

# partie 1

---

modèles pour l'accès aux données : les ORM

# entités et données persistantes

- **les entités** : objets métier de la couche 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
  - tables, clés primaires (PK), clés étrangères (FK), table pivot
  - collection mongo, fichier xml

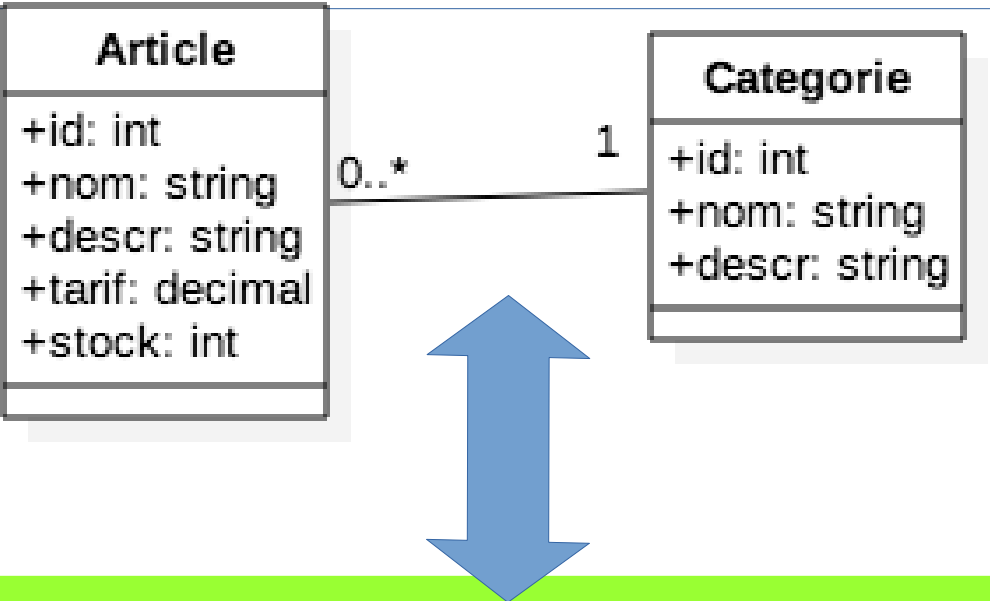
# le rôle d'un ORM

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

**entités ↔ données persistantes**

- Utiliser des **objets** pour manipuler et **interagir** avec la gestion de données :
  - créer des objets et les stocker dans la base
  - interroger la base et récupérer des objets
- **ORM (Object-Relational Mapper)** : librairie réalisant le lien entre les objets dans l'application et les données persistantes dans les tables de la base de données relationnelle
- construction rapide des objets métiers et du lien avec la couche persistance,

# exemple



**article**

id	nom	descr	tarif	stock	cat_id

**categorie**

id	nom	descr

**base relationnelle**

# un outil pour la persistance des objets : l'ORM

---

# 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 ...

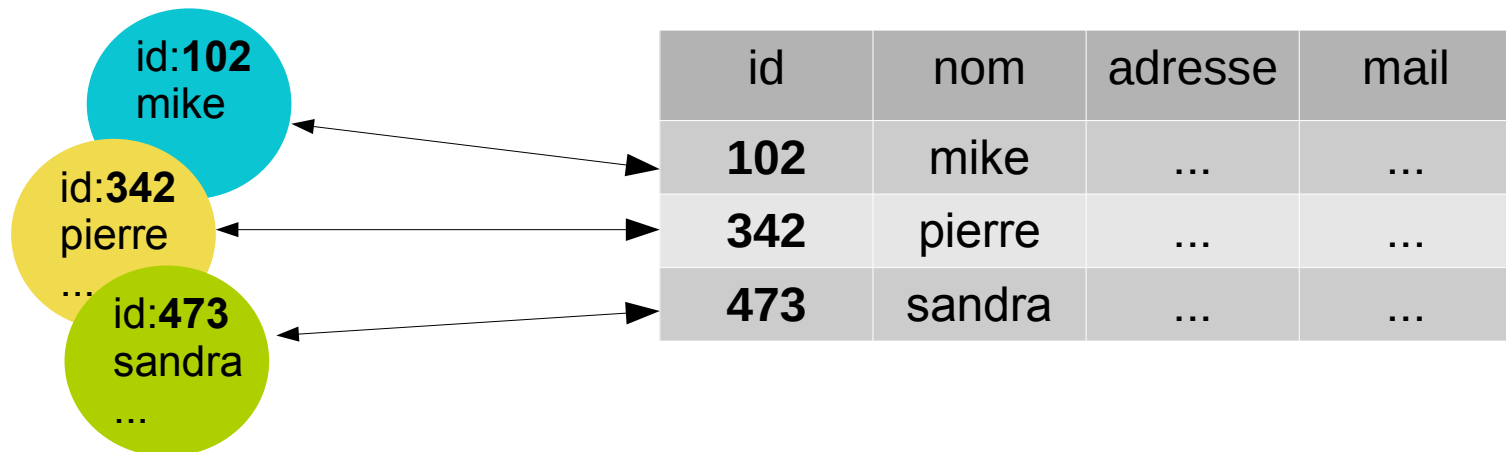
# principes de base (I)

---

- un ORM propose des services ***génériques*** indépendants de la structure des données persistantes et des objets métiers :
  - **sous-classer** des classes abstraites génériques
  - **générer** des classes à partir de descriptions
  - **configurer** des classes génériques

# principes de base (II)

- le lien entre objet et ligne de table est réalisé grâce à un identifiant d'objet unique pour chaque type d'objet, utilisé comme PK dans la table
  - soit généré par la BD au moment de l'insertion, en général par auto-incrément d'un entier
  - soit généré/calculé par l'application, souvent sous la forme d'une valeur random, par exemple un UUID





# principes de base (III)

---

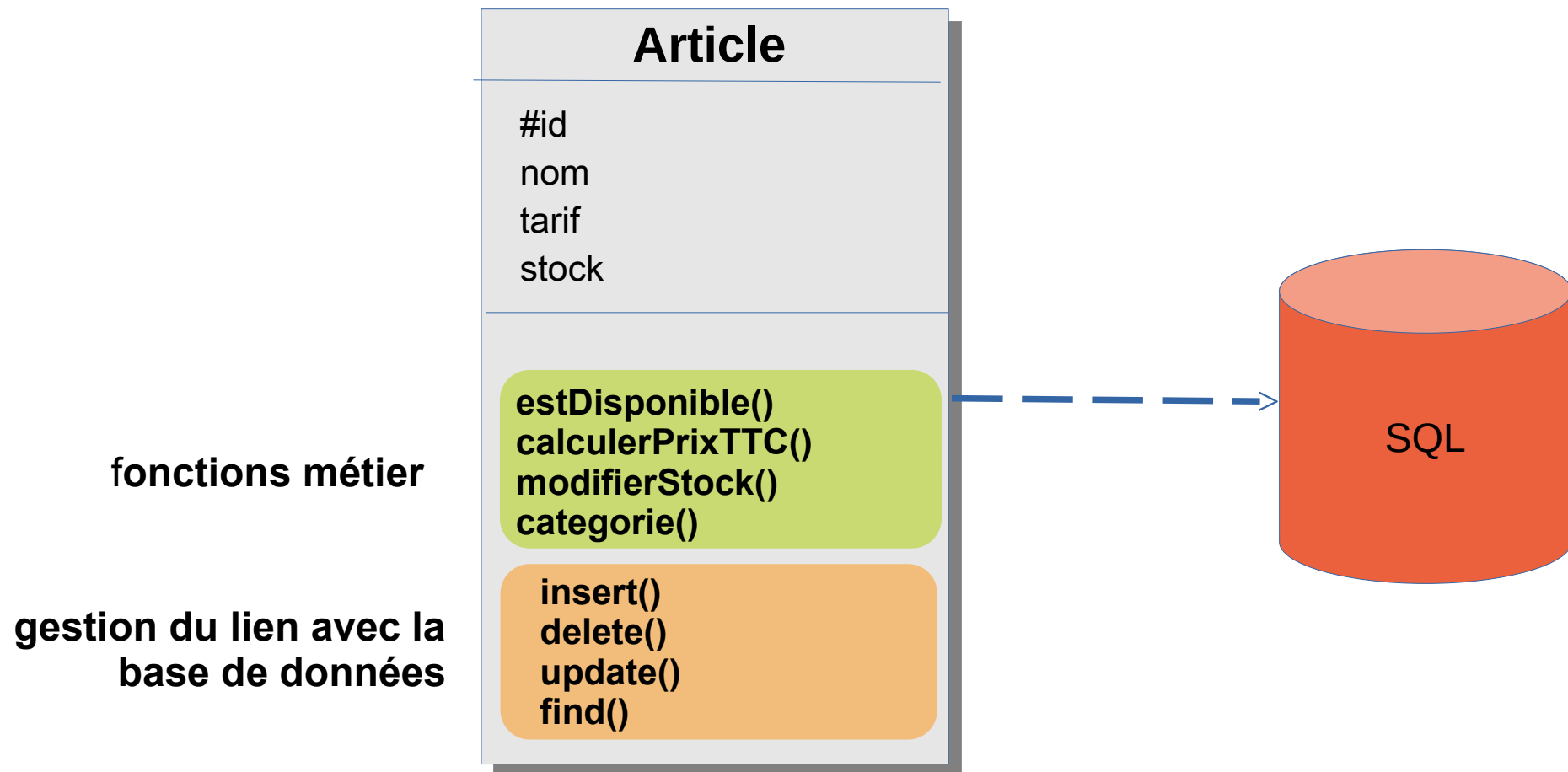
- Un ORM se base sur un **modèle de conception** : principe **architectural** organisant les différentes classes implantant la transformation objet<->relation
- **Exemples** :
- Modèle **Active Record** : Eloquent, RedBean, ActiveRecord
- Modèle **DataMapper** ou **DAO** : Hibernate, Doctrine

# 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



# 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

- Regroupe logique métier et accès aux données dans le même objet : les entités sont fortement couplés à l'ORM
  - Impossible de changer d'ORM sans re-définir toutes les entités
  - Impose une correspondance stricte entre « objet métier » et « table »
  - Rend difficile la réutilisation de la logique métier dans 1 autre application
  - Rend difficile l'évolution de l'accès aux données
  - Rend difficile la co-existence de plusieurs systèmes de stockage au sein de la même application métier

# 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

```
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 ;  
    }  
}
```

```
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();
}
```

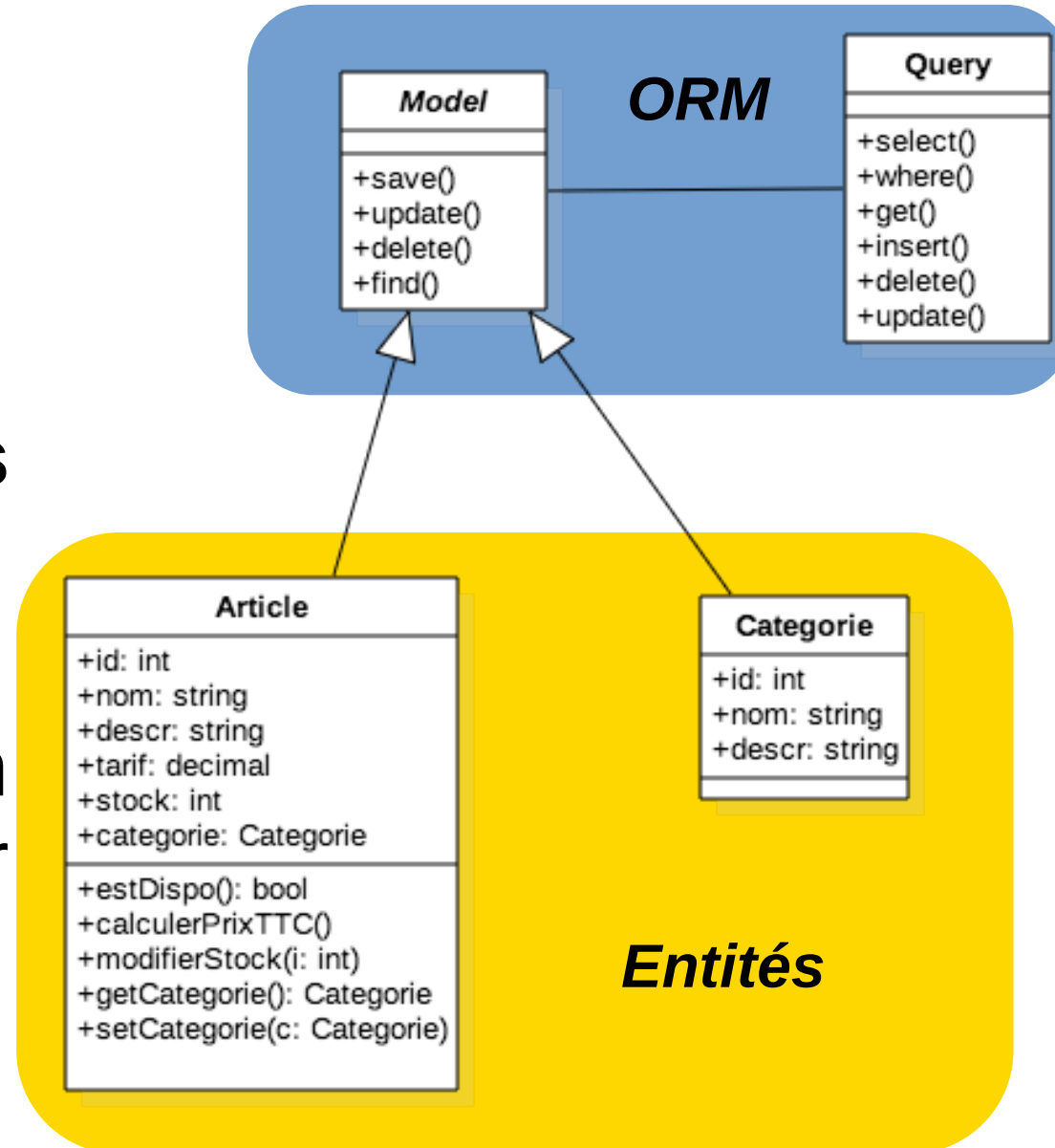


# anatomie d'un orm : implantation générique de l'active record

- **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

```
$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

```
$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();
```

```
class Query {
```

```
    private $sqltable;
```

```
    private $fields = '*';
```

```
    private $where = null;
```

```
    private $args = [];
```

```
    private $sql = '';
```

les différentes parties de la requête

le tableau d'arguments pour la requête  
préparée PDO

le texte complet, pour affichage si  
besoin

```
    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;
```

```
    }
```

```
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

```
class Article extends \hellokant\model\Model {  
    protected static $table='article';  
    protected static $idColumn='id';  
}
```

```
/** 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' ;`

```
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);  
}
```

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