CHAPTER 12

# Static

The `static` keyword can be used to declare properties and methods that can be accessed without having to create an instance of the class. Static (class) members only exist in one copy, which belongs to the class itself, whereas instance (non-static) members are created as new copies for each new object.

```
class MyCircle
{
  // Instance members (one per object)
  public $r = 10;
  function getArea() {}

  // Static/class members (only one copy)
  static $pi = 3.14;
  static function newArea($a) {}
}
```

Static methods cannot use instance members since these methods are not part of an instance. They can use other static members, however.

## Referencing Static Members

Unlike instance members, static members are not accessed using the single arrow operator (`->`). Instead, to reference static members inside a class, the member must be prefixed with the `self` keyword followed by the scope resolution operator (`::`). The `self` keyword is an alias for the class name, so alternatively, the actual name of the class can be used.

```
static function newArea($a)
{
  return self::$pi * $a * $a;     // ok
  return MyCircle::$pi * $a * $a; // alternative
}
```


M. Olsson, *PHP 7 Quick Scripting Reference*, DOI 10.1007/978-1-4842-1922-5_12

This same syntax is used to access static members from an instance method. Note that in contrast to static methods, instance methods can use both static and instance members.

```
function getArea()
{
  return self::newArea($this->$r);
}
```

To access static members from outside the class, the name of the class needs to be used, followed by the scope resolution operator (`::`).

```
class MyCircle
{
  static $pi = 3.14;

  static function newArea($a)
  {
    return self::$pi * $a * $a;
  }
}

echo MyCircle::$pi; // "3.14"
echo MyCircle::newArea(10); // "314"
```

The advantage of static members can be seen here; they can be used without having to create an instance of the class. Therefore, methods should be declared static if they perform a generic function independently of instance variables. Likewise, properties should be declared static if there is only need for a single instance of the variable.

# Static Variables

Local variables can be declared static to make the function remember its value. Such a static variable only exists in the local function's scope, but it does not lose its value when the function ends. This can be used to count the number of times a function is called, for example.

```
function add()
{
  static $val = 0;
  echo $val++;
}

add(); // "0"
add(); // "1"
add(); // "2"
```

The initial value that a static variable is given is only set once. Keep in mind that static properties and static variables may only be initialized with a constant; but not with an expression, such as another variable or a function return value.

# Late Static Bindings

As mentioned before, the `self` keyword is an alias for the class name of the enclosing class. This means that the keyword refers to its enclosing class even when it is called from the context of a child class.

```
class MyParent
{
  protected static $val = 'parent';

  public static function getVal()
  {
    return self::$val;
  }
}

class MyChild extends MyParent
{
  protected static $val = 'child';
}

echo MyChild::getVal(); // "parent"
```

To get the class reference to evaluate to the actual calling class, the `static` keyword needs to be used instead of the `self` keyword. This feature is called *late static bindings* and it was added in PHP 5.3.

```
class MyParent
{
  protected static $val = 'parent';

  public static function getLateBindingVal()
  {
    return static::$val;
  }
}

class MyChild extends MyParent
{
  protected static $val = 'child';
}
echo MyChild::getLateBindingVal(); // "child"
```