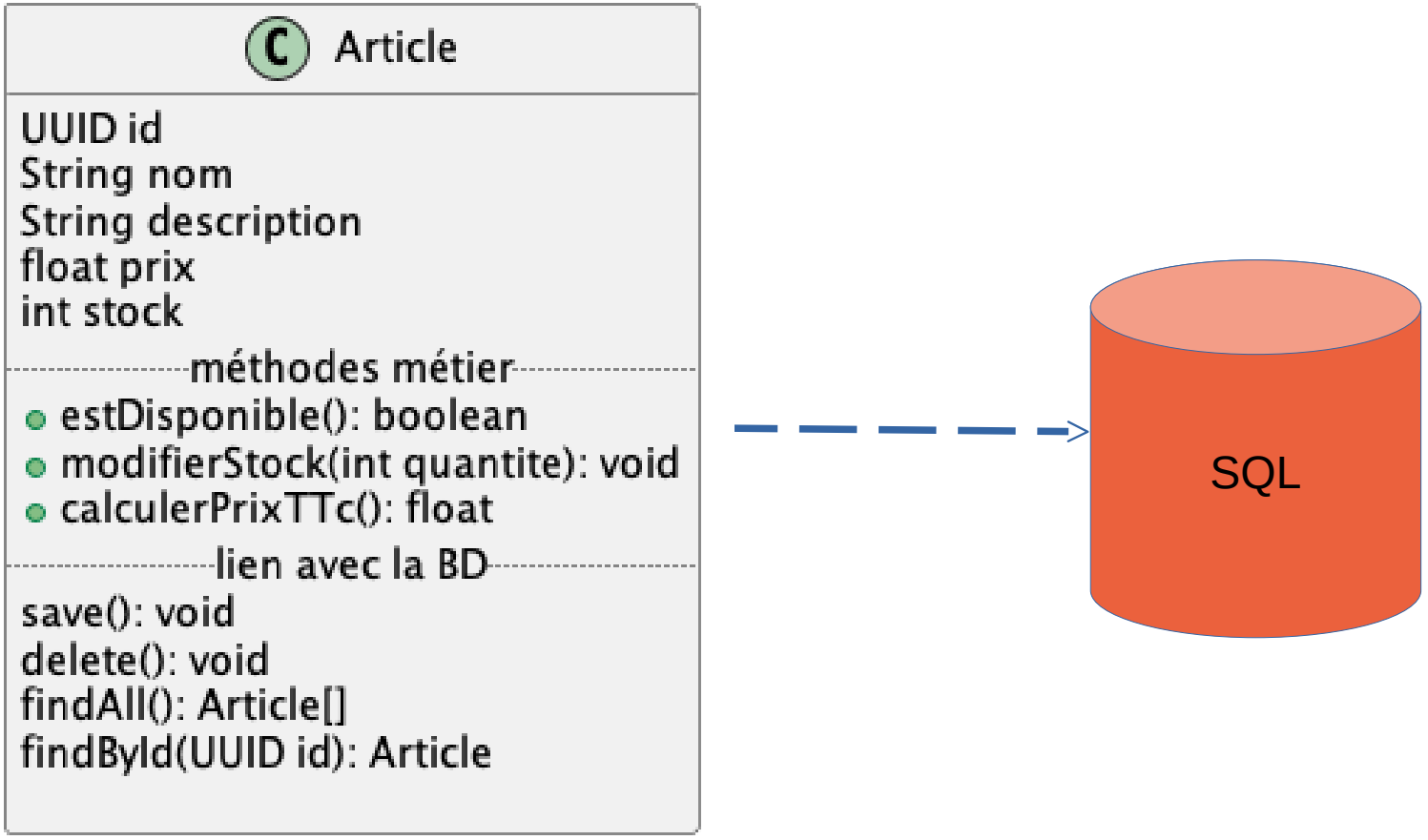
# Le Modèle « **Active Record** »

- *Un objet correspond à une ligne dans une table, et encapsule l'accès aux données et les fonctionnalités du domaine sur ces données*

- *1 table dans la base* <==> *1 classe dans le modèle*
- 1 ligne dans une table <==> 1 objet
- 1 colonne <==> 1 attribut dans la classe

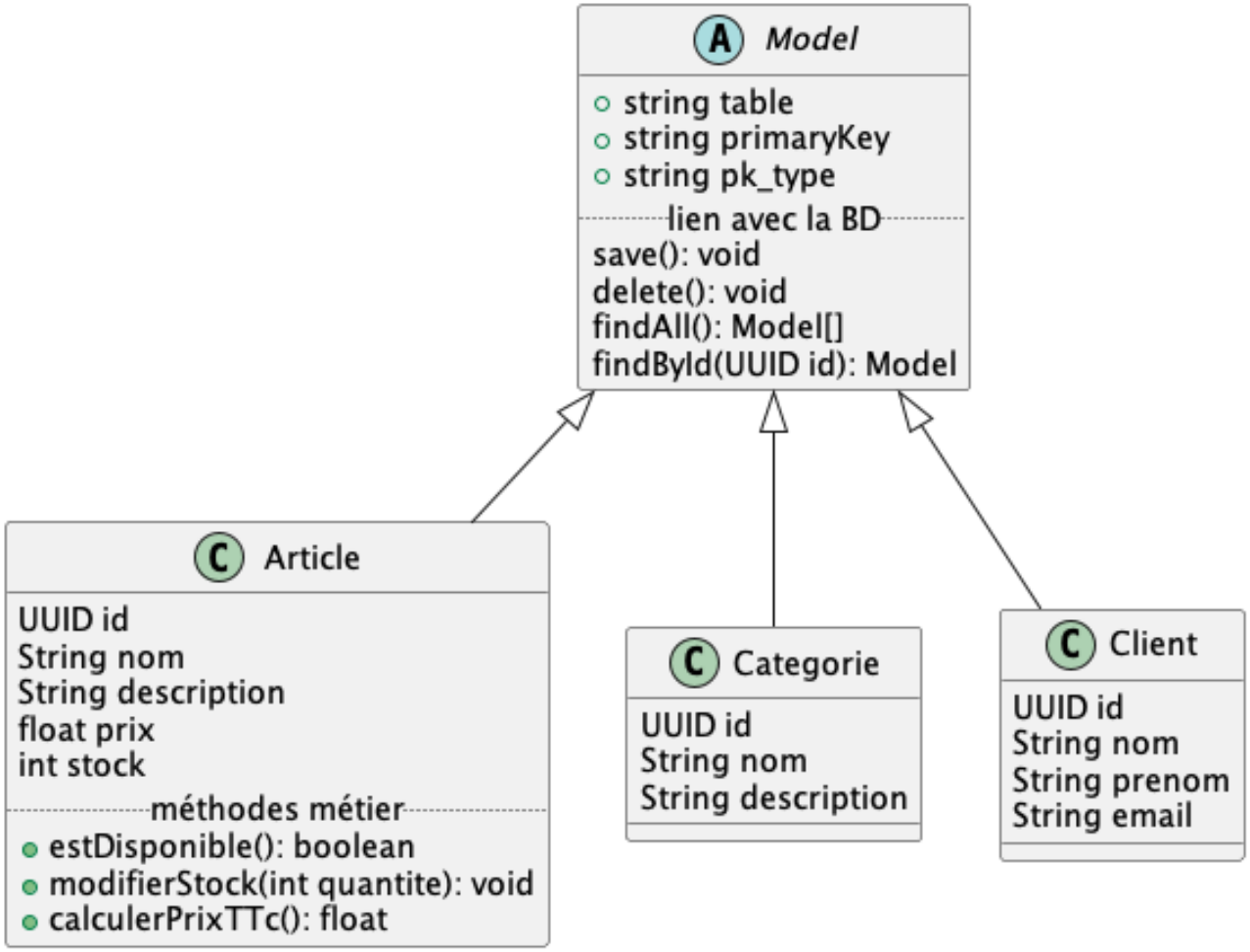
# active record

- les objets métiers sont étendus avec les fonctionnalités d'interaction avec la BD



# ORM basé sur ActiveRecord

- Les entités sous-classent une classe abstraite implantant active record de façon générique



# Exemple avec Eloquent

```php
class Commande extends \Illuminate\Database\Eloquent\Model
{

    protected $table = 'commande';
    protected $primaryKey = 'id';
    public $timestamps = false;
    public $incrementing = false;
    public $keyType = 'string';


    public function items(): \Illuminate\Database\Eloquent\Relations\HasMany
    {
        return $this->hasMany(Item::class, 'commande_id');
    }

    public function calculerMontantTotal(): float
    {
       ...
    }
```

```php
public function accederCommande(string $idCommande): CommandeDTO
{
    try {
        $commande=Commande::where('id', $idCommande)->firstOrFail();
    } catch (ModelNotFoundException $e) {
        throw new ServiceCommandeNotFoundException("Commande $idCommande not found", 404, $e);
    }
    return $commande->toDTO();
}
```

# les plus

- Intérêt du Pattern « Active Record »
  - Regroupe dans une seule classe toutes les fonctionnalités liées à une table
  - structuration simple

- Utilisé dans Rails et Cake et de nombreux outils PHP dont Eloquent

# les moins

- Impose une correspondance stricte entre « entité » et « table », qui n’est pas toujours souhaitable
- Les entités héritent d’une classe définie par l’ORM : on impose une dépendance métier → infrastructure
  - Impossible de changer d'ORM sans re-définir toutes les entités
  - réutilisation des services métier difficile

# Cas d’utilisation

- Bien adapté pour des applications basées sur l’architecture MVC
- Non utilisable pour réaliser une architecture hexagonale car impossible d’inverser la dépendance métier → infra

# Programmer un Active Record

- Programmation ad-hoc
- Programmation générique

- On peut utiliser le modèle Active Record sans utiliser un ORM : on implante toutes les classes modèles

```php
class Article  {

    private $id, $nom, $description, $tarif, $stock;

    public function __construct( array $t=null) {
        /* initialiser les attributs */
    }

    public static function findById(Int $id) : Article {
        $pdo= new \PDO('dsn', 'user', 'pass');

        $sql = 'select * from article where id= ?';

        $stmt=$pdo->prepare($sql);
        $stmt->bindParam(1, $id, \PDO::PARAM_INT);
        if ($stmt->execute()) {
            $article_data = $stmt->fetch(\PDO::FETCH_ASSOC);
            return new \models\Article( $article_data );
        } else return null ;
  }
```

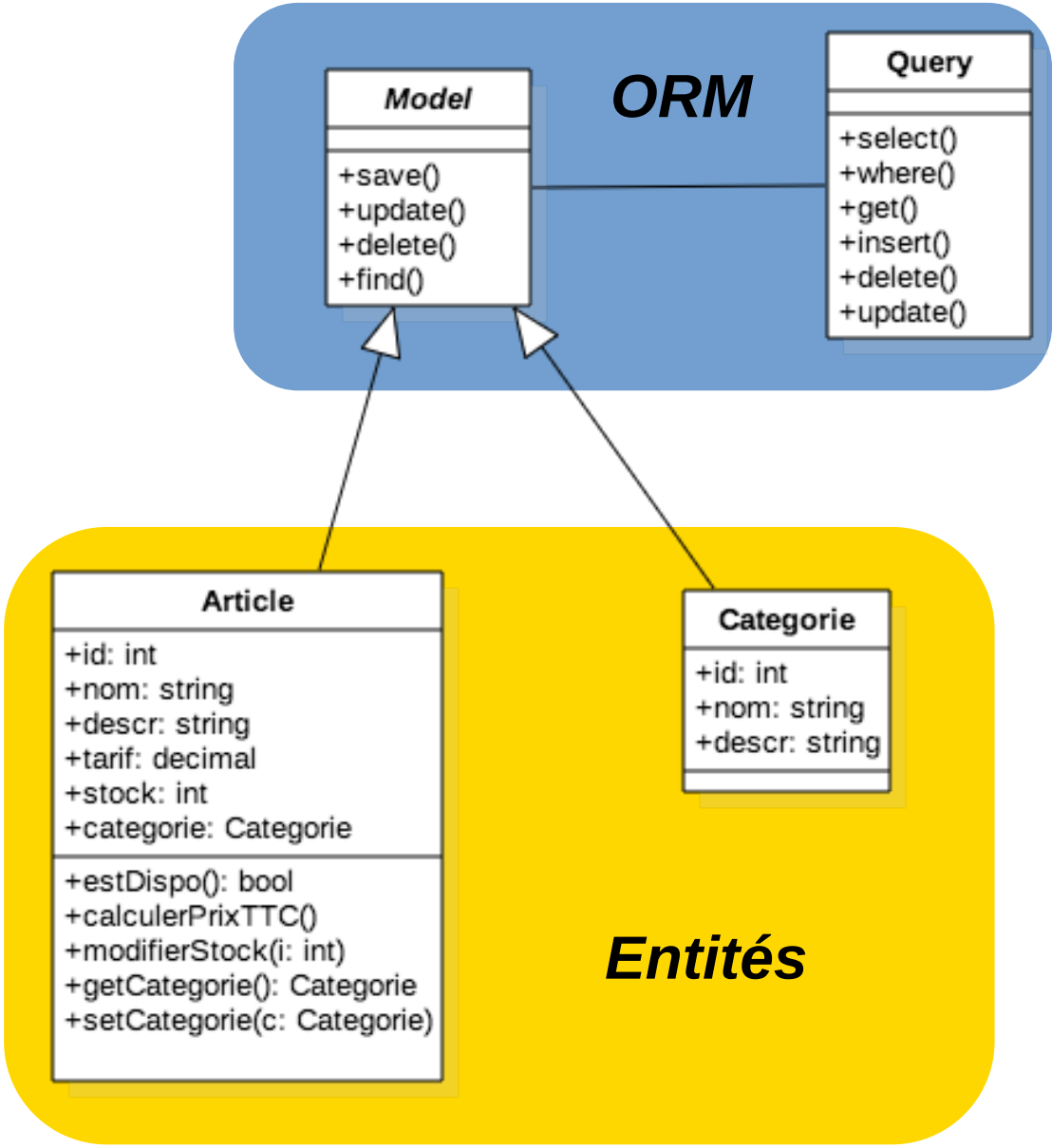
```php
class Article {
   /* … */

 private function insert() : int {
  $sql= 'insert into article(`nom`,`descr`,`tarif`,`stock`)
         values (?,?,?,?)';
  /* … */
 }
 private function update() : int {
  $sql = 'update article set .... where id= ?';
 }
 private function delete() : int {
  $sql = 'delete from article where id= ?';
 }

 public function estDisponible() : bool {
  return ($this->stock > 0) ;
 }
 public function modifierStock($nb) : Int {
  $this->stock += $nb;
  $this->update();
 }
```

# anatomie d'un orm : implantation générique de l'active record

- **Objectif** : construire un ORM générique pour éviter de programmer complètement les objets métiers
- l'orm fournit une classe abstraite implantant de manière générique toutes les fonctionnalités de lien objets-tables :
  - insertion, suppression, mise à jour d'objets métiers
  - recherche et accès à des objets métiers stockés dans la base

# implantation

- une **classe de dialogue SQL** permettant de construire des **requêtes**
- une **classe abstraite** transformant les résultats de requêtes en **objets** et les actions sur les objets en requêtes



# la classe de construction de requêtes

- classe implantant le pattern "chainage de méthodes" et permettant une construction incrémentale des requêtes

```php
$q = Query::table('article')
        ->select(['id', 'nom', 'descr', 'tarif'])
        ->where('tarif', '<', 1000)
        ->get();
```

- chaque méthode complète et retourne la **`Query`** en cours de construction en résultat
- la méthode **`get()`** génère la requête complète et l'exécute

# utilisation

```php
$q = Query::table('article') ;
$q = $q->where('tarif', '<', 1000);
$q = $q->select(['id', 'nom', 'descr', 'tarif']);
$res = $q->get();



$id = Query::table('article')
    ->insert(['nom'=>'grovelo', 'tarif'=>200]);

echo 'article inséré id : ' . $id .'\n';

$qd = Query::table('article')->where('id', '=', $id) ;
$qd->delete();


$q = Query::table('article')
    ->select(['id', 'nom', 'descr', 'tarif'])
    ->where('tarif', '<', 1000)
    ->get();
```

```php
class Query {

    private $sqltable;
    private $fields = '*';
    private $where = null;
    private $args = [];
    private $sql = '';

    public static function table( string $table) : Query {

        $query = new Query;
        $query->sqltable= $table;
        return $query;
    }

    public function select( array $fields) : Query {
        $this->fields = implode( ',', $fields);
        return $this;
    }
```

les différentes parties de la requête

le tableau d'arguments pour la requête préparée PDO

le texte complet, pour affichage si besoin

```php
public function where(string $col,
                      string $op,
                      mixed $val) : Query {
    /* … */
    $this->args[]=$val;
    return $this;

}

public function get() : Array  {

    $this->sql  = 'select '. $this->fields .
                  ' from ' . $this->sqltable;
      /* … */
    $stmt = $pdo->prepare($this->sql);
    $stmt->execute($this->args);
    return $stmt->fetchAll(\PDO::FETCH_ASSOC);
}
```

# la classe abstraite *Model*

- utilise la classe Query pour construire les requêtes SQL, et transforme les résultats de requêtes en objets
- Faite pour être sous-classée
- Les **sous-classes** concrètes déclarent les valeurs **spécifiques à chaque modèle** :
  - nom de la table
  - nom de la colonne clé primaire
- ces valeurs sont utilisées pour construire les requêtes

# Principes de base

- Le nom de la table associé est stocké dans un attribut statique dans chaque classe concrète,
- Le nom de la colonne clé primaire (PK) est également stocké dans un attribut statique dans chaque classe concrète

# utilisation

```php
class Article extends \hellokant\model\Model {

    protected static $table='article';
    protected static $idColumn='id';
}
```

```php
/** Utilisation  */
$a = new Article(); $a->nom = 'velo';  $a->tarif=273;
$a->insert();
print $a->id ;

$liste = Article::all();
foreach( $liste as $article) {
      print $article->nom;
}
```

# Principes de base

- les attributs ne sont plus déclarés dans les classes modèles mais sont **stockés dans 1 tableau** :

```
[ 'id'=> 23,
  'nom'=>'velo',
  'tarif'=> 450 ]
```

- l'accès aux attributs est programmé grâce aux méthodes magiques `__get()` et `__set()` afin de conserver les notations habituelles :
  - `$a->titre = 'velo' ;`
  - `$a->descr = 'beau vélo rouge et bleu' ;`

```php
class Model {
 protected static $table;
 protected static $idColumn = 'id';

 protected $atts = [];


 public function __construct(array $t = null) {
    if (!is_null($t)) $this->_atts = $t;
 }

 public function __get(string $name) : mixed {
    if (array_key_exists($name, $this->_atts))
        return $this->_atts[$name];
 }

 public function __set(string $name, mixed $val)) : void {

   $this->atts[$name] = $val;
 }
```

```
public function delete() {
 /* … */
return Query::table(static::$table)
    ->where( static::$idColumn, '=',
                        $this->atts[static::$idColumn] )
    ->delete()
}
```

```
public static function first(int $id): Model {

    $row=Query::table(static::$table)
            ->where(static::$idColumn, '=', $id)
            ->one();

    return new static($row);

}
```

```php
public static function all() : array {
    $all = Query::table(static::$table)->get();
    $return=[];
    foreach( $all as $m) {
        $return[] = new static($m);
    }
    return $return;
}
```