

The Grader: Student Guide

Manoj Thulasidas

Abstract—Grader is designed to make the logistical part of grading the WAD lab tests as seamless as possible. It can validate the files uploaded by the students, display the files, help grade with rubrics displayed, attach comments to graded files and much more. Since you are a student, you will not actually be grading anybody. You can, however, see the Grader in action using the included sample files. You can also look at the Quick Start Guide (Grader-QuickStart.pdf) to see how an instructor would use it. For much more details on the design and usage of the Grader, please refer to the conference paper (Grader.pdf), presented at EDUCON 2024 in Greece in May, 2024. Keep in mind, however, that what is described in the paper is an earlier version.

I. QUICK START

- 1) Unzip Grader.zip to your Document Root DR.
- 2) Open localhost/Grader/index.php in your browser and follow the online documentation.

II. LEARNING POINTS

Most of the ideas in the Grader package are from what we learned in IS113. However, it does use a couple of object-oriented features that we have not covered.

- **Inheritance:** In OOP, we can have a Base class and a Derived class. The (public and protected) properties and methods in the Base class also available to the Derived class. Inheritance helps us avoid repeating code.
- **Access Modifier protected:** The Base class methods or properties with visibility protected are accessible to the Derived class, while the private ones are not. In this sense, protected is “in between” the private and public visibilities. Read more about it here: <https://stackoverflow.com/q/8353272>
- **Declaration static:** Properties or methods that are declared static exist only on a per-class basis (as opposed to per-object). They are similar to the class constants that we learned about in the course. We access them using `self::` (or `static::`), but not `$this->`. Since they are available without instantiating the object, we cannot use `$this` in static methods. Remember that `self` (or `static`) stands for the class, while `$this` stands for an object instantiated from the class. More info here: <https://www.php.net/manual/en/language.oop5.static.php>
- **Late Static Binding:** This new feature of PHP is a bit harder to explain. Here is some info: <https://stackoverflow.com/q/1912902>. I (or rather, ChatGPT) will explain it further as a section below. As you will see, Grader uses this feature quite extensively to perform its magic when it comes to creating the landing HTML pages.

- **self:: vs static:::** The distinction between these two keywords becomes significant when using late static binding, as explained below. <https://stackoverflow.com/a/64072873>
- **HTML Stuff:** Grader also uses iframes (with dynamically generated source), some JavaScript for automation and some styling. It also uses syntax highlighting from a publicly available package.
- **UML Diagram:** Study the UML diagram in Fig. 1 (which is also included as Grader-UML.png) to understand the various classes and how their dependence on one another.

III. DETAILED EXPLANATIONS (FROM CHATGPT)

A. Understanding Inheritance in PHP

Inheritance in PHP is a fundamental concept of object-oriented programming that allows classes to inherit properties and methods from other classes. This feature facilitates code reuse and the organization of code into hierarchical structures.

1) *Defining Classes and Inheritance:* To define a class in PHP, you use the `class` keyword. Inheritance is implemented using the `extends` keyword, indicating that one class (the child or subclass) will inherit from another class (the parent or superclass).

```
1 <?php
2 class Vehicle {
3     public $brand;
4
5     public function setBrand($brand) {
6         $this->brand = $brand;
7     }
8
9     public function getBrand() {
10        return $this->brand;
11    }
12 }
13
14 class Car extends Vehicle {
15     private $model;
16
17     public function setModel($model) {
18         $this->model = $model;
19     }
20
21     public function getModel() {
22         return $this->model;
23     }
24 }
```

2) *Access Modifiers:* PHP supports three access modifiers:

- **Public:** Accessible from anywhere.

- **Protected:** Accessible within the class itself, by classes that extend it, and parent classes.
- **Private:** Accessible only within the class it is defined.

3) *Overriding Methods:* Subclasses in PHP can override methods of their superclasses. This allows the subclass to provide its own implementation of a method that is already defined in its superclass.

```

1 <?php
2 class Vehicle {
3     public function startEngine() {
4         echo "Engine started";
5     }
6 }
7
8 class ElectricCar extends Vehicle {
9     public function startEngine() {
10        echo "Electric engine started silently";
11    }
12 }

```

4) *The parent Keyword:* The parent keyword is used to call methods or access properties of the parent class.

```

1 <?php
2 class Vehicle {
3     public function startEngine() {
4         echo "Engine started";
5     }
6 }
7
8 class ElectricCar extends Vehicle {
9     public function startEngine() {
10        parent::startEngine(); // Calls the
11        parent method
12        echo " with an electric boost";
13    }
14 }

```

Inheritance in PHP allows for the creation of a class hierarchy that enhances code reuse and organization. By leveraging inheritance along with access modifiers, method overriding, and the parent keyword, developers can write more modular, maintainable, and flexible PHP code.

B. Understanding Late Static Binding in PHP

Late static binding in PHP is a feature that allows the resolution of static method or property calls to the class that was last non-statically called in the hierarchy. This is particularly useful in object-oriented programming when dealing with inheritance and static method overrides, enabling more dynamic behavior.

Late static binding allows for referencing the called class in a context of static inheritance. Typically, when a static method in PHP is called using the `self` keyword, it refers to the class in which the method is defined. However, with late static binding, using the `static` keyword instead of `self` will resolve to the class that originally made the call, even within a parent class.

This feature addresses the limitations of early binding, which binds static method calls to the class where a method is defined at compile time. Late static binding defers this resolution to runtime, thus allowing a static method to know which class it was called on.

Here's an example illustrating late static binding:

```

1 <?php
2 class A {
3     public static function who() {
4         echo __CLASS__;
5     }
6
7     public static function test() {
8         self::who(); // Early Binding
9         static::who(); // Late Static Binding
10    }
11 }
12
13 class B extends A {
14     public static function who() {
15         echo __CLASS__;
16     }
17 }
18
19 B::test(); // Outputs: AB

```

In this example, calling `B::test()` outputs 'AB', demonstrating the difference between early binding (`self::who()`) and late static binding (`static::who()`). Late static binding correctly identifies the call as being made from class B, despite the method being called within the parent class A.

C. Understanding Code through UML Class Diagrams

UML class diagrams are a fundamental tool in object-oriented programming for visualizing the structure of a system's classes, their attributes, methods, and the relationships between classes. Here is how class diagrams can be used to enhance the understanding of code.

1) Visualization of Class Hierarchy and Relationships:

Class diagrams provide a clear visual representation of the class hierarchy, illustrating inheritance between classes, which is invaluable for understanding the object-oriented design of a system.

- **Inheritance:** Class diagrams show inheritance relationships, making it easier to see how classes are derived from one another and how they share or override functionality.
- **Association:** These diagrams also detail how classes interact with each other through associations, including aggregation and composition relationships, providing insight into the system's architecture.

2) *Attributes and Methods:* By visualizing class attributes and methods, class diagrams offer a comprehensive overview of what each class does, what information it contains, and how it behaves.

- **Attributes:** Seeing the data each class holds can help developers understand the role of each class within the system.

