

# PHP Classes and Objects

## Chapter 10

# Objectives

**1**

Object-Oriented  
Overview

**2**

Classes and Objects in  
PHP

**3**

Object Oriented Design

Section 1 of 3

# OBJECT-ORIENTED OVERVIEW

# Terminology

## Object-Oriented Terminology

The notion of programming with objects allows the developer to think about an item with particular **properties** (also called attributes or **data members**) and methods (functions).

The structure of these **objects** is defined by **classes**, which outline the properties and methods like a blueprint.

Each variable created from a class is called an object or **instance**, and each object maintains its own set of variables, and behaves (largely) independently from the class once created.

# UML Class diagram

By example

Every Artist has a

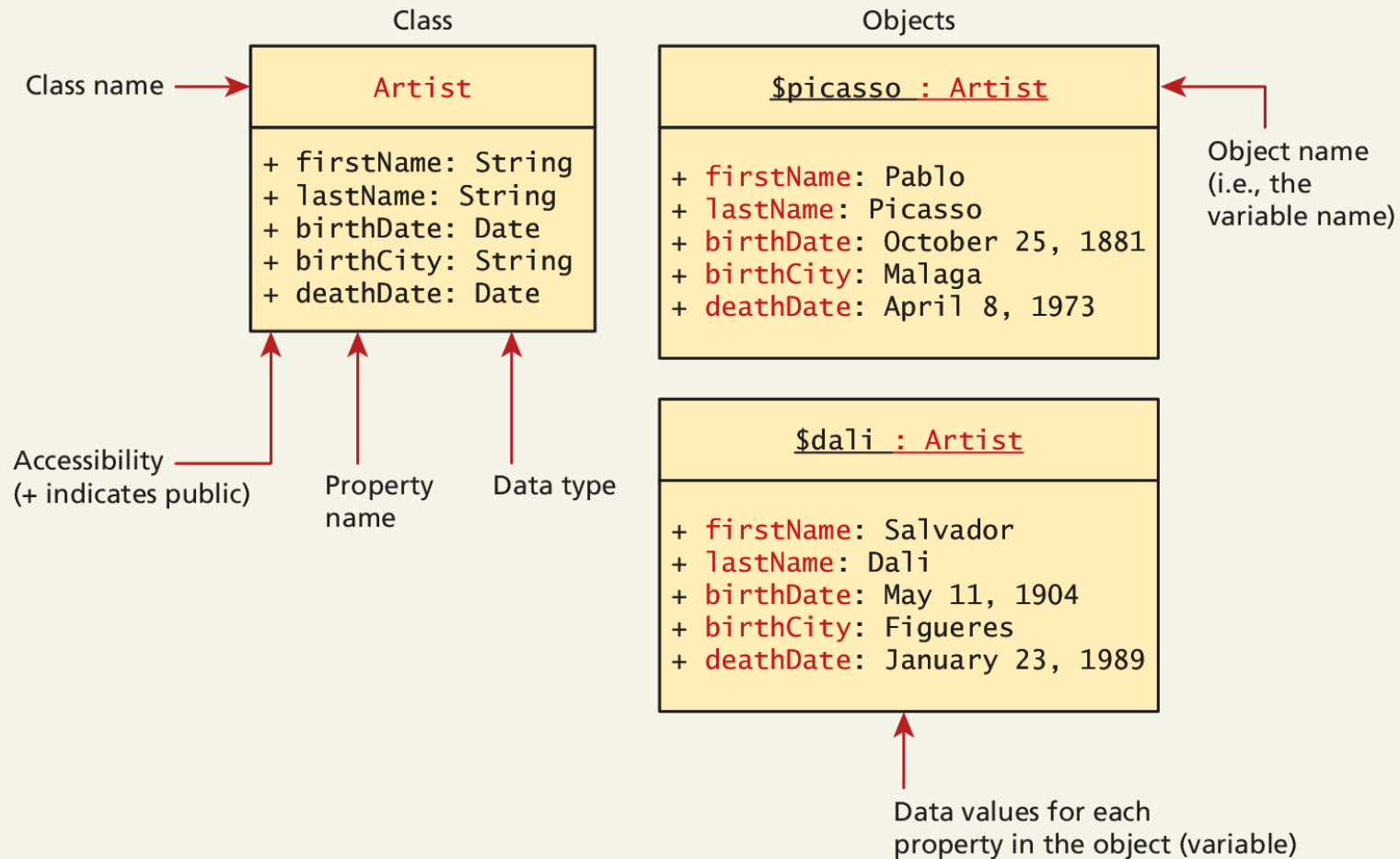
- first name,
- last name,
- birth date,
- birth city, and
- death date.

Using objects we can encapsulate those properties together into a class definition for an Artist.

UML articulates that design

# UML Class diagram

Class and a couple of objects



Section 2 of 3

# OBJECTS AND CLASSES IN PHP

# Defining Classes

In PHP

The PHP syntax for defining a class uses the **class** keyword followed by **the class name** and **{ }** braces

```
class Artist {  
    public $firstName;  
    public $lastName;  
    public $birthDate;  
    public $birthCity;  
    public $deathDate;  
}
```

**LISTING 10.1** A simple Artist class



# Instantiating Objects

In PHP

Defining a class is not the same as using it. To make use of a class, one must **instantiate** (create) objects from its definition using the *new* keyword.

```
$picasso = new Artist();
```

```
$dali = new Artist();
```

# Properties

The things in the objects

Once you have instances of an object, you can **access and modify the properties of each one** separately using the variable name and an arrow (->).

```
$picasso = new Artist();  
$dali = new Artist();  
$picasso->firstName = "Pablo";  
$picasso->lastName = "Picasso";  
$picasso->birthCity = "Malaga";  
$picasso->birthDate = "October 25 1881";  
$picasso->deathDate = "April 8 1973";
```

**LISTING 10.2** Instantiating two Artist objects and setting one of those object's properties

# Constructors

A Better way to build

**Constructors** let you specify parameters during instantiation to initialize the properties within a class right away.

In PHP, **constructors are defined as functions** (as you shall see, all methods use **the function keyword**) with the name **\_\_construct()**.

Notice that in the constructor **each parameter is assigned to an internal class variable using the \$this-> syntax**. you **must** always use the **\$this** syntax to reference all properties and methods associated with this particular instance of a class.

# Constructors

## An Example

```
class Artist {  
    // variables from previous listing still go here  
    ...  
  
    function __construct($firstName, $lastName, $city, $birth,  
                        $death=null) {  
        $this->firstName = $firstName;  
        $this->lastName = $lastName;  
        $this->birthCity = $city;  
        $this->birthDate = $birth;  
        $this->deathDate = $death;  
    }  
}
```

**LISTING 10.3** A constructor added to the class definition

# Constructors

Using the constructor

```
$picasso = new Artist("Pablo","Picasso","Malaga","Oct 25,1881","Apr 8,1973");
```

```
$dali = new Artist("Salvador","Dali","Figures","May 11 1904", "Jan 23 1989");
```

# Methods

Functions In a class

**Methods** and are like **functions**, except they are **associated with a class**.

They define the tasks each instance of a class can perform and are useful since they **associate behavior with objects**.

```
$picasso = new Artist( . . . )  
echo $picasso->outputAsTable();
```

# Methods

The example definition

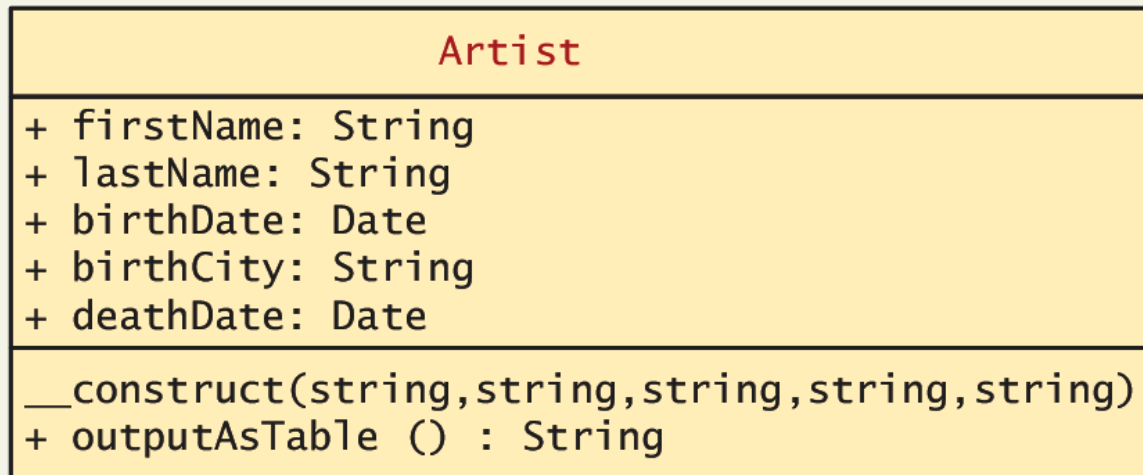
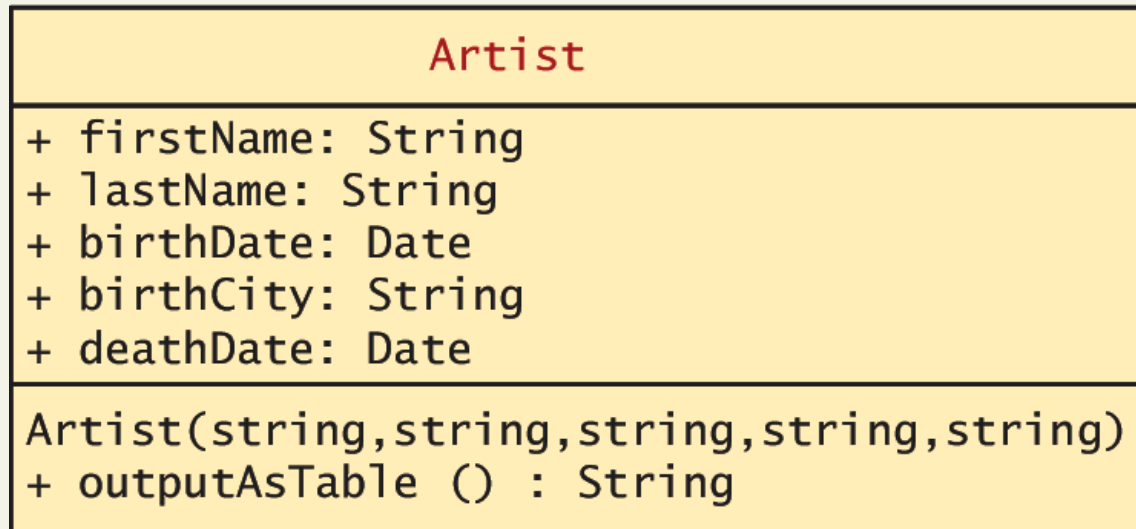
```
class Artist {  
    ...  
    public function outputAsTable() {  
        $table = "<table>";  
        $table .= "<tr><th colspan='2'>";  
        $table .= $this->firstName . " " . $this->lastName;  
        $table .= "</th></tr>";  
        $table .= "<tr><td>Birth:</td>";  
        $table .= "<td>" . $this->birthDate;  
        $table .= "(" . $this->birthCity . ")</td></tr>";  
        $table .= "<tr><td>Death:</td>";  
        $table .= "<td>" . $this->deathDate . "</td></tr>";  
        $table .= "</table>";  
        return $table;  
    }  
}
```

**LISTING 10.4** Method definition

```
$picasso = new Artist( ... )  
echo $picasso->outputAsTable();
```

# Methods

UML class diagrams adding the method





# Visibility

Or accessibility

The **visibility** of a property or method determines the accessibility of a **class member** and can be set to:

- **Public** the property or method is accessible to any code that has a reference to the object
- **Private** sets a method or variable to only be accessible from within the class
- **Protected** is related to inheritance...

# Visibility

Or accessibility

*// within some PHP page  
// or within some other class*

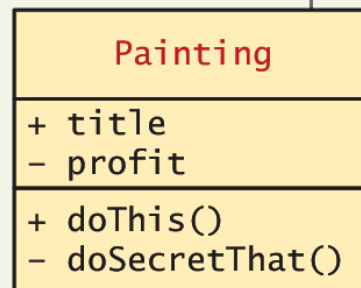
```
$p1 = new Painting();
```

```
$x = $p1->title; ✓ allowed
```

```
$y = $p1->profit; ✗ not allowed
```

```
$p1->doThis(); ✓ allowed
```

```
$p1->doSecretThat(); ✗ not allowed
```



```
class Painting {
```

```
    public $title;
```

```
    private $profit;
```

```
    public function doThis()
```

```
    {
```

```
        $a = $this->profit; ✓
```

```
        $b = $this->title; ✓
```

```
        $c = $this->doSecretThat(); ✓
```

```
        ...
```

```
    }
```

```
    private function doSecretThat()
```

```
    {
```

```
        $a = $this->profit;
```

```
        $b = $this->title;
```

```
        ...
```

```
    }
```

```
}
```

# Static Members

A **static** member is a property or method that all instances of a class share.

**Unlike an instance property**, where each object gets its own value for that property, there is only one value for a class's static property.

Static members use the **self:: syntax** and are not associated with one object

They **can be accessed without any instance** of an Artist object by using the class name, that is, via **Artist::\$artistCount**.

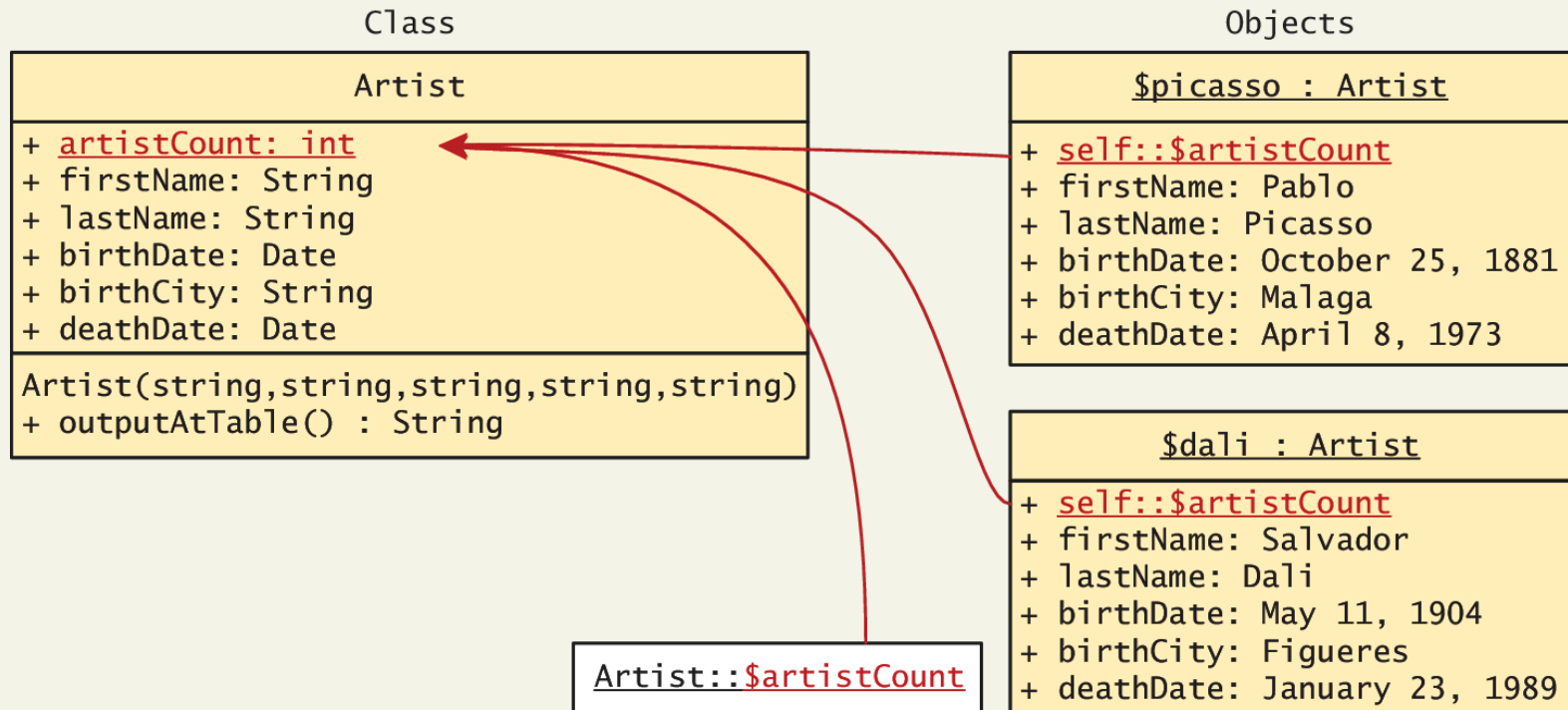
# Static Members

```
class Artist {  
    public static $artistCount = 0;  
    public $firstName;  
    public $lastName;  
    public $birthDate;  
    public $birthCity;  
    public $deathDate;  
  
    function __construct($firstName, $lastName, $city, $birth,  
                        $death=null) {  
        $this->firstName = $firstName;  
        $this->lastName = $lastName;  
        $this->birthCity = $city;  
        $this->birthDate = $birth;  
        $this->deathDate = $death;  
        self::$artistCount++;  
    }  
}
```

**LISTING 10.5** Class definition modified with static members

# Static Members

Uml again



# Class constants

Never changes

Constant values can be stored more efficiently as class constants so long as they are not calculated or updated

They are added to a class using the **const** keyword.

```
const EARLIEST_DATE = 'January 1, 1200';
```

Unlike all other variables, **constants don't use the \$ symbol when declaring or using them.**

Accessed both inside and outside the class using

- **self::EARLIEST\_DATE** in the class and
- **classReference::EARLIEST\_DATE** outside.

Section 2 of 3

# OBJECT ORIENTED DESIGN

# Data Encapsulation

What is it?

Perhaps the most important advantage to object-oriented design is the possibility of **encapsulation**, which generally refers to **restricting access to an object's internal components**.

Another way of understanding encapsulation is: it is the **hiding of an object's implementation details**

A properly encapsulated class will define an interface to the world in the form of its **public methods**, and leave its **data, that is, its properties, hidden (that is, private)**.



# Data Encapsulation

Getters and setters

If a properly encapsulated class makes its properties private, then how do you access them?

- **getters**
- **setters**

# Data Encapsulation

## Getters

A getter to return a variable's value is often very straightforward and should not modify the property.

```
public function getFirstName() {  
  
    return $this->firstName;  
  
}
```

# Data Encapsulation

## Setters

Setter methods modify properties, and allow extra logic to be added to prevent properties from being set to strange values.

```
public function setBirthDate($birthdate){  
    // set variable only if passed a valid date string  
    $date = date_create($birthdate);  
    if ( ! $date ) {  
        $this->birthDate = $this->getEarliestAllowedDate();  
    }  
    else {  
        // if very early date then change it to  
        // the earliest allowed date  
        if ( $date < $this->getEarliestAllowedDate() ) {  
            $date = $this->getEarliestAllowedDate();  
        }  
        $this->birthDate = $date;  
    }  
}
```

# Data Encapsulation

UML

Artist
<ul style="list-style-type: none"><li>- <u>artistCount</u>: int</li><li>- firstName: String</li><li>- lastName: String</li><li>- birthDate: Date</li><li>- deathDate: Date</li><li>- birthCity: String</li></ul>
Artist(string,string,string,string,string) + outputAsTable () : String  + getFirstName() : String + getLastName() : String + getBirthCity() : String + getDeathCity() : String + getBirthDate() : Date + getDeathDate() : Date + getEarliestAllowedDate() : Date + <u>getArtistCount</u> () : int  + setLastName(\$lastname) : void + setFirstName(\$firstname) : void + setBirthCity(\$birthCity) : void + setBirthDate(\$deathdate) : void + setDeathDate(\$deathdate) : void

Artist
<ul style="list-style-type: none"><li>- artistCount: Date</li><li>- firstName: String</li><li>- lastName: String</li><li>- birthDate: Date</li><li>- deathDate: Date</li><li>- birthCity: String</li></ul>
Artist(string,string,string,string,string) + outputAsTable () : String + getEarliestAllowedDate() : Date

# Data Encapsulation

## Using an encapsulated class

```
<html>
<body>
<h2>Tester for Artist class</h2>

<?php
// first must include the class definition
include 'Artist.class.php';

// now create one instance of the Artist class
$picasso = new Artist("Pablo","Picasso","Malaga","Oct 25,1881",
                      "Apr 8,1973");

// output some of its fields to test the getters
echo $picasso->getLastName() . ': ';
echo date_format($picasso->getBirthDate(),'d M Y') . ' to ';
echo date_format($picasso->getDeathDate(),'d M Y') . '<hr>';

// create another instance and test it
$dali = new Artist("Salvador","Dali","Figures","May 11,1904",
                  "January 23,1989");

echo $dali->getLastName() . ': ';
echo date_format($dali->getBirthDate(),'d M Y') . ' to ';
echo date_format($dali->getDeathDate(),'d M Y') . '<hr>';

// test the output method
echo $picasso->outputAsTable();

// finally test the static method: notice its syntax
echo '<hr>';
echo 'Number of Instantiated artists: ' . Artist::getArtistCount();

?>
</body>
</html>
```

### Example

**LISTING 10.7** Using the encapsulated class

# Inheritance

Inheritance enables you to create new **PHP classes** that **reuse, extend, and modify** the behavior that is defined in another PHP class.

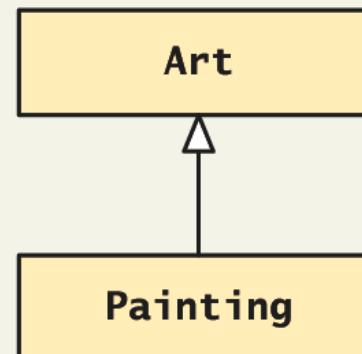
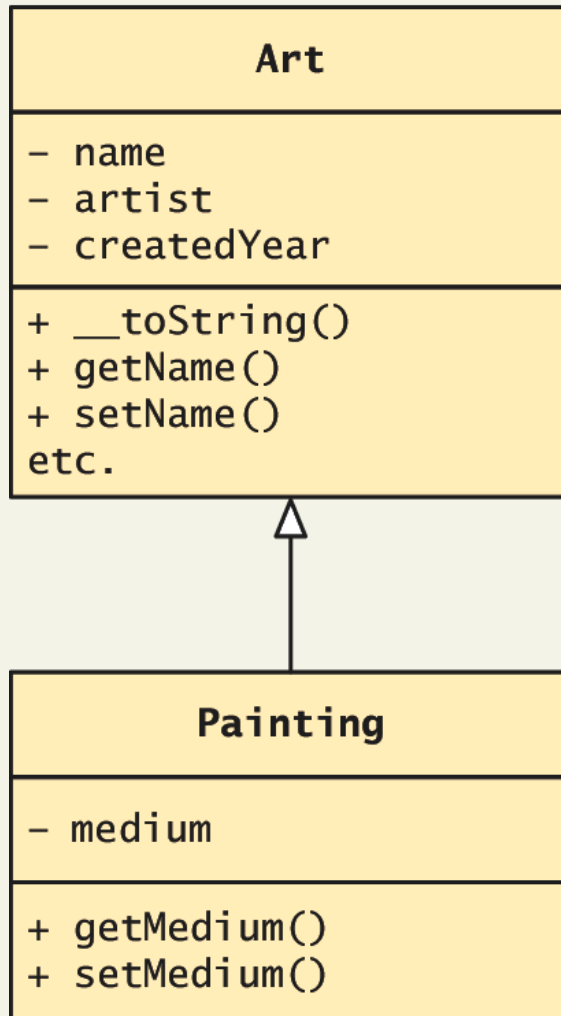
- PHP only allows you to **inherit from one class at a time**
- A class that is inheriting from another class is said to be a **subclass** or a **derived class**
- The class that is being inherited from is typically called a **superclass** or a **base class**

A PHP class is defined as a subclass by using the ***extends*** keyword.

```
class Painting extends Art { . . . }
```

# Inheritance

There's UML for that too



# Example usage

```
$p = new Painting();
```

```
...
```

```
echo $p->getName(); // defined in base class
```

```
echo $p->getMedium(); // defined in subclass
```



```

<?php
class person
{
    // explicitly adding class properties are optional - but it
    // is good practice
    var $name;
    function __construct($persons_name) {
        $this->name = $persons_name;
    }

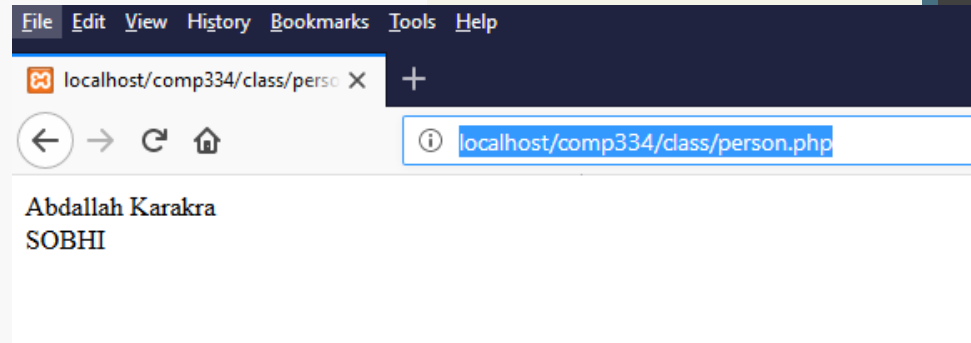
    public function get_name() {
        return $this->name;
    }

    public function set_name($new_name) {
        if ($this->name != "Abdallah") {
            $this->name = strtoupper($new_name);
        }
    }
}

// 'extends' is the keyword that enables inheritance
class employee extends person
{
    function __construct($employee_name){
        $this->set_name($employee_name);
    }
}

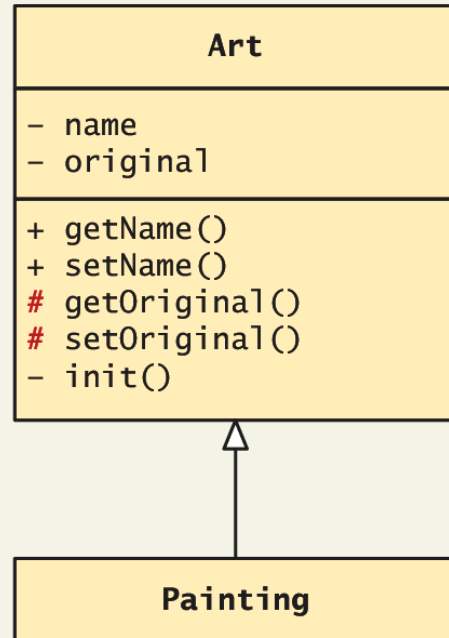
$personObj = new Person("Abdallah Karakra");
$employeeObj= new employee("Sobhi");
echo $personObj->get_name()."<br>";
echo $employeeObj->get_name();

```



# Protected access modifier

Remember Protected?



```
class Painting extends Art {
    ...
    private function foo() {
        ...
        // these are allowed
        ✓ $w = parent::getName();
        ✓ $x = parent::getOriginal();

        // this is not allowed
        ✗ $y = parent::init();
    }
}
```

```
// in some page or other class
$p = new Painting();
$a = new Art();
```

```
// neither of these references are allowed
✗ $w = $p->getOriginal();
✗ $y = $a->getOriginal();
```

# Interface VS Abstract Class

Interface	Abstract class
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does'n Contains Data Member	Abstract class contains Data Member
Interface does'n contains Cunstructors	Abstract class contains Cunstructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

# Extended example

All art has certain properties

*/\* The abstract class that  
contains functionality required by  
all types of Art \*/*

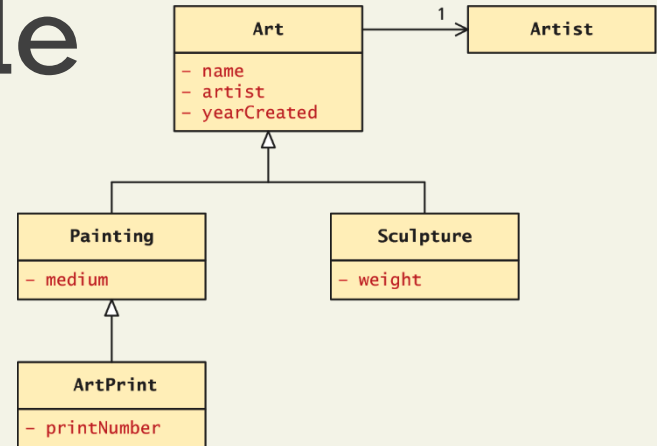
**abstract class Art {**

private \$name;

private \$artist;

private \$yearCreated;

*//... constructor, getters, setters*



# Extended example

Painting require a “medium”

```
class Painting extends Art {
```

```
    private $medium;
```

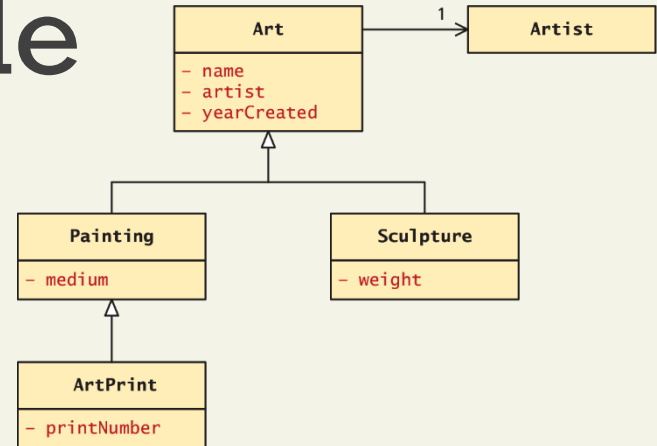
```
    //...constructor, getters, setters
```

```
    public function __toString() {
```

```
        return parent::__toString() . ", Medium: " .  
            $this->getMedium();
```

```
    }
```

```
}
```



# Extended example

Sculptures have weight

```
class Sculpture extends Art {
```

```
    private $weight;
```

```
    //...constructor, getters, setters
```

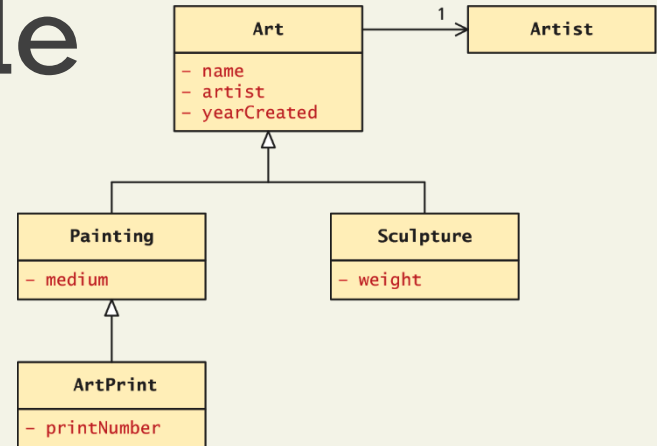
```
    public function __toString() {
```

```
        return parent::__toString() . ", Weight: " .
```

```
            $this->getWeight() ."kg";
```

```
    }
```

```
}
```



# Extended example

Using the classes

...

```
$picasso = new Artist("Pablo","Picasso","Malaga","May 11,1904","Apr 8, 1973");
```

```
$guernica = new Painting("1937",$picasso,"Guernica", "Oil on canvas");
```

```
$woman = new Sculpture("1909",$picasso,"Head of a Woman", 30.5);
```

```
?>
```

```
<h2>Paintings</h2>
```

```
<p><em>Use the __toString() methods </em></p>
```

```
<p><?php echo $guernica; ?></p>
```

```
<h2>Sculptures</h2>
```

```
<p> <?php echo $woman; ?></p>
```

# Polymorphism

No thank you, I'll have water

**Polymorphism** is the notion that an **object can in fact be multiple things at the same time.**

Consider an instance of a Painting object named \$guernica created as follows:

```
$guernica = new Painting("1937",$picasso,"Guernica","Oil on canvas");
```

The **variable \$guernica is both a *Painting* object and an *Art* object** due to its inheritance.

The advantage of polymorphism is that **we can manage a list of Art objects**, and call the same overridden method on each.



# Polymorphism

```
$picasso = new Artist("Pablo","Picasso","Malaga","Oct 25, 1881",
    "Apr 8, 1973");

// create the paintings
$guernica = new Painting("1937",$picasso,"Guernica","Oil on canvas");
$chicago = new Sculpture("1967",$picasso,"Chicago", 454);

// create an array of art
$works = array();
$works[0] = $guernica;
$works[1] = $chicago;
// to test polymorphism, loop through art array
foreach ($works as $art)
{
    // the beauty of polymorphism:
    // the appropriate __toString() method will be called!
    echo $art;
}

// add works to artist ... any type of art class will work
$picasso->addWork($guernica);
$picasso->addWork($chicago);
// do the same type of loop
foreach ($picasso->getWorks() as $art) {
    echo $art; // again polymorphism at work
}
```

**LISTING 10.10** Using polymorphism

# Interfaces

Defining the interface

An object **interface** is a way of defining a formal list of methods that **a class must implement without specifying their implementation.**

Interfaces are defined using the interface keyword, and look similar to standard PHP classes, except an interface contains no properties and its methods do not have method bodies defined.

```
interface Viewable {  
  
    public function getSize();  
  
    public function getPNG();  
  
}
```

# Interfaces

## Implementing the Interface

In PHP, a class can be said to *implement* an interface, using the implements keyword:

```
class Painting extends Art implements Viewable { ... }
```

This means then that **the class *Painting* must provide implementations for the `getSize()` and `getPNG()` methods.**

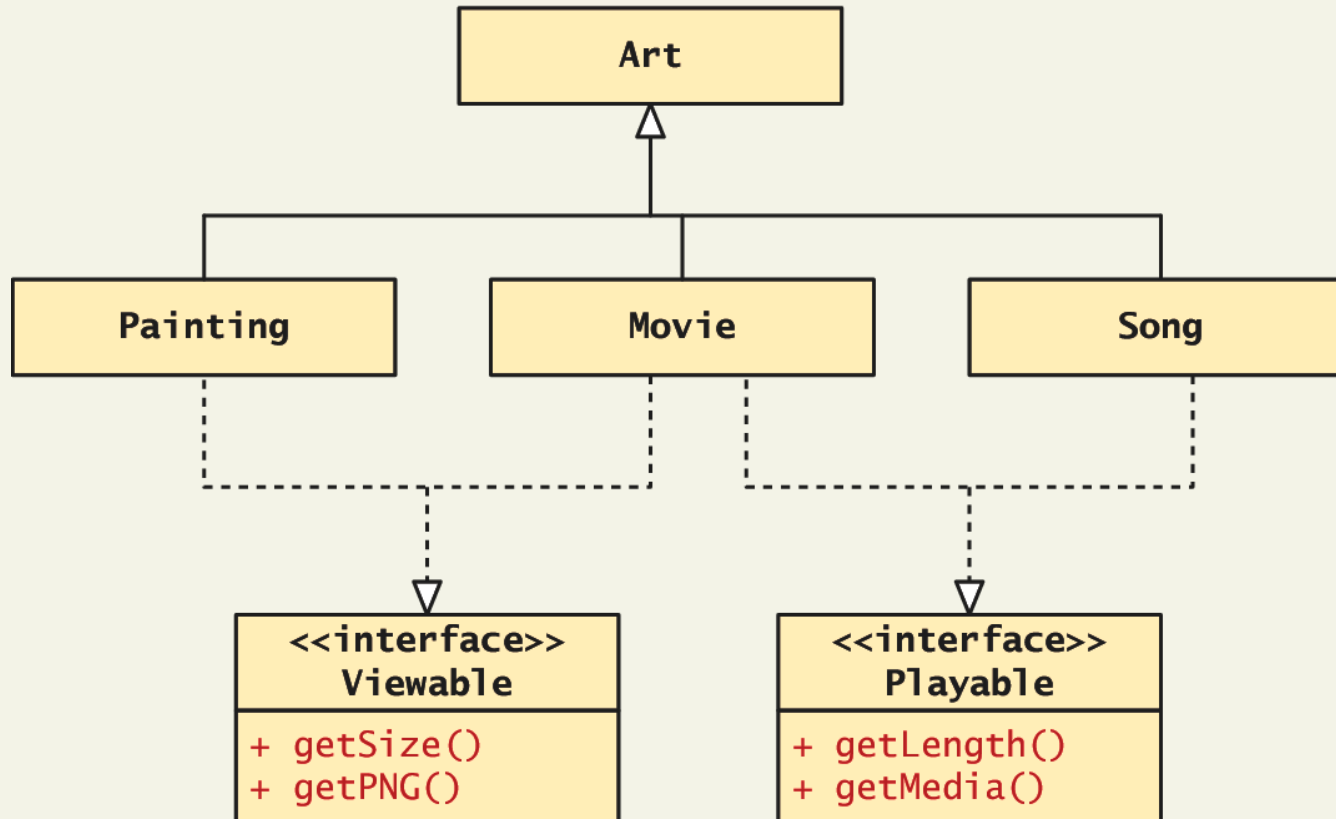
# Interface Example

```
interface Viewable {  
    public function getSize();  
    public function getPNG();  
}  
  
class Painting extends Art implements Viewable {  
    ...  
    public function getPNG() {  
        //return image data would go here  
        ...  
    }  
    public function getSize() {  
        //return image size would go here  
        ...  
    }  
}
```

**LISTING 10.11** Painting class implementing an interface

# Interfaces

An Extended example



# What You've Learned

**1**

Object-Oriented  
Overview

**2**

Classes and Objects in  
PHP

**3**

Object Oriented Design