Unit 3 : Object Oriented Concepts: Definition, Creation, Declaration, Accessing of Class & Object, Object properties, Object methods, **Constructor:** Definition, Types, Destructor, **Polymorphism:** Method Overloading, Property Overloading. Access Specifiers **Inheritance:** Definition, Single Inheritance, Multilevel Inheritance, Hierarchical Inheritance, Interfaces, Abstract Class, Overriding. **Exception Handling:** try, catch, multi try, multi catch, throw, finally.

What is OOP?

OOP stands for Object-Oriented Programming

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the PHP code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Before we go in detail, lets define important terms related to Object Oriented Programming here

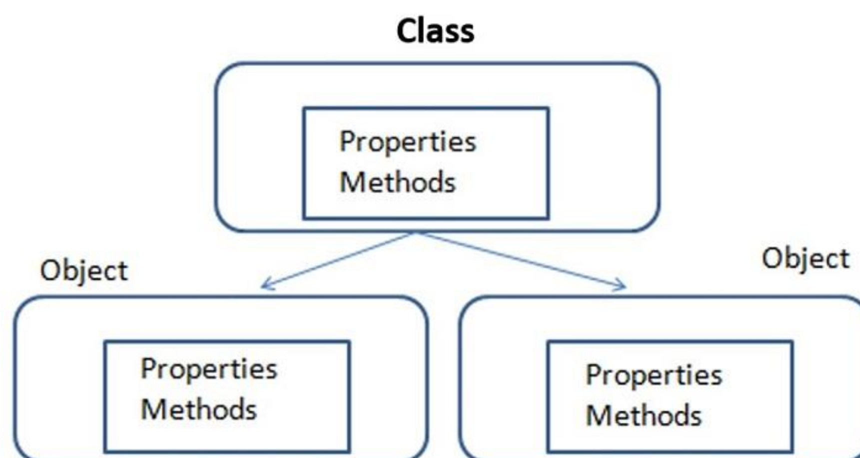
- **Class** − This is a programmer-defined data type, which includes local functions as well as local data. You can think of a class as a template for making many instances of the same kind (or class) of object.
- **Object** − An individual instance of the data structure defined by a class. You define a class once and then make many objects that belong to it. Objects are also known as instance.
- **Member Variable** − These are the variables defined inside a class. This data will be invisible to the outside of the class and can be accessed via member functions. These variables are called attribute of the object once an object is created.
- **Member function** − These are the function defined inside a class and are used to access object data.
- **Inheritance** − When a class is defined by inheriting existing function of a parent class then it is called inheritance. Here child class will inherit all or few member functions and variables of a parent class.
- **Parent class** − A class that is inherited from by another class. This is also called a base class or super class.
- **Child Class** − A class that inherits from another class. This is also called a subclass or derived class.
- **Polymorphism** − This is an object oriented concept where same function can be used for

different purposes. For example function name will remain same but it take different number of arguments and can do different task.

- **Overloading** − a type of polymorphism in which some or all of operators have different implementations depending on the types of their arguments. Similarly functions can also be overloaded with different implementation.
- **Data Abstraction** − Any representation of data in which the implementation details are hidden (abstracted).
- **Encapsulation** − refers to a concept where we encapsulate all the data and member functions together to form an object.
- **Constructor** − refers to a special type of function which will be called automatically whenever there is an object formation from a class.
- **Destructor** − refers to a special type of function which will be called automatically whenever an object is deleted or goes out of scope.

PHP - Classes and Objects

The concept of classes and objects is central to PHP's object-oriented programming methodology. A **class** is the template description of its objects. It includes the properties and functions that process the properties. An **object** is the instance of its class. It is characterized by the properties and functions defined in the class.



Defining a Class in PHP

To define a class, PHP has a keyword "**class**". Similarly, PHP provides the keyword "**new**" to declare an object of any given class.

A class is defined by using the class keyword, followed by the name of the class and a pair of curly braces ({}). All its properties and methods go inside the braces:

Syntax

```
<?php
class Fruit {
// code goes here
}
```

```
?>
```

The general form for defining a new class in PHP is as follows –

```
<?php
   class phpClass {
      var $var1;
      var $var2 = "constant string";

      function myfunc ($arg1, $arg2) {
         [..]
      }
      [..]
   }
?>
```

The keyword **class** is followed by the name of the class that you want to define. Class name follows the same naming conventions as used for a PHP variable. It is followed by a pair of braces enclosing any number of variable declarations (properties) and function definitions.

Variable declarations start with another reserved keyword **var**, which is followed by a conventional **$variable** name; they may also have an initial assignment to a constant value.

Function definitions look much like standalone PHP functions but are local to the class and will be used to set and access object data. Functions inside a class are also called methods.

```
class Book {

  /* Member variables */
  var $price;
  var $title;

  /* Member functions */
  function setPrice($par){
    $this->price = $par;
  }

  function getPrice(){
    echo $this->price ."<br/>";
  }
```

```
    function setTitle($par){
      $this->title = $par;
    }

    function getTitle(){
      echo $this->title ." <br/>";
    }
  }
```

The pseudo-variable **$this** is available when a method is called from within an object context. **$this** refers to the calling object.

The Book class has two **member variables** (or properties) - **$title** and **$price**. The member variables (also sometimes called instance variables) usually have different values for each object; like each book has a title and price different from the other.

The Book class has functions (functions defined inside the class are called **methods**) setTitle() and setPrice(). These functions are called with reference to an object and a parameter, used to set the value of title and price member variables respectively.

The Book class also has **getTitle()** and **getPrice()** methods. When called, they return the title and price of the object whose reference is passed.

Defining an Object in PHP

Classes are nothing without objects! We can create multiple objects from a class. Each object has all the properties and methods defined in the class, but they will have different property values.

Objects of a class is created using the new keyword.

An object is an instance of a class. When you create an object, you follow the blueprint provided by the class. Each object can have different attributes.
Once a class is defined, you can declare one or more objects, using new operator.
$b1 = new Book;
$b2 = new Book;

The **new** operator allocates the memory required for the member variables and methods of each object. Here we have created two objects and these objects are independent of each other and they will have their existence separately.

Each object has access to its member variables and methods with the "**->**" operator. For example,

the **$title** property of **b1** object is "**$b1->title**" and to call setTitle() method, use the "**$b1->setTitle()**" statement.

To set the title and price of **b1** object,
$b1->setTitle("PHP Programming");
$b1->setPrice(450);

Similarly, the following statements fetch the title and price of **b1** book –
echo $b1->getPrice();
echo $b1->getTitle();

Example :

```
<?php
  class Book {

    /* Member variables */
    var $price;
    var $title;

    /* Member functions */
    function setPrice($par){
      $this->price = $par;
    }

    function getPrice(){
      echo $this->price ."\n";
    }

    function setTitle($par){
      $this->title = $par;
    }

    function getTitle(){
      echo $this->title ."\n";
    }
  }

  $b1 = new Book;
  $b2 =new Book;

  $b1->setTitle("PHP Programming");
```

```
  $b1->setPrice(450);
  $b2->setTitle("PHP Fundamentals");
  $b2->setPrice(275);
  $b1->getTitle();
  $b1->getPrice();
  $b2->getTitle();
  $b2->getPrice();
?>
```

Output :

It will produce the following **output** –

PHP Programming

450

PHP Fundamentals

275

Member Functions:

After creating our objects, we can call member functions related to that object. A member function typically accesses members of current object only.

Example:

```
$physics->setTitle( "Physics for High School" );
$chemistry->setTitle( "Advanced Chemistry" );
$maths->setTitle( "Algebra" );
$physics->setPrice( 10 );
$chemistry->setPrice( 15 );
$maths->setPrice( 7 );
```

Example:

```
<?php
class Books {
/* Member variables */
var $price;
var $title;
/* Member functions */
function setPrice($par){
$this->price = $par;
}
function getPrice(){
echo $this->price."<br>";
}
```

```
function setTitle($par){
$this->title = $par;
}
function getTitle(){
echo $this->title."<br>" ;
}
}
/* Creating New object using "new" operator */
$maths = new Books;
/* Setting title and prices for the object */
$maths->setTitle( "Algebra" );
$maths->setPrice( 7 );
/* Calling Member Functions */
$maths->getTitle();
$maths->getPrice();
?>
```

Defining Class Properties

To add data to a class, properties, or class-specific variables, are used. These work exactly like regular
variables, except they're bound to the object and therefore can only be accessed using the object.

```
<?php
class MyClass
{
public $prop1 = "I'm a class property!";
}
$obj = new MyClass;
echo $obj->prop1; // Output the property
?>
```

Output: I'm a class property!

Defining Class Methods

Methods are class-specific functions. Individual actions that an object will be able to perform are defined within the class as methods.

```
<?php
class MyClass
{
public $prop1 = "I'm a class property!";
public function setProperty($newval)
{
$this->prop1 = $newval;
```

```
}
public function getProperty()
{
return $this->prop1 . "<br />";
}
}
$obj = new MyClass;
echo $obj->getProperty(); // Get the property value
$obj->setProperty("I'm a new property value!"); // Set a new one
echo $obj->getProperty(); // Read it out again to show the change
?>
```

Output: I'm a class property!
I'm a new property value!

Example:

```
<?php
class MyClass
{
public $prop1 = "I'm a class property!";
public function setProperty($newval)
{
$this->prop1 = $newval;
}
public function getProperty()
{
return $this->prop1 . "<br />";
}
}
// Create two objects
$obj = new MyClass;
$obj2 = new MyClass;
// Get the value of $prop1 from both objects
echo $obj->getProperty();
echo $obj2->getProperty();
// Set new values for both objects
$obj->setProperty("I'm a new property value!");
$obj2->setProperty("I belong to the second instance!");
// Output both objects' $prop1 value
echo $obj->getProperty();
echo $obj2->getProperty();
?>
```

Output: I'm a class property!
I'm a class property!
I'm a new property value!
I belong to the second instance!

Why Use Classes and Objects?

Using classes and objects can help you –

- **Organize Code:** Place related properties and methods together.
- **Reuse Code:** Create several objects from the same class without changing the code.
- **Encapsulation:** It is the process of keeping properties and methods safe from outside interference.

PHP - The $this Keyword

The $this keyword refers to the current object, and is only available inside methods.

Look at the following example:

```
<?php
class Fruit {
 public $name;
}
$apple = new Fruit();
?>
```

So, where can we change the value of the $name property? There are two ways:

1. Inside the class (by adding a set_name() method and use $this):
```
<?php
class Fruit {
 public $name;
 function set_name($name) {
 $this->name = $name;
 }
}
$apple = new Fruit();
$apple->set_name("Apple");
?>
```
2. Outside the class (by directly changing the property value):
```
<?php
class Fruit {
 public $name;
```

```
}
$apple = new Fruit();
$apple->name = "Apple";
?>
```

PHP - instanceof

You can use the instanceof keyword to check if an object belongs to a specific class:

```
<?php
$apple = new Fruit();
var_dump($apple instanceof Fruit);
?>
```

The Access Specifier

There are three access specifier in PHP

- Public
- Protected
- Private

Public: Public properties can be accessed by any code, whether that code is inside or outside the class. If a property is declared public, its value can be read or changed from anywhere in your script.

Private: Private properties of a class can be accessed only by code inside the class. So if we create a property that's declared private, only methods and objects inside the same class can access its contents.

Protected: Protected class properties are a bit like private properties in that they can't be accessed by code outside the class, but there's one little difference in any class that inherits from the class i.e. base class can also access the properties.

Class Member Access Specifier	Access from own class	Accessible from derived class	Accessible by Object
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

Example 1: In this example, we will see the Public Access Specifier.

```
<?php
class Arithmetic
{
public $x = 100 ; # public attributes
public $y = 50 ;
```

```
function add()
{
echo $a = $this->x + $this->y ;
echo " ";
}
}
class child extends Arithmetic
{
function sub()
{
echo $s = $this->x - $this->y ;
}
}
$obj = new child;
$obj->add() ; // It will return the addition result
$obj->sub() ; /* It's a derived class of the main class, which has a public object and therefore
can be accessed, returning the subtracted result. */
?>
```

Example 2: In this example, we will see the Private Access Specifier.

```
<?php
class Arithmetic
{
private $a = 75 ; # private attributes
private $b = 5 ;
private function div() # private member function
{
echo $d = $this->a / $this->b ;
echo " ";
}
}
class child extends Arithmetic
{
function mul()
{
echo $m = $this->a * $this->b ;
}
}
$obj= new child; /* It's supposed to return the division result but since the data and
function are private they can't be accessed by a derived class which will lead to fatal error.*/
$obj->div(); /* It's a derived class of the main class, which's accessing the private data which
```

```
again will lead to fatal error.*/
$obj->mul();
?>
```

Example 3: In this example, we will see the Protected Access Specifier.

```
<?php
class Arithmetic
{
protected $x = 1000 ; # protected attributes
protected $y = 100 ;
function div()
{
echo $d = $this->x / $this->y ;
echo " ";
}
}
class child extends Arithmetic
{
function sub()
{
echo $s = $this->x - $this->y ;
}
}
class derived # Outside Class
{
function mul()
{
echo $m = $this->x * $this->y ;
}
}
$obj= new child; // It will return the division result
$obj->div(); // Since it's a derived class of the main class,
$obj->sub(); // Since it's an outside class, therefore it will produce a fatal error.
$obj->mul();
?>
```

Object properties and methods

We call properties to the variables inside a class. Properties can accept values like strings, integers, and boolean (true/false values), like any other variable. Let's add some properties to the Car class.

Example:

```
Class Car
```

```
{
public $comp;
public $color = 'beige';
public $hasSunRoof = true;
}
```

- We put the public keyword in front of a class property.
- The naming convention is to start the property name with a lower case letter.
- If the name contains more than one word, all of the words, except for the first word, start with an upper case letter. For example, $color or $hasSunRoof.
- A property can have a default value. For example, $color = 'beige'.

- We can also create a property without a default value. See the property $comp in the above example.

Constructor and Destructor

Constructor

- A constructor allows you to initialize an object's properties upon creation of the object.
- If you create a construct() function, PHP will automatically call this function when you create an object from a class.
- Notice that the construct function starts with two underscores (__)

Following example will create one constructor for Books class and it will initialize price and title for the book at the time of object creation.

```
function __construct( $par1, $par2 )
{
$this->title = $par1;
$this->price = $par2;
}
```

Now we don't need to call set function separately to set price and title. We can initialize these two member variables at the time of object creation only. Check following example below –

```
$physics = new Books( "Physics for High School", 10 );
$maths = new Books ( "Advanced Chemistry", 15 );
$chemistry = new Books ("Algebra", 7 );
```

Example:

```
<?Php
class Example
{
public function __construct()
{
echo "Hello javatpoint";
```

```
}
}
$obj = new Example();
$obj = new Example();
?>
```

Example

```
<?php
class BankAccount
{
private $accountNumber;
private $balance;
function __construct($accountNumber, $balance)
{
$this->accountNumber = $accountNumber;
$this->balance = $balance;
}
}
```

Uses of constructor in php

- In PHP, a constructor is a special method within a class that is automatically called when an object of that class is instantiated (i.e., when a new instance of the class is created using the new keyword).
- Constructors are useful for initializing object properties or performing any setup tasks that are required when an object is created.
- **Initialization**: Constructors are used to initialize object properties with default values or values passed as arguments during object creation. This ensures that objects are in a valid state when they are created.
- **Dependency Injection**: Constructors can be used for dependency injection, where objects are passed as arguments to the constructor. This allows for greater flexibility and decoupling of classes, making the code more maintainable and testable.
- **Resource Allocation**: Constructors are often used to allocate resources such as opening files, establishing database connections, or initializing other objects needed by the class. This ensures that resources are properly initialized and available for use when needed.
- **Configuration**: Constructors can be used to configure objects with initial settings or parameters. This can include setting up default values, loading configuration from external sources, or performing any necessary setup tasks before the object is used.

Types of Constructor

Default Constructor: It has no parameters, but the values to the default constructor can be passed dynamically.

Parameterized Constructor: It takes the parameters, and also you can pass different values to the

data members.

Copy Constructor: It accepts the address of the other objects as a parameter.

Default Constructor

A default constructor is automatically provided by PHP if a class does not explicitly define a constructor. It doesn't take any parameters and doesn't perform any specific initialization tasks. When an object of the class is created, this default constructor is called by default.

Example:

```
<?php
class Example
{
//defining constructor
function __construct( )
{
$this->sayHello();
echo "This is Constructor in Example Class";
}
function sayHello( )
{
echo "Hello<br>";
}
}
$obj= new Example();
?>
```

Parameterized Constructor

A parameterized constructor is explicitly defined within a class and accepts parameters to customize the initialization process. It allows you to pass values during object creation and use those values to set properties or perform other tasks. This type of constructor is useful when objects require specific information to function correctly.

Example:

```
<?php
class Example
{
//defining constructor
function __construct($name)
{
echo "Hello ".$name."<br>";
}
}
$obj1= new Example("World");
```

```
$obj2= new Example("Owlbuddy");
?>
```

Copy Constructor

The copy constructor is used to create a duplicate of an already existing object. It accepts the address of another object as a parameter, allowing for the creation of an identical copy.

Example:

```
<?php
class Example
{
public $name;
public $age;
//default constructor
public function __construct( )
{
}
// This is a copy constructor
public function copyCon(Example $o)
{
$this->name = $o->name;
$this->age = $o->age;
}
function show( )
{
echo "Name = ".$this->name.'<br>';
echo "Age = ".$this->age.'<br>';
}
}
$obj1 = new Example();
$obj1->name = 'Aman';
$obj1->age = '21';
$obj1->show();
$obj2 = new Example();
//calling copyCon method
$obj2->copyCon($obj1);
$obj2->show();
?>
```

Destructor

Like a constructor function you can define a destructor function using function __destruct( ). You can release all the resources with-in a destructor.

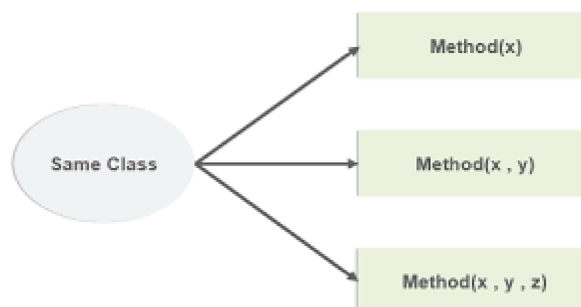
Example:

```
<?php
class demo
{
public function demo( )
{
echo "constructor1...";
}
}
class demo1 extends demo
{
public function __construct( )
{
echo parent::demo();
echo "constructor2...";
}
public function __destruct()
{
echo "destroy.....";
}
}
$obj= new demo1();
?>
```

Overloading

What is function overloading?

Function overloading is the ability to create multiple functions of the same name with different implementations.



Function overloading in PHP?

Function overloading in PHP is used to dynamically create properties and methods. Function overloading contains same function name and that function performs different task according to number of arguments. For example, find the area of certain shapes where radius are given then it should return area of circle if height and width are given then it should give area of rectangle and

others. In PHP function overloading is done with the help of function __call(). This function takes function name and arguments.

Property and Rules of overloading in PHP:

- All overloading methods must be defined as Public.
- After creating the object for a class, we can access a set of entities that are properties or methods not defined within the scope of the class.
- Such entities are said to be overloaded properties or methods, and the process is called as overloading.
- For working with these overloaded properties or functions, PHP magic methods are used.
- Most of the magic methods will be triggered in object context except __callStatic() method which is used in a static context.

Types of Overloading in PHP: There are two types of overloading in PHP.

- Property Overloading
- Method Overloading

Property Overloading:

PHP property overloading is used to create dynamic properties in the object context. For creating these properties no separate line of code is needed. A property associated with a class instance, and if it is not declared within the scope of the class, it is considered as overloaded property. Following operations are performed with overloaded properties in PHP.

- Setting and getting overloaded properties.
- Evaluating overloaded properties setting.
- Undo such properties setting.

Before performing the operations, we should define appropriate magic methods. which are,

- __set(): triggered while initializing overloaded properties.
- __get(): triggered while using overloaded properties with PHP print statements.
- __isset(): This magic method is invoked when we check overloaded properties with isset() function
- __unset(): Similarly, this function will be invoked on using PHP unset() for overloaded properties.

Example:

```
<?php
class Toys
{
private $str;
public function __set($name, $value)
{
```

```
$this->str[$name] = $value;
}
public function __get($name)
{
echo "Overloaded Property name = " . $this->str[$name] . "<br/>";
}
public function __isset($name)
{
if (isset($this->str[$name]))
{
echo "Property \$$name is set.<br/>";
}
else
{
echo "Property \$$name is not set.<br/>";
}
}
public function __unset($name)
{
unset($this->str[$name]);
echo "\$$name is unset <br/>";
}
}
$objToys = new Toys();
/* setters and getters on dynamic properties */
$objToys->overloaded_property = "new";
echo $objToys->overloaded_property . "\n\n";
/* Operations with dynamic properties values */
isset($objToys->overloaded_property);
unset($objToys->overloaded_property);

isset($objToys->overloaded_property);
?>
```

Method Overloading

It is a type of overloading for creating dynamic methods that are not declared within the class scope. PHP method overloading also triggers magic methods dedicated to the appropriate purpose. Unlike property overloading, PHP method overloading allows function call on both object and static context. The related magic functions are:

- __call() – triggered while invoking overloaded methods in the object context.
- __callStatic() – triggered while invoking overloaded methods in static context.

Example:

```
<?php
class Toys
{
public function __call($name, $param)
{
echo "Magic method invoked while method overloading with object reference<br/>";
}
public static function __callStatic($name, $param)
{
echo "Magic method invoked while method overloading with static access<br/>";
}
}
$objToys = new Toys();
$objToys->overloaded_method();
Toys::overloaded_property();
?>
```

Function Overriding: Function overriding is same as other OOPs programming languages. In function overriding, both parent and child classes should have same function name with and number of arguments. It is used to replace parent method in child class. The purpose of overriding is to change the behavior of parent class method. The two methods with the same name and same parameter is called overriding.

Example:

```
<?php
class Vehicle
{
function run()
{
echo "Vehicle is running<br>";
}
}
class Bike extends Vehicle
{
function run()
{
echo "Bike is running";
}
}
$v = new Vehicle;
$b= new Bike;
```

```
$v->run();
$b->run();
?>
```

Output:

Vehicle is running
Bike is running

INHERITANCE: Inheritance in PHP is a fundamental object-oriented programming concept that allows a class to inherit properties and methods from another class, referred to as the parent or base class.

- The class that inherits properties and methods is called the child or derived class.

Inheritance is one of the four pillars of Object-Oriented Programming (OOPs).

- Inheritance is the phenomenon by which a child class can inherit all the properties and characteristics of the parent class.

Uses of inheritance:

- **Code Reusability**: One of the primary advantages of inheritance is code reuse. By defining common properties and methods in a parent class, subclasses can inherit and reuse this code, reducing redundancy and promoting modular design.
- **Abstraction and Encapsulation**: Inheritance allows for abstraction by defining a generalized parent class that provides a common interface for its subclasses. This abstraction hides the implementation details of the parent class and allows subclasses to focus on specific behaviors or attributes. Encapsulation is also promoted as related properties and methods are grouped together in a single class hierarchy.
- **Promoting Polymorphism**: Inheritance facilitates polymorphism, which allows objects of different classes to be treated uniformly through a common interface. This enables flexibility in designing systems where different objects can respond to the same message or method call in different ways.
- **Enhancing Maintainability and Extensibility**: Inheritance promotes maintainability by centralizing common code in a parent class, making it easier to modify or update functionality without affecting multiple subclasses. It also enhances extensibility by allowing new subclasses to be created without modifying existing code, thus minimizing the risk of introducing bugs.
- **Simplifying Class Hierarchies**: Inheritance helps in organizing class hierarchies by creating a logical relationship between classes. This hierarchical structure makes it easier to understand and manage complex systems by grouping related classes together and defining their relationships.
- **Specialization and Generalization**: Inheritance allows for specialization, where subclasses can extend or override behavior inherited from a parent class to meet specific requirements.

Advantages and Disadvantages of inheritance in php

Advantages:

1. **Code Reusability**: Inheritance allows you to reuse code from existing classes in new classes, reducing redundancy and promoting a more modular design.

2. **Promotes Polymorphism**: Inheritance enables polymorphism, where objects of different classes can be treated interchangeably if they share a common parent class. This enhances flexibility and promotes cleaner, more flexible code.
3. **Encourages Abstraction**: Inheritance allows you to define a common interface or behavior in a parent class, which can be inherited by subclasses. This abstraction hides implementation details and promotes a clearer separation of concerns.
4. **Simplifies Maintenance**: By centralizing common functionality in a parent class, inheritance can make code maintenance easier. Changes made to the parent class are automatically propagated to its subclasses, reducing the risk of introducing errors.
5. **Enhances Extensibility**: Inheritance facilitates the creation of new subclasses without modifying existing code. This makes it easier to extend the functionality of a system by adding new features or behaviors.

Disadvantages:

1. **Tight Coupling**: Inheritance can lead to tight coupling between classes, where changes in the parent class can have unintended effects on subclasses. This can make the codebase more fragile and difficult to maintain.
2. **Inheritance Hierarchy Complexity**: As the inheritance hierarchy grows, it can become more complex and difficult to understand. This complexity can make it harder to reason about the behavior of subclasses and their relationships.
3. **Overuse**: Overuse of inheritance can lead to an overly complex class hierarchy, known as the "diamond problem" or "inheritance hell." This can make the codebase harder to maintain and extend.
4. **Inflexibility**: Inheritance ties subclasses to the implementation details of the parent class, making it harder to change or refactor the inheritance hierarchy without affecting existing code.

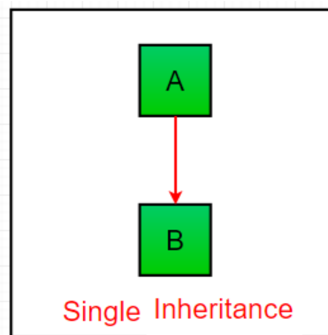
TYPES OF INHERITANCES: PHP offers mainly three types of inheritance based on their functionality. These three types are as follows:

1. **Single Inheritance:** Deriving a class from only one base class(super class) is known as Single inheritance. In below image, the class A serves as a base class for the derived class B.

- In single inheritance, a class can inherit properties and methods from only one parent class.
- PHP supports single inheritance directly. A class can extend only one parent class at a time.
- This is the most common type of inheritance in PHP.

Syntax:

```
class ParentClass {
   // Properties and methods
}
class ChildClass extends ParentClass {
   // Inherits properties and methods from ParentClass
}
```



Example:

```
<?php
class demo
{
public function display()
{
echo "Example of inheritance ";
}
}
class demo1 extends demo
{
public function view()
{
echo "in php";
}
}
$obj= new demo1();
$obj->display();
$obj->view();
?>
```

Output:

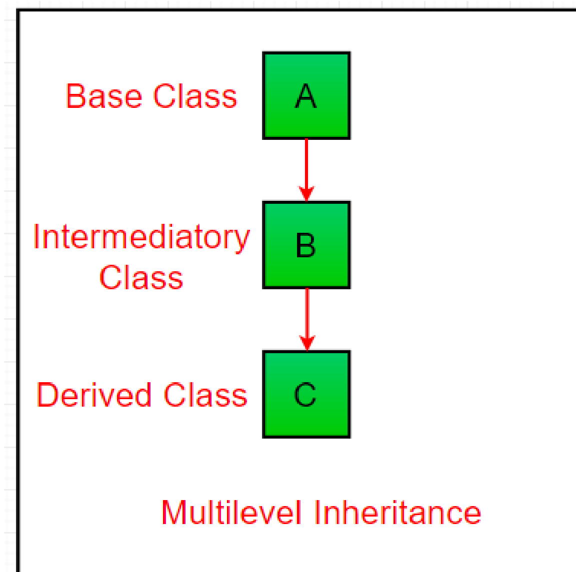
```
Example of inheritance
```

2. **Multilevel Inheritance:** Deriving a class from another derived class is known as Multi level inheritance. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

Syntax:

```
class ParentClass {
  // Parent class properties and methods
}
class ChildClass extends ParentClass {
  // Child class properties and methods
}
```

```
class GrandChildClass extends ChildClass {
   // Grandchild class properties and methods
}
```



Multilevel inheritance is a concept in object-oriented programming where a class extends another class, which in turn extends yet another class, forming a chain or hierarchy of inheritance. This creates a parent-child relationship between the classes involved, with each child class inheriting properties and methods from its parent class, as well as from all ancestors up the inheritance chain.
Here's a step-by-step explanation of multilevel inheritance:

1. **Parent Class**: At the top of the hierarchy is the parent class. This class typically contains common properties and methods that are shared among multiple child classes.
2. **First-level Child Class**: This class extends the parent class, inheriting all of its properties and methods. The first-level child class can also add its own unique properties and methods.
3. **Second-level Child Class (Optional)**: If desired, another class can extend the first-level child class, creating a second level of inheritance. This class inherits properties and methods from both the parent class and the first-level child class.
4. **Subsequent Levels (Optional)**: The process can continue with additional levels of inheritance, with each subsequent child class inheriting properties and methods from its immediate parent class and all ancestors up the inheritance chain.

Example:

```
<?php
class A
{
function showA()
{
echo "I am show method in A Class<br>";
}
```

```
}
class B extends A
{
function showB()
{
echo "I am show method in B Class<br>";
}
}
class C extends B
{
function showC()
{
echo "I am show method in C Class<br>";
}
}
$obj=new C;
$obj->showA();
$obj->showB();
$obj->showC();
?>
```

Output:

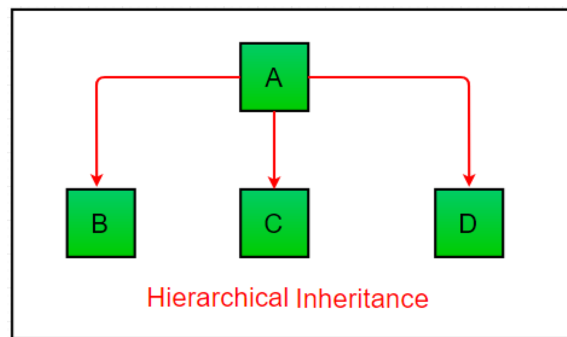
I am show method in A Class I am show method in B Class I am show method in C Class

Hierarchical Inheritance: Deriving several classes from one base class is known as Hierarchical inheritance. In below image, the class A serves as a base class for the derived class B,C and D.

In PHP, hierarchical inheritance is achieved by extending classes. Subclasses inherit properties and methods from their parent classes. This promotes code reuse and allows for a structured organization of code.

Syntax:

```
class ParentClass {
 // Parent class properties and methods
}
class ChildClass extends ParentClass {
// Child class properties and methods
}
```



Example:

```
<?php
// base class named "Jewellery"
class Jewellery
{
public function show()
{
echo 'This class is Jewellery ';
}
}
// Derived class named "Necklace"
class Necklace extends Jewellery
{
public function show()
{
echo 'This class is Necklace ';
echo"<br>";
}
}
// Derived class named "Necklace"
class Bracelet extends Jewellery
{
public function show()
{
echo 'This class is Bracelet ';
}
}
// creating objects of derived classes "Necklace" and "Bracelet"
$n = new Necklace();
$n->show();
$b = new Bracelet();
$b->show();
?>
```

Output:
This class is Necklace This class is Bracelet

Advantages of Inheritance
1. Reduce code redundancy.
2. Provides code reusability.
3. Reduces source code size and improves code readability.
4. The code is easy to manage and divided into parent and child classes.
5. Supports code extensibility by overriding the base class functionality within child classes.

Class Constants

PHP allows an identifier in a class to be defined as a "class constant" with a constant value, the one that remains unchanged on a per class basis. To differentiate from a variable or property within class, the name of the constant is not prefixed with the usual "$" symbol and is defined with the "const" qualifier. Note that a PHP program can also have a global constant created using the **define()** function.

The default visibility of a constant is public, although other modifiers may be used in the definition. The value of a constant must be an expression and not a variable, nor a function call/property. The value of a constant is accessed through the class name using the scope resolution operator. Inside a method though, it can be referred to through **self** variable.

Accessing Class Constants in PHP
Here is the syntax you can follow for accessing class constants in PHP −

```
class SomeClass {
  const CONSTANT = 'constant value';
}
echo SomeClass::CONSTANT;
```

Key Points About Class Constants
Here are some key points about class constant which you need to know before working with it −

- **Immutability:** Once it is set, the value cannot be modified.
- **Scope:** Class constants are accessible within the class that defines them .
- **Static:** They are automatically static, so you do not need to create a class instance to access them.

Why Use Class Constants?

Using class constants provides many advantages –

- **Readability:** Makes your code easier to read and understand.
- **Maintainability:** If you need to modify the value, do it only once.
- **Avoid Magic Numbers:** By avoiding magic numbers or strings in your code, you can better understand what each value represents.

Abstract Classes

An abstract class in PHP is a class that cannot be created on its own. This means you can't create objects straight from an abstract class. Abstract classes are intended to be extended by subsequent classes. They serve as a blueprint for other classes, defining the common methods and properties that inheriting classes must implement.

The list of reserved words in PHP includes the "abstract" keyword. When a class is defined with the "abstract" keyword, it cannot be instantiated, i.e., you cannot declare a new object of such a class. An abstract class can be extended by another class.

Here is the syntax you can use for defining the **abstract class** –

```
abstract class myclass {
  // class body
}
```

Create an Object of Abstract Class
As mentioned above, you **cannot declare an object of this class**. Hence, the following statement –

```
$obj = new myclass;
```

will result in an **error** message as shown below –

PHP Fatal error: Uncaught Error: Cannot instantiate abstract class myclass

An abstract class may include properties, constants or methods. The class members may be of public, private or protected type. One or more methods in a class may also be defined as abstract. If any method in a class is abstract, the class itself must be an abstract class. In other words, a normal class cannot have an abstract method defined in it.

This will raise an **error** –

```
class myclass {
  abstract function myabsmethod($arg1, $arg2);
  function mymethod() #this is a normal method {
    echo "Hello";
  }
}
```

he **error message** will be shown as –
PHP Fatal error: Class myclass contains 1 abstract method
and must therefore be declared abstract

You can use an abstract class as a parent and extend it with a child class. However, the child class must provide concrete implementation of each of the abstract methods in the parent class, otherwise an error will be encountered.

Why Use Abstract Classes?
There are two key reasons that you need to use abstract classes in PHP –

- **Code Reusability:** Abstract classes allow you to describe common attributes and methods in one place which reduces code duplication.
- **Enforcing Structure:** They require child classes to implement specific methods, resulting in a consistent structure throughout all classes.

Example :
In the following code, **myclass** is an **abstract class** with **myabsmethod()** as an **abstract method**. Its derived class is **mynewclass**, but it doesn't have the implementation of the abstract method in its parent.

```
<?php
  abstract class myclass {
    abstract function myabsmethod($arg1, $arg2);
    function mymethod() {
      echo "Hello";
    }
  }
  class newclass extends myclass {
    function newmethod() {
      echo "World";
    }
  }
  $m1 = new newclass;
  $m1->mymethod();
?>
```

The **error message** in such a situation is –

PHP Fatal error: Class newclass contains 1 abstract method and must
therefore be declared abstract or implement the remaining
methods (myclass::myabsmethod)

It indicates that newclass should either implement the abstract method or it should be declared as an abstract class.

Example

In the following PHP script, we have **marks** as an abstract class with percent() being an abstract method in it. Another **student** class extends the **marks** class and implements its percent() method.

```
<?php
  abstract class marks {
    protected int $m1, $m2, $m3;
    abstract public function percent(): float;
  }

  class student extends marks {
    public function __construct($x, $y, $z) {
      $this->m1 = $x;
      $this->m2 = $y;
      $this->m3 = $z;
    }
    public function percent(): float {
      return ($this->m1+$this->m2+$this->m3)*100/300;
    }
  }

  $s1 = new student(50, 60, 70);
  echo "Percentage of marks: ". $s1->percent() . PHP_EOL;
?>
```

It will produce the following **output** –
Percentage of marks: 60

Important Facts About Abstract Classes in PHP

1. Cannot Create Instances of an Abstract Class

Like Java, an instance of an abstract class cannot be created in PHP.

2. Abstract Classes Can Have Constructors

Like in C++ and Java, abstract classes in PHP can have constructors.php

3. Abstract Methods Cannot Have a Body in PHP

Unlike Java, PHP does not allow an abstract method to have a body.

Interfaces

Just as a class is a template for its objects, an **interface** in PHP can be called as a template for classes. We know that when a class is instantiated, the properties and methods defined in a class are available to it. Similarly, an interface in PHP declares the methods along with their arguments and return value. These methods do not have any body, i.e., no functionality is defined in the interface.

A **concrete** class has to implement the methods in the interface. In other words, when a class implements an interface, it must provide the functionality for all methods in the interface.

Create an Interface in PHP

An interface is defined in the same way as a class is defined, except that the keyword "**interface**" is used in place of class.

```
interface myInterface {
  public function myfunction(int $arg1, int $arg2);
  public function mymethod(string $arg1, int $arg2);
}
```

Note that the methods inside the interface have not been provided with any functionality. Definitions of these methods must be provided by the class that implements this interface.

When we define a child class, we use the keyword "**extends**". In this case, the class that must use the keyword "**implements**".

Using the Interface

All the methods declared in the interface must be defined, with the same number and type of arguments and return value.

```
class myclass implements myinterface {
  public function myfunction(int $arg1, int $arg2) {
    ## implementation of myfunction;
  }
  public function mymethod(string $arg1, int $arg2) {
    # implementation of mymethod;
  }
}
```

Note that all the methods declared in an interface must be public.

Difference between Interface and Abstract Class in PHP

The concept of abstract class in PHP is very similar to interface. However, there are a couple of differences between an interface and an abstract class.

Abstract class	**Interface**
Use abstract keyword to define abstract class	Use interface keyword to define interface
Abstract class cannot be instantiated	Interface cannot be instantiated.
Abstract class may have normal and abstract methods	Interface must declare the methods with arguments and return types only and not with any body.
Abstract class is extended by child class which must implement all abstract methods	Interface must be implemented by another class, which must provide functionality of all methods in the interface.
Can have public, private or protected properties	Properties cannot be declared in interface

Exception Handling

What Is Exception Handling?
Exception handling is a mechanism in programming that allows a system to handle unexpected events or errors that occur during the execution of a program. These unexpected events, known as exceptions, can disrupt the normal flow of an application. Exception handling provides a controlled way to respond to these exceptions, allowing the program to either correct the issue or gracefully terminate.

Why Do We Need Exception Handling?

1. **Maintaining Application Flow:** Without exception handling, an unexpected error could terminate the program abruptly. Exception handling ensures that the program can continue running or terminate gracefully.
2. **Informative Feedback:** When an exception occurs, it provides valuable information about the problem, helping developers to debug and users to understand the issue.
3. **Resource Management:** Exception handling can ensure that resources like database connections or open files are closed properly even if an error occurs.

4. **Enhanced Control:** It allows developers to specify how the program should respond to specific types of errors.

What is an Exception

Exception handling is used to change the normal flow of the code execution if a specified error (exceptional) condition occurs. This condition is called an exception.

This is what normally happens when an exception is triggered:

- The current code state is saved
- The code execution will switch to a predefined (custom) exception handler function
- Depending on the situation, the handler may then resume the execution from the saved code state, terminate the script execution or continue the script from a different location in the code

Basic Use of Exceptions

When an exception is thrown, the code following it will not be executed, and PHP will try to find the matching "catch" block.

If an exception is not caught, a fatal error will be issued with an "Uncaught Exception" message.

Lets try to throw an exception without catching it:

```
<?php
//create function with an exception
function checkNum($number) {
  if($number>1) {
    throw new Exception("Value must be 1 or below");
  }
  return true;
}

//trigger exception
checkNum(2);
?>
```

The code above will get an error like this:
Fatal error: Uncaught exception 'Exception'
with message 'Value must be 1 or below' in C:\webfolder\test.php:6
Stack trace: #0 C:\webfolder\test.php(12):
checkNum(28) #1 {main} thrown in **C:\webfolder\test.php** on line **6**

Try, throw and catch

To avoid the error from the example above, we need to create the proper code to handle an exception.

Proper exception code should include:

1. **try** - A function using an exception should be in a "try" block. If the exception does not trigger, the code will continue as normal. However if the exception triggers, an exception is "thrown"
2. **throw** - This is how you trigger an exception. Each "throw" must have at least one "catch"
3. **catch** - A "catch" block retrieves an exception and creates an object containing the exception information
4. **Finally** - The finally block may also be specified after or instead of catch blocks. Code within the finally block will always be executed after the try and catch blocks, regardless of whether an exception has been thrown, and before normal execution resumes. This is useful for scenarios like closing a database connection regardless if an exception occurred or not.

Lets try to trigger an exception with valid code:

```
<?php
//create function with an exception
function checkNum($number) {
  if($number>1) {
    throw new Exception("Value must be 1 or below");
  }
  return true;
}

//trigger exception in a "try" block
try {
  checkNum(2);
  //If the exception is thrown, this text will not be shown
  echo 'If you see this, the number is 1 or below';
}

//catch exception
catch(Exception $e) {
  echo 'Message: ' .$e->getMessage();
}
?>
```

The code above will get an error like this:
Message: Value must be 1 or below

Example explained:
The code above throws an exception and catches it:

1. The checkNum() function is created. It checks if a number is greater than 1. If it is, an exception is thrown
2. The checkNum() function is called in a "try" block
3. The exception within the checkNum() function is thrown
4. The "catch" block retrieves the exception and creates an object ($e) containing the exception information
5. The error message from the exception is echoed by calling $e->getMessage() from the exception object

However, one way to get around the "every throw must have a catch" rule is to set a top level exception handler to handle errors that slip through.

Creating a Custom Exception Class
To create a custom exception handler you must create a special class with functions that can be called when an exception occurs in PHP. The class must be an extension of the exception class.
The custom exception class inherits the properties from PHP's exception class and you can add custom functions to it.
Lets create an exception class:

```
<?php
class customException extends Exception {
  public function errorMessage() {
    //error message
    $errorMsg = 'Error on line '.$this->getLine().' in '.$this->getFile()
    .': <b>'.$this->getMessage().'</b> is not a valid E-Mail address';
    return $errorMsg;
  }
}

$email = "someone@example...com";

try {
  //check if
  if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE) {
    //throw exception if email is not valid
    throw new customException($email);
```

```
  }
}

catch (customException $e) {
  //display custom message
  echo $e->errorMessage();
}
?>
```

The new class is a copy of the old exception class with an addition of the errorMessage() function. Since it is a copy of the old class, and it inherits the properties and methods from the old class, we can use the exception class methods like getLine() and getFile() and getMessage().

Example explained:

The code above throws an exception and catches it with a custom exception class:

1. The customException() class is created as an extension of the old exception class. This way it inherits all methods and properties from the old exception class
2. The errorMessage() function is created. This function returns an error message if an e-mail address is invalid
3. The $email variable is set to a string that is not a valid e-mail address
4. The "try" block is executed and an exception is thrown since the e-mail address is invalid
5. The "catch" block catches the exception and displays the error message

Multiple Exceptions

It is possible for a script to use multiple exceptions to check for multiple conditions.

It is possible to use several if..else blocks, a switch, or nest multiple exceptions. These exceptions can use different exception classes and return different error messages:

```
<?php
class customException extends Exception {
  public function errorMessage() {
    //error message
    $errorMsg = 'Error on line '.$this->getLine().' in '.$this->getFile()
    .': <b>'.$this->getMessage().'</b> is not a valid E-Mail address';
    return $errorMsg;
  }
}

$email = "someone@example.com";

try {
  //check if
```

```
  if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE) {
    //throw exception if email is not valid
    throw new customException($email);
  }
  //check for "example" in mail address
  if(strpos($email, "example") !== FALSE) {
    throw new Exception("$email is an example e-mail");
  }
}

catch (customException $e) {
  echo $e->errorMessage();
}

catch(Exception $e) {
  echo $e->getMessage();
}
?>
```

Example explained:

The code above tests two conditions and throws an exception if any of the conditions are not met:

1. The customException() class is created as an extension of the old exception class. This way it inherits all methods and properties from the old exception class
2. The errorMessage() function is created. This function returns an error message if an e-mail address is invalid
3. The $email variable is set to a string that is a valid e-mail address, but contains the string "example"
4. The "try" block is executed and an exception is not thrown on the first condition
5. The second condition triggers an exception since the e-mail contains the string "example"
6. The "catch" block catches the exception and displays the correct error message

If the exception thrown were of the class custom Exception and there were no custom Exception catch, only the base exception catch, the exception would be handled there.

Re-throwing Exceptions

Sometimes, when an exception is thrown, you may wish to handle it differently than the standard way. It is possible to throw an exception a second time within a "catch" block.

A script should hide system errors from users. System errors may be important for the coder, but are of no interest to the user. To make things easier for the user you can re-throw the exception with a user friendly message:

```
<?php
class customException extends Exception {
  public function errorMessage() {
    //error message
    $errorMsg = $this->getMessage().' is not a valid E-Mail address.';
    return $errorMsg;
  }
}

$email = "someone@example.com";

try {
  try {
    //check for "example" in mail address
    if(strpos($email, "example") !== FALSE) {
      //throw exception if email is not valid
      throw new Exception($email);
    }
  }
  catch(Exception $e) {
    //re-throw exception
    throw new customException($email);
  }
}

catch (customException $e) {
  //display custom message
  echo $e->errorMessage();
}
?>
```

Example explained:

The code above tests if the email-address contains the string "example" in it, if it does, the exception is re-thrown:

1. The customException() class is created as an extension of the old exception class. This way it inherits all methods and properties from the old exception class
2. The errorMessage() function is created. This function returns an error message if an e-mail address is invalid
3. The $email variable is set to a string that is a valid e-mail address, but contains the string "example"
4. The "try" block contains another "try" block to make it possible to re-throw the exception

5. The exception is triggered since the e-mail contains the string "example"
6. The "catch" block catches the exception and re-throws a "customException"
7. The "customException" is caught and displays an error message

If the exception is not caught in its current "try" block, it will search for a catch block on "higher levels".

Set a Top Level Exception Handler

The set_exception_handler() function sets a user-defined function to handle all uncaught exceptions:

```
<?php
function myException($exception) {
  echo "<b>Exception:</b> " . $exception->getMessage();
}

set_exception_handler('myException');

throw new Exception('Uncaught Exception occurred');
?>
```

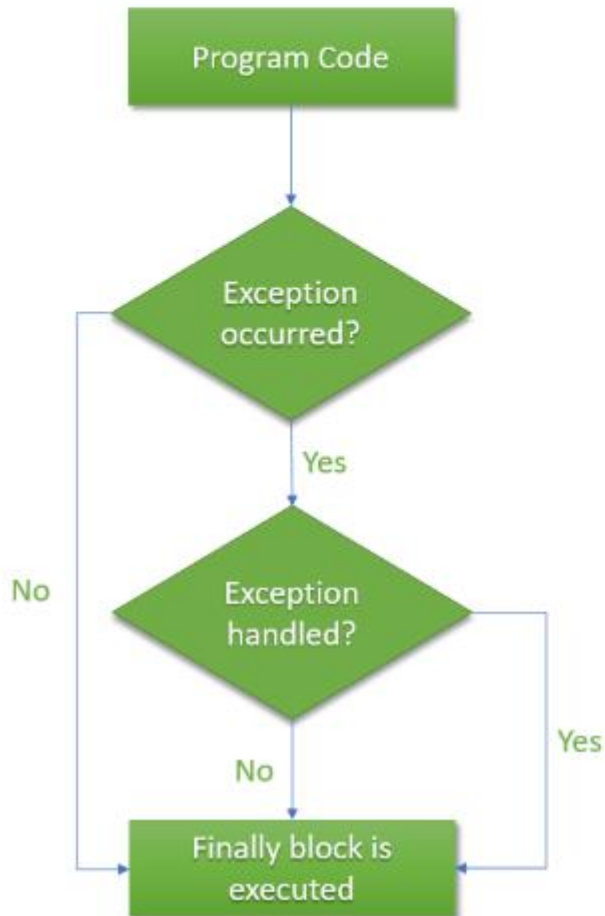
The output of the code above should be something like this:
Exception: Uncaught Exception occurred
In the code above there was no "catch" block. Instead, the top level exception handler triggered. This function should be used to catch uncaught exceptions.

When to Use try-catch-finally

PHP version 5.5, introduced the finally block. Sometimes in your PHP error handling code, you will also want to use a finally section. Finally is useful for more than just exception handling, it is used to perform cleanup code such as closing a file, closing a database connection, etc.
The finally block always executes when the try catch block exits. This ensures that the finally block is executed even if an unexpected exception occurs.
Example for try catch-finally:

```
try {
   print "this is our try block n";
   throw new Exception("An error occurred!");
} catch (Exception $e) {
   print "something went wrong, caught yah! n";
} finally {
   print "this part is always executed n";
}
```

Below is the diagram showing how the program works.



Rules for exceptions

- Code may be surrounded in a try block, to help catch potential exceptions
- Each try block or "throw" must have at least one corresponding catch block
- Multiple catch blocks can be used to catch different classes of exceptions
- Exceptions can be thrown (or re-thrown) in a catch block within a try block

A simple rule: If you throw something, you have to catch it.