PHP マニュアル
オブジェクト指向入門編

1.オブジェクト指向の概要
・なぜオブジェクト指向は難しいのか
・難しいのは「当然すぎる」から
・実は不自然だった手続き指向
・「アクセス制御」が不可欠
・メッセージ・パッシングとは何か
・関数呼び出しとメッセージ・パッシングの違い
・オブジェクトとクラスの違い

2.オブジェクト指向プログラミング
・クラス
・オブジェクト
・インスタンス化
・コンストラクタ、デストラクタ
・アクセス修飾子
・ゲッダメソッド、セッダメソッド
・静的メソッド
・クラス内定数
・継承
・parent 命令
・final 修飾子
・ポリモーフィズム
・抽象メソッド
・インターフェス

3.PHP5独自のオブジェクト構文
・_autoload 関数
・_call メソッド
・_get,_set メソッド

4.Apache との連携
・.htaccess
・http.conf の AllowOverride ディレクティブを"All"に設定
・.htaccess ファイルで設定可能なパラメータは決まっている

1.オブジェクト指向の概要

「オブジェクト指向は分かりにくい」 登場から 30 年以上経っても、未だにオブジェクト指向に対する敷居は高いままです。ここではオブジェクト指向について考えていきたいと思います。今や IT 関連製品や技術は、オブジェクト指向抜きには語れないと言ってよいほど普及し一般的になっています。Java や C++をはじめとするプログラミング言語に及ばず、パソコンのユーザー・インターフェースは、ひとつひとつのプログラムをアイコンで表し、直感的な操作を可能にする GUI（グラフィカル・ユーザー・インタフェース）が主流です。GUI は言うまでもなくオブジェクト指向の考え方で作られたものです。

そもそもオブジェクト指向について正確に解説すると、「プログラムの構造をオブジェクト群の相互作用とその雛形であるクラス群の関係として捉え、相互にメッセージを送りあう オブジェクトの集まりとしてプログラムを構成する技法のことを指します」とあるように非常に説明しずらくわかりずらいものとなっています。このオブジェクト指向を取り入れたプログラミング言語はオブジェクト指向プログラミング言語と呼ばれ、これらの言語ではクラスとその継承などの仕組みを利用することで開発効率を高められるようになっています。オブジェクト指向構文は今となっては開発現場ではあたりまえとなっていますが、習得するのはなかなか難しいのが現状です。特に大多数でシステム構築するような複雑で大規模なものほどオブジェクト指向という考え方はなくてはならなくなってきますので習得するにはそれなりの時間がかかりますが必ず習得してほしい分野です。現在では C++、Java、JavaScript、Ruby、PHP、Perl などほとんどの開発現場でこのオブジェクト指向という考え方を元にした開発言語が主流になってきています。今やオブジェクト指向は、システム開発における“共通の土台”となりつつある考え方です。逆に言えば、オブジェクト指向の考え方を習得することでこれらの言語も初見でもある程度は理解することが可能となります。難しいと思いますが、少しづつでいいので理解できるまで何度もチャレンジしてくれることを願っています。

オブジェクト指向の概要では実際の PHP でのコード記述紹介のまえにオブジェクト指向とはどういったものなのかということを現実世界のモノに例え解説していきます。

✓ なぜオブジェクト指向は難しいのか

オブジェクト指向の難しさはどこにあるのか。よく言われるのは耳慣れない用語とオブジェクト指向という考え方にあると聞きます。オブジェクト指向にはこれから紹介する「クラス」、「オブジェクト」、「継承」、「抽象化」、「メッセージ」、「振る舞い」などといった IT らしくない言葉が多数登場します。これらの言葉が災いして、慣れない人からすれば、なぜそんな言い回しをするのかと不思議と思うのが当然です。オブジェクト指向の定義も実はまちまちで、オブジェクト指向の解説書や、オブジェクト指向言語の代名詞でもある Java の参考書ですら表現が異なっているので混乱するのも当然といえば当然です。もしかしたら、わざと混乱させるためにこんなまわりくどい言い回しをつかっているんじゃないかと疑うこともあります。

書名	出版社	内容
Java プログラムデザイン	ソフトバンク パブリッシング	「オブジェクト指向とは，実世界のことがらや仮想的な事象の1つひとつを対象にしてモジュール化するパラダイムである」，「オブジェクトとは，1つの実体として頭に描くことのできる概念を表現したものである」
オブジェクト 指向の基礎	ソフト・リサーチ・ センター	「オブジェクト指向システムとは，オブジェクト，メッセージ・パッシング，カプセル化，クラスとインスタンス，階層構造，継承，ポリモルフィズムというすべての性質を満たしたシステム」，「オブジェクトとはデータ構造とこれを操作する演算とを一体化させたもの」
図解でわかる 分散オブジェクト 技術のすべて	日本実業出版社	「オブジェクト指向とは，『物』を中心にした考え方であるとともに，それぞれのオブジェクト同士の関係において，ソフトウェアを構築し，動作させるための技術」，「処理対象を，必要としたデータを中心に持つ独立した『物』としてとらえ，それぞれのオブジェクトの相互的な作用によって処理を行う手法」

そしてもう一点、従来の手法（手続き型言語などの非オブジェクト指向言語）になれたプログラマやエンジニアにとってオブジェクト指向を理解するには、180度異なる発想が必要とされる点があるためです。オブジェクト指向が登場する以前は、プロセッサが行うひとつひとつの作業、すなわち「手続き」を順に記述して処理をこなしていく方法しかなかったため、この考え方でプログラムを考えてしまいオブジェクト指向での発想転換が難しいのです。

✓ 難しいのは「当然すぎる」から

では、これらの難しさをどのように乗り越えていけばいいのか。それには用語やテクニックにとらわれることなく、オブジェクト指向の根本的な考え方を理解するということがキーになってきます。それができれば最初から苦労はしないのですが、そこさえ乗り越えれば様々な難解な概念もおのずと意味が分かり、実際のシステム構築の場でも生きてきます。その意味でぜひ覚えておくべき大原則として「オブジェクト指向とは人間にとってごく自然な考え方」というものがあります。あなたも自分にとって当たり前すぎることを改めて問いただされたら、答えに窮するのではないでしょうか。オブジェクト指向が難しい理由は、オブジェクト指向が高度だからではなく、オブジェクト指向とは、「ソフトウェアを人間にとって扱いやすいよう、現実の物事になぞらえて考える」手法だからです。手続き指向に慣れた、あるいは手続き指向のシステムを担当する多くの IT エンジニアにとっては、このごく自然な考え方を理解することがかえって難しいのです。

✓ 実は不自然だった手続き指向

「でも、本当にオブジェクト指向は自然な考え方なのか？ 手続き指向でひとつひとつ処理を記述していった方が、よっぽど簡単だし自然なような気がするけど」 こんなことを思っている方もいると思います。手続き指向では、プログラムに対してデータを入力して何らかの処理を実行する、つまりプログラム（処理）とデータをまったく別物として扱う考え方でした。これはプログラム上だけの話ではなく、実際、コンピュータの中ではプログラムとデータを物理的に異なるメモリー領域に格納していることからも伺えます。注目してほしいのは、コンピュータ内に存在しているデータは特別な意味を持たず、単なる数字の羅列（2進法で考えて）に過ぎないということです。例えば、コンピュータの中に「1000」というデータが存在するとします。すると、そのデータは指定された処理（プログラム）によっては「1000 円」にも「1000 個」にも意味を変えます。

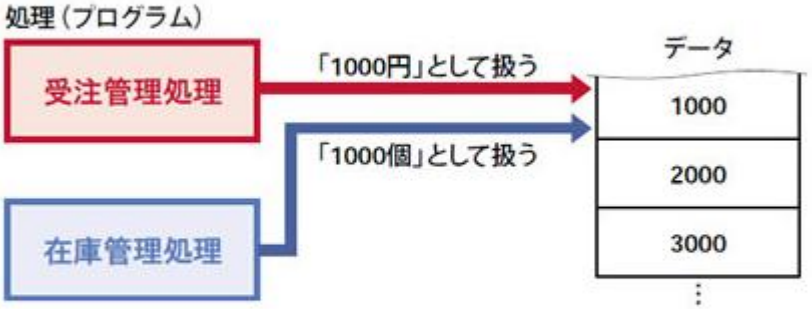
ところが、現実の物事のあり方と照らし合わせてみると、コンピュータ流の方式は極めて不自然です。なぜなら、現実の世界では「1000」という数値だけが存在することはあり得ないからです。普段はあまり意識したことがないかもしれませんが、必ずデータには固有の意味があります。「1000 円」、「1000 個」、「1000km」、「1000 人」といった具合です。

そうして考えてみるとひとつの疑問がわくと思います。「なぜコンピュータは、こんな方式でプログラムとデータを分離して管理しているのだろうか？」、と。答えは簡単で、コンピュータ内部の処理が手続き指向だからです。コンピュータのプロセッサはメモリー上のプログラムを順番に読み込みデータを処理しています。データにどんな意味があるのかは全く関与していません。加えて昔のコンピュータでは処理性能が現在とは桁違いに低かったため貧弱なコンピュータ資源を有効活用するにはコンピュータが最も処理のしやすい形態、つまり手続き指向でプログラムを構築するしかなかったのです。しかし現在では処理性能の向上により、人間にとって分かりやすく使いやすいオブジェクト指向という考え方が浸透してきたというだけの話です。そして前述したようにデータが固有の意味を持つということは、同時にそのデータを使って実行できる処理も必然的に決まります。例えば「1000 円」というお金をつかってできることは、「代金を支払う」とか「両替する」といったごく限られたことしかできません。オブジェクト指向とは、まさにこうした現実と同じような物事のありようを、ソフトウェアで実現しようという試みそのものなのです。

では、どうすればモノのように扱えるソフトをつくれるのか。答えを言えば、処理とデータをひとまとめのセットとして扱えばよいということです。具体的には、このデータとこの処理はひとつでワンセットであるとプログラム上で定義し、同時にデータへアクセスすることのできる処理をこのワンセットのみに限定します。例えば、「1000 円」というお金のデータを表現したいとき、1000 というデータとそのデータがお金であるということを前提にしたプログラムをひとつのセットとしてまとめます。これで、単なる 1000 というデータが 1000 円というお金のデータに固定できます。これがオブジェクト指向の解説書に当たり前のように出てくる「処理とデータの一体化」と同じ意味になります。そしてこのデータの意味を固定した処理をオブジェクト指向ではメソッドと名付け、1000 円というデータを属性（プロパティ）と名付けただけのことなのです。そしてここまで説明してきたような具体的な処理とデータをひとまとめのセットにしたものがオブジェクトにあたります。

手続き指向

・データと処理が分離している
・データそのものは特定の意味をもたず，処理内容によって意味の異なるデータとして扱われる



手続き指向では，データと処理が分離している。これに対してオブジェクト指向のシステムではデータと処理をひとまとめにして扱い，オブジェクトは他のオブジェクトが持つデータへは直接アクセスできない

注）オブジェクト指向ではオブジェクトが持つ処理をメソッド，
データ項目を属性と呼ぶ

オブジェクト指向

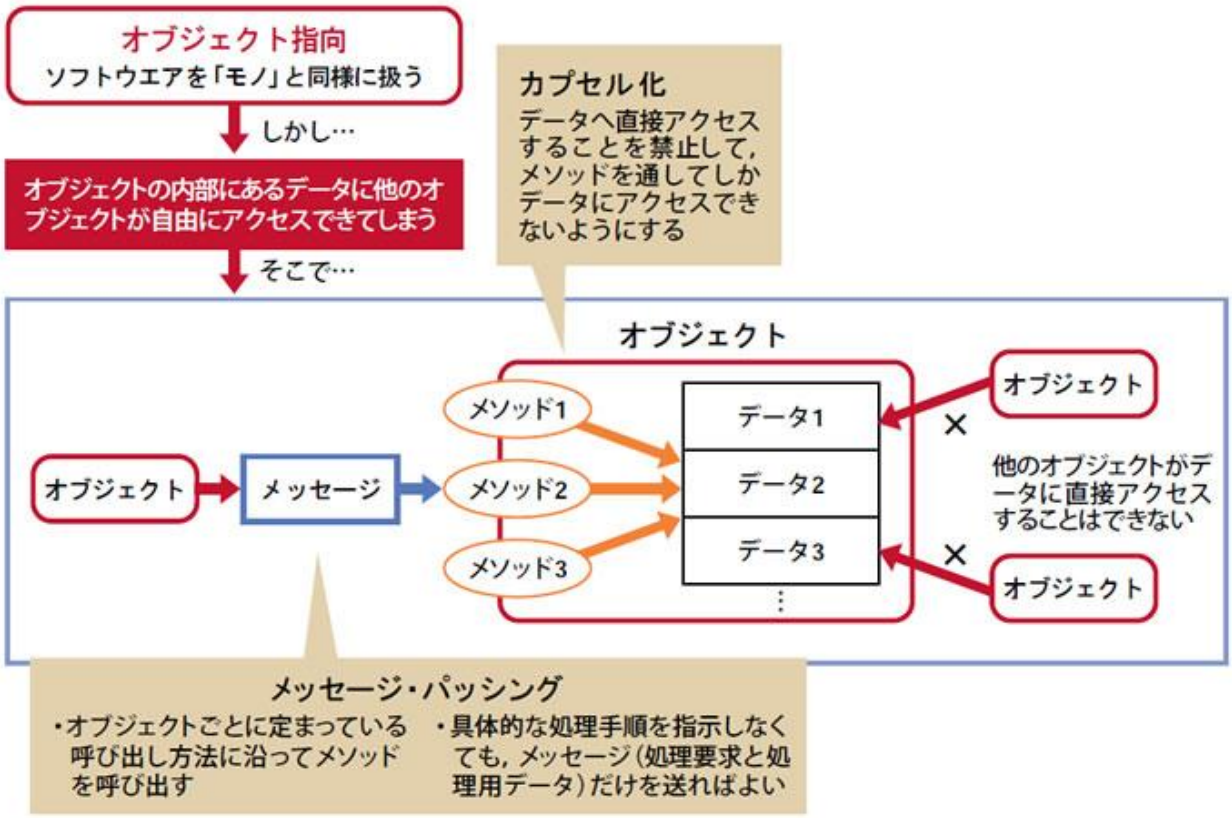
・データと処理を一体化し，各オブジェクトが持つデータへは，そのオブジェクト内の処理(メソッド)のみがアクセスできる(カプセル化)

✓ **「アクセス制御」が不可欠**

しかし一体化だけではまだ不十分です。いくら「データとメソッドを一体化してオブジェクトをつくった」といっても、それをつくったプログラマ以外の人間がその作法に従わなければ元も子もないためです。やろうと思えば、オブジェクト指向の考え方を無視してデータを直接参照したり変更したりとオブジェクト外部からやりたい放題する危険性があります。

この事態を防止してオブジェクト指向の目的を達成するためには、外部からのオブジェクト内部へとつながる自由なアクセスを遮断しなければいけません。そこで設けられたのがアクセス制御です。データアクセスの入り口を制御して属性にアクセスできるのはオブジェクトのなかのメソッドだけに限定します。そうすれば外部から属性を参照したい場合はオブジェクトのなかにあるメソッドを介してしかアクセスできなくなります。不正な行為をしようとするプログラムがあればこのメソッドの段階でアクセスをさせないようにすればいいのです。簡単にいえば両替機のなかのお金を盗まれないためにお金が入った箱に鍵をかけてなおかつ、偽札が両替機に入ってきたら無効にする機能がアクセス制御にあたります。

オブジェクト指向ではこのオブジェクト内のメソッド（偽札防止機能がついた両替機能）のことをインターフェースと呼びます。また、お金が入った箱がいくらはいっているのか外部から見えないようにすることをオブジェクト指向でも現実世界と同じように「隠蔽（いんぺい）」すると呼びます。現実世界にあるモノ（両替機）を表現するための「一体化」と「隠蔽」の説明は以上です。最後にこのような「一体化」と「隠蔽」をそなえもたせること（現実世界では両替機に防犯機能をもたせること）をオブジェクト指向では「カプセル化」と呼びます。



✓ メッセージ・パッシングとは何か

現実のモノをソフトウェアに置き換える際にカプセル化という名の防犯機能がいかに重要かということをお分かりいただけたと思います。では、このオブジェクトに何らかの処理を実行させる（両替機に実際のお金を両替させてもらう）ためにはどうすればいいのかをこれからお話します。

話は簡単で現実世界の私たちは両替機の内部の仕組みを知りません。というか両替機をつくって販売している会社（この場合はつくったプログラマにあたる）は両替機の内部の仕組みを使用者に詳しく教えません。なぜなら、両替機の内部の仕組みを教えてしまえば、仕組みの隙間をついて精巧な偽札がつくれてしまうからです。これでは悪用してくださいと言っているのと同じです。なんということでしょう、私たちは両替機の内部の仕組みを知らずに自然と両替機を使用していたのです！　・・・当たり前ですが。私たちが両替機に対して知っている知識といえば千円をいれれば百円玉が１０枚返却口からでてくるというごくあたりまえの両替機に対する使用方法ぐらいのものです。なにをいれれば、なにが返ってくるのか。そう、最初になにを両替機に入れれば、なにが返却口から戻ってくるのか。入力値と結果さえわかっていれば詳しい内容（両替機の内部の仕組み）など知ったことではないのです。

オブジェクト指向も同じです。結果からいうと、カプセル化されたオブジェクトの中身など知ったことではないのです。なにを入力値として渡せば、なにが結果として返ってくるのか。それさえわかっていれば処理の内容を知らなくても利用できます。最低限必要な知識としては両替機の使い方、つまり、オブジェクトの使い方だけなのです。ここで登場するのは上記の図にあるメッセージ・パッシングという用語です。

メッセージ・パッシングとやけに格好付けて名乗っていますが、用は両替機の使い方と同じ意味なのです。例えば、5000 円札を両替機に入れると「1000 円札に両替」ボタンと「100 円玉に両替」ボタンが押せるようになります。メッセージ・パッシングでは、「これから 5000 円札を入れるので 100 円玉に両替してください」というメッセージを両替機につたえていると考えればなんのことはないのです。

✓ 関数呼び出しとメッセージ・パッシングの違い

手続き型手法をつかったことがある人ならばなぜわざわざ処理を呼び出すのに関数呼び出しという用語があるにもかかわらず、メッセージ・パッシングなどという新しい用語をもってくるんだと卑屈になっている方がいるかもしれません。それは、そもそも関数とオブジェクトのなかにある処理は動きが異なっているので同じ呼び名では混乱するため、あえてメッセージ・パッシング（または、単にメッセージと呼ぶ）といった具合に分けているのです。

そもそもデータという観点から見ると関数とオブジェクトではぜんぜん扱いが異なります。そのため関数では関数呼び出しと、オブジェクトではメッセージ・パッシングと混乱を避けるために呼び名を区別したというわけです。かえってこれが混乱の元になることもあるので注意してください。関数とオブジェクトのデータの扱いに関する違いはオブジェクト指向プログラミングの章で解説していますのでそちらを参照してください。

✓ オブジェクトとクラスの違い

先ほどのメッセージ･パッシングの説明で、「カプセル化されたオブジェクトの中身など知ったことではない」と言いましたが、オブジェクト指向の最大のメリットは、プログラム全体を理解せずともプログラムを書けるという点です。使用方法さえわかっていればいいだけなので、オブジェクト内部を理解する時間をそのまま省くことができます。これにより開発スピードが格段にあがるという利点があります。

これはとくに大人数でシステムを構築する際、機能ごとに分業が可能になるということも意味しています。しかし、私たちはあくまでもつくる側。つまり、両替機をつくって販売している会社の技術者なので両替機が故障したり、新しい偽札防止機能を追加してくれと偉い人に命令されれば両替機の中身に機能追加したり、あたらしくつくり変えることをしなければなりません。

あなたが両替機の技術者として働いていたとしましょう。偉い人が唐突に「明日までに親会社が新しくつくった偽札防止機能を急遽、追加してくれ。できなければクビだ」と通告されたとしたらどうでしょう。あなたはまずなにをしますか？　すでに出来上がっている実際の両替機（この場合はオブジェクト）を分解し、仕組みを理解しようとすると思います。しかし両替機の偽札防止機能の箇所は外注で他の会社の技術者がつくったものなので仕組みがいまいち理解できません。しかも、その外注した会社は先月、唐突に倒産してしまい連絡すらとれない状態です。親会社のお高くとまっている技術者が親切に教えてくれるわけがない（前回も怒鳴りながら電話口にでてきたしな･･･）。さて、あなたはここでどう行動にでますか？

1.気合で何とかする。
2.こっそりオリジナルの偽札防止機能にすりかえる。
3.他の外注先をいまから探す。
4.コンビニでとらば一ゆを立ち読みしてから、明日までに辞表を書く。
5.そういえば納品されたものに設計書があったよな。それをみれば、もしかして･･･。

あなたの行動する選択肢として答えは5だと願っています。そう、ここでいう設計書こそオブジェクト指向にあたるクラスになります。クラスとはオブジェクトの設計書。実際のモノである両替機を、文面に抽象化し要約した設計図にあたるのです。

まとめ

いままでのながれでクラスとはオブジェクトの設計図でオブジェクトはクラスをもとにつくられたプログラムとわかっていただけたと思います。もう少し噛み砕いて説明すると、クラスとはオブジェクトの特徴（どんな性質があって、どんな動きをするのか）をひとくくりにまとめたもので、オブジェクトは実際のモノとして考えてください。

以上で簡単ですが現実世界の例を用いたオブジェクト指向の解説は終わりです。
次からは PHP の文法をまじえたオブジェクト指向編へと続きます。
難しいと思いますが、がんばりましょう。

2.オブジェクト指向プログラミング

- クラス

クラスとは、一般に関数の集合体として定義されていますが、処理の集合として捉えて考えた場合、関数と比べデータの扱いという点で見比べてみるとその違いがよりわかりやすくなります。関数は与えられた入力値（引数）を元にして処理結果を戻り値として出力するだけの仕組みですが、クラスまたはオブジェクトは、自分自身のなかにデータを格納する変数としての役割もはたしています。その結果、関数でのデータの扱いは、データが処理を「通過」していくだけなのにたいして、クラス、オブジェクトではさらに処理前、処理後のデータを内部で保持しておくことが可能になります。プログラマは必要に応じこの値を取り出して再度利用することができるようになるわけです。関数とクラスの違いは以上です。わかりやすく下記にその違いをまとめてみました。

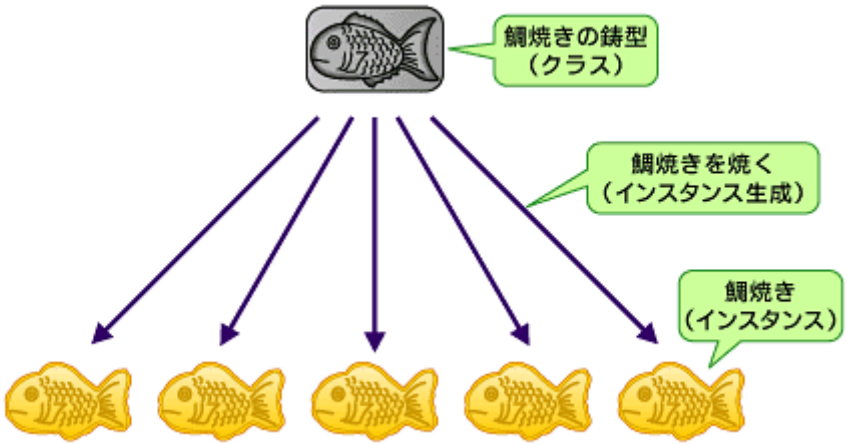
	データを保持	データを処理
変数	○	×
関数	×	○
クラス、オブジェクト	○	○

- オブジェクト

クラスを理解するうえで重要になってくるのがオブジェクトです。クラスとオブジェクトは似て非なるものです。オブジェクトとはクラスを元にしてつくられたコピーを指します。クラスでは「データを保持できる」という性質をもっているため、複数の処理でひとつのクラスにアクセスした場合、データの上書き（不整合）を引き起こす危険があります。そのため、処理に応じクラスのコピーを作成し、そのコピーに対してアクセスすることが考えだされました。実際に使う場合には、まずクラス本体には手を加えず、オブジェクトと呼ばれるクラスのコピーを作成しそれを処理の対象としてプログラムを組んでいきます。

・例えばこんな感じ・・・

この後にも紹介しますが、クラス（雛形/設計図）からオブジェクト（実際のモノ）を生成することをインスタンス化と呼びます。処理をする場合、クラスには手を加えずインスタンス化されたオブジェクトを処理対象として扱います。下の図の場合、インスタンスとはオブジェクトと同じ意味になります。



- インスタンス化

クラスのコピー（オブジェクト）を生成することを「インスタンス化」といいます。インスタンス化とは、クラスを扱うための「自分専用の領域」を確保する行為ともいえます。PHP では（他の言語でも大抵は同意です）、クラスのインスタンス化を次のように new 演算子を使って行ないます。また、インスタンス化の際に、必要に応じて初期化のためのデータを引き渡すことも可能です。

```
$変数名 = new クラス名([引数,...]);
```

引数が必要ない場合でもカッコは省略できない点に注意してください。インスタンス化したオブジェクトは「$変数名」に格納され、以降はこの変数をオブジェクトとして扱うことになります。また、インスタンス化した変数を「オブジェクト変数」と呼称します。この場合、オブジェクト変数のなかのクラスに属する関数と変数のことを「メンバ関数」と「メンバ変数」、または「メソッド」と「プロパティ」と呼びます。

・処理の適応、戻り値の取得

```
[戻り値] = $変数名 -> メソッド名([引数,...]);
```

・値の参照、代入

```
$変数名 -> プロパティ名 [ = 値];
```

- コンストラクタ、デストラクタ

インスタンス化の際に実行される特別なメソッド（メンバ関数）のことを、コンストラクタと呼びます。PHP4では、クラス名と同名のメソッドをコンストラクタと見なしていましたが、PHP5からは、「_construct」という統一名を採用しています。もし、クラスと同じ名前のメソッドがクラス（オブジェクト）内に存在する場合には「_construct」が優先して処理されます。コンストラクタは、インスタンス化のタイミングで実行されるという性質上、プロパティ（メンバ変数）の初期化や、クラスで使用する各種リソースの初期化といった呼び出されて最初の一回だけ処理を必要とするものを記述するのが一般的です。特に初期化処理が必要ない場合は、コンストラクタは省略可能です。

コンストラクタとは反対に、オブジェクト（インスタンス）が破棄されるタイミングで実行されるのがデストラクタです。デストラクタは PHP5以降でのみ利用可能で、統一名として「_destruct」を採用しています。デストラクタには、クラス内で使用したリソースを破棄するなど主に終了処理を記述するのが一般的です。PHP では、Java と同じくメモリの解放などは自動である程度おこなってくれるので使用頻度としてはコンストラクタより多くはみかけません。

➢ アクセス修飾子

クラスを理解するうえでもう一つ重要な概念となるのが「アクセス修飾子」です。アクセス修飾子とは、クラス内のメンバ変数（プロパティ）やメンバ関数（メソッド）に対するアクセスの可否を決めるための命令になります。

・PHP5で利用可能なアクセス修飾子

アクセス修飾子	概要
public	どこからでもアクセス可能（デフォルト）
protected	現在のクラスとサブクラスでのみアクセス可能
private	現在のクラス内部でのみアクセス可能

アクセス修飾子を省略した場合、そのメソッドとプロパティは public 扱いと見なされ、デフォルトでどこからでもアクセス可能になります。また、PHP4ではプロパティを定義するための var 命令がありましたが、こちらも public 修飾子と同様の意味であると見なされます。これら、修飾子として の役割は、クラス内部の機能をクラス外部から隠蔽することが目的で使われます。こうした機能をオブジェクト指向では「カプセル化」と言います。

➢ ゲッタメソッド、セッタメソッド

クラス内の private 変数にアクセスするためのメソッドのことを「アクセサメソッド（Accesor Method）」と総称します。また、「get_」メソッドと「set_」メソッドをそれぞれ「ゲッタメソッド（Getter Method）」、「セッタメソッド（Setter Method）」と呼ぶこともあります。

```
public function get_プロパティ名(){
    return $this -> プロパティ名;
}

public function set_プロパティ名(){
    $this -> プロパティ名 = 引数;
}
```

アクセス修飾子、「$this」は、現在のクラスを示すための命令で、クラス内のメンバにアクセスするためには、この$this を介して行う必要があります。アクセサメソッドの名前は、一般的に「get_プロパティ名」、「set_プロパティ名」とするのが慣例ですが構文規則ではないので必ずしもこの規則に従う必要はありません。get_メソッドが定義された場合にはプロパティは読み取り専用に set_メソッドが定義された場合にはプロパティは書き込み専用になります。 プロパティを隠蔽して、アクセサメソッドを介して参照、設定することで、

- 読み書きの制御が可能となる
- プロパティ値を設定する際に値の検証を行える
- プロパティ値を参照する際に値の加工を行える

などのメリットがあります。

- 静的メソッド

クラスを利用するにあたっては、必ず「インスタンス化」を行い、クラスのコピーを生成する必要があると紹介しましたが、例外的に「インスタンス化」を行わなくても利用できるメソッドがあります。このようなメソッドのことを「静的メソッド」と言います。静的メソッドを定義するには、メソッド定義の先頭に static 修飾子を付加するだけです。

・MyClass.class.php

```
<?php
    class MyClass{
        public static function triangle ($width, $height){
            return $width * $height / 2;
        }
    }
?>
```

クラス外部から静的メソッドを呼び出すには、一般的なメソッド、プロパティを呼び出す「->」演算子ではなく、「::」（スコープ）演算子を使用します。

・static.php

```
<?php
    require_once('MyClass.class.php');
    print('三角形の面積は' . MyClass::triangle(10,5) . 'です<br>');
?>
```

「::」演算子を利用することでオブジェクト変数名でなくクラス名から直接的にメソッドを呼び出すことが可能となります。ではなぜこのように、インスタンス化を必要とするメソッドとそうでないメソッドが存在するのかというと、それは、外部から与えられた値を処理し出力するだけということだけをしているからです。クラス内でプロパティとしてデータを留めることなくメソッドとしてだけ処理し呼び出し元に返すだけのいわゆる関数的な処理を求められているのでわざわざオブジェクトをつくらなくとも、データの上書き（不整合）が発生しないためです。そもそも、静的メソッドでは、クラス内にプロパティ（データ）を設定もとい参照しようにもプロパティ自体が存在しないのでできません。こうなると、$this 命令も当然ながら使えません。

- クラス内定数

クラス内定数とは、その名の通り class ブロックの中で定義された定数のことです。PHP5からは const 命令を利用することでクラス内部で定数を定義することができます。

```
<?php
    class MyClass {
        const AUTHOR = '名無し';
    }
    print('著者名 : ' . MyClass::AUTHOR);
?>
```

クラス内定数を参照するには静的メソッド同様、「::」演算子を使います。クラス内定数を利用すると、特定のクラスでしか利用しない定数をクラス内部で管理でき、クラス外部に出さなくて済むのでより分かりやすいコードを記述することができます。

- 継承

継承とは、クラスに含まれるプロパティやメソッドを引き続きながら新たな機能を追加したり、元の機能を一部だけ修正することができる機能のことです。必要に応じ元になるクラスの機能を引き継ぎつつ機能追加、変更などの新しいクラスをつくる場合に使われます。継承の際、基になるクラスのことを「スーパークラ ス（親クラス、基底クラス）」と呼び、継承の結果できたクラスのことを「サブクラス（子クラス、派生クラス）」と呼びます。

```
<?php
   class サブクラス名 extends スーパークラス名 {
      サブクラスの処理定義
   }
?>
```

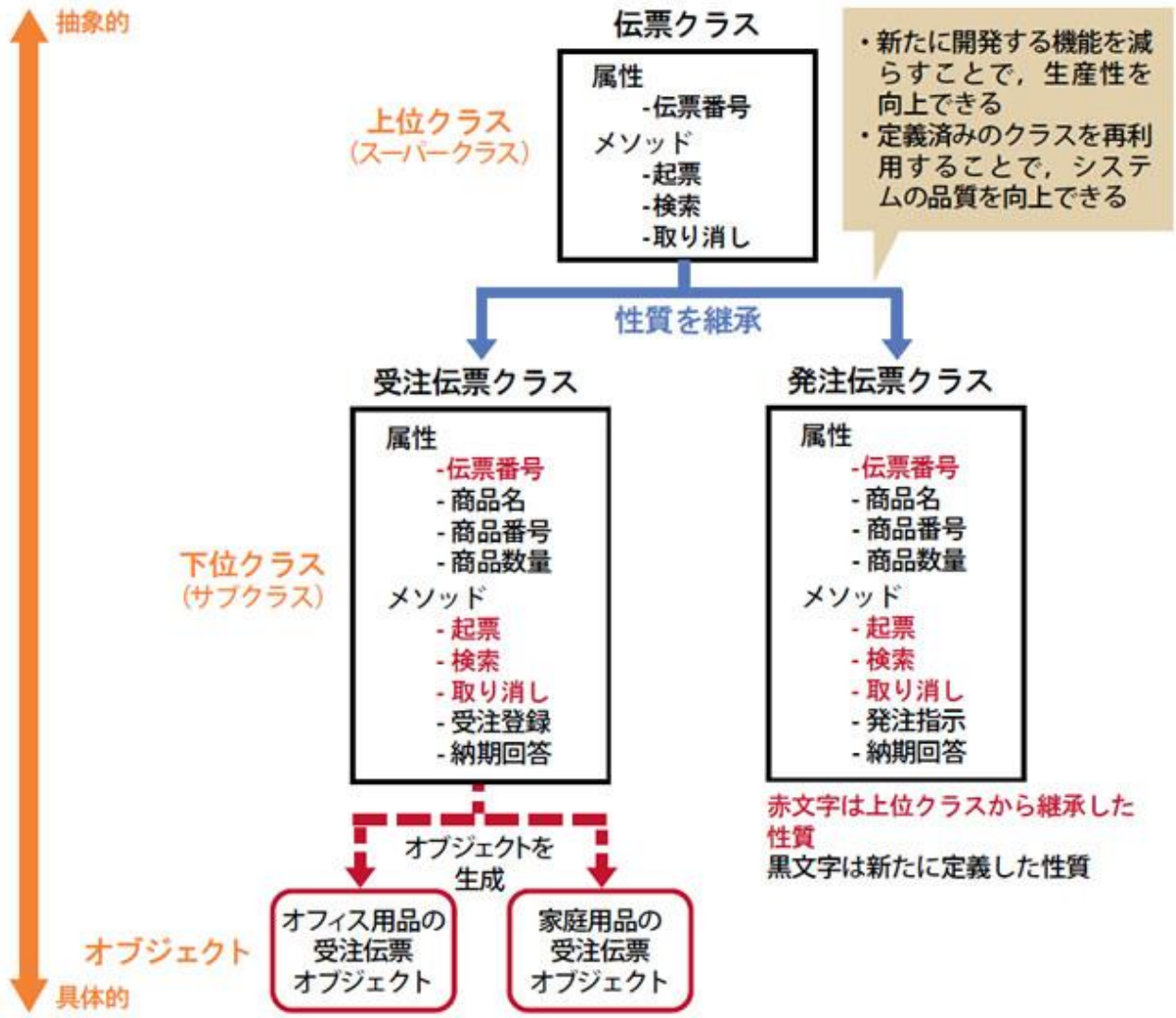
継承によりスーパークラスの機能をサブクラスで上書きすることを「オーバーライド」と言います。オーバーライドされなかったスーパークラスのメソッドはサブクラスでもそのまま引き継がれ、あたかも自分自身で定義したメソッドであるかのように利用できます。

```
<?php
   class MyClass {
      protected $data;
      public function _construct($data) {
         $this->data = $data;
      }
      public function showData() {
         return '入力値は「'.$this->data.'」です。';
      }
   }
   class MySubClass extends MyClass {
     public function showData(){
        return '***入力値は「'.$this->data.'」です。***';
     }
   }

   $obj=new MySubClass ('PEAR');
   print($obj->showData());
?>
```

[出力]
入力値は「PEAR」です。

✓ **オブジェクト指向における「クラス」と「継承」の概要**



「クラス」とはオブジェクトに共通の性質を抜き出してグループ化した、オブジェクトの「ひな型」のようなものです。継承では、上位クラス（スーパークラス）と呼ばれる親からメソッドや属性といった性質を引き継いだ下位クラス（サブクラス）と呼ばれる子供へ引き継がれることをいいます。また、子供は親の性質をもちつつ子供だけにしかない特性を持つこと（定義すること）ができます。

- parent 命令

同じくオーバーライドするにも、スーパークラスの機能を完全に塗り替えるのではなく、スーパークラスの機能を受け継ぎながらサブクラス側で機能を追加したいという場合があります。そのような場合には parent 命令でスーパークラスのメソッドを呼び出すことも可能です。parent 命令を利用するとサブクラスでスーパークラスのメソッドを少しだけ改変したいと思った場合でもそれを最初から書き直す必要はなくなります。スーパークラスのメソッドの戻り値を利用しつつ必要な部分だけを修正するということが可能になります。

```
<?php
    class MyClass {
        protected $data;
        public function _construct($data){
            $this->tdata=$data;
        }
        public function showData(){
            return '入力値は「'.$this->data.'」です。';
        }
    }
    class MySubClass extends MyClass {
        public function showData(){
            return '***'.parent::showData().'***';
        }
    }
    $obj=new MySubClass('PERA');
    print($obj->showData());
?>
```

- final 修飾子

final 修飾子は特定のメソッドをオーバーライドできないように制限するための修飾子です。継承をすることを想定していないメソッドについては final 修飾子をつけることで不用意にオーバーライドされることを防ぐことができます。

```
<?php
    class MyClass {
        protected $data;
        public function _construct($data){
            $this->data=$data;
        }
        public final function showData(){
            return '入力値は「'.$this->data.'」です。';
        }
    }
    class MySubClass extends MyClass{
        public function showData(){
            return '***入力値は「'.$this->data.'」です。***';
        }
    }
?>
```

[出力]

Fatal error: Cannot override final methot MyClass::showData() in filan.php

- ポリモーフィズム

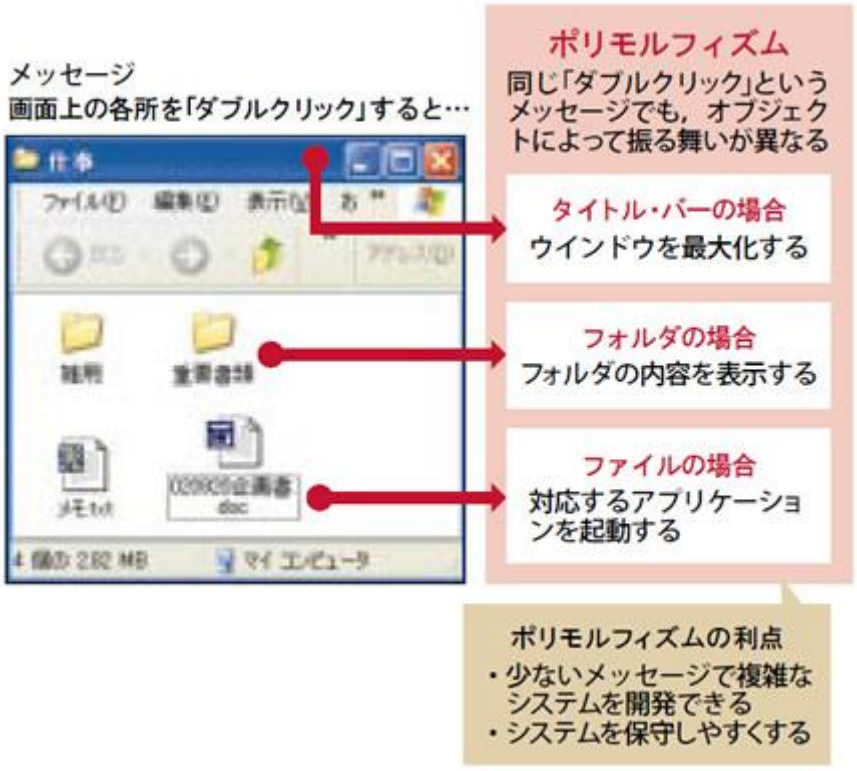
ポリモーフィズムとは同名のメソッドで異なる挙動を実現することをいいます。ポリモーフィズムのメソッドは、同じ目的の機能に同名のメソッドを割り当てる ことができるということです。これは一連の関連するクラスを利用する場合に、異なるメソッド名を覚えなくていいため、利用者にとって利便性が増します。実 装には、単なる継承とオーバーライドだけではそれぞれのサブクラスが同名のメソッドを持つことが必ずしも保証できないため、抽象メソッドという概念を PHP の場合利用します。

・polymorphism.php

```
<?php
class Figure {
     protected $width;
     protected $height;
     public function _construct($width, $height){
        $this->width=$width;
        $this->height=$height;
    }
    protected function getArea(){}
}
class Triangle extends Figure {
    public final function getArea(){
        return $this->width * $this->height / 2;
    }
}
class Square extends Figure {
    public final function getArea(){
        return $this->width * $this->height;
    }
}
$tri = new Triangle(10, 5);
$sqr = new Square(10, 5);
print('三角形の面積は'.$tri->getArea().'です。<br>');
print('四角形の面積は'.$sqr->getArea().'です。<br>');
?>
```

[出力]
三角形の面積は 25 です。
四角形の面積は 50 です。

✓ **「ポリモルフィズム」の例**



オブジェクト指向では、同じメッセージに対してオブジェクトごとに異なる処理を実行させることができます。これを「ポリモルフィズム」と呼びます。ポリモルフィズムを活用すれば、簡潔な構造で多様な処理を実行するシステムを開発できます。

ここまで読み進んできて、オブジェクト指向に対する理解が深まったと思います。その上で、もう 1 つ覚えてほしいポイントとして、オブジェクト指向のシステム開発を実践する際には、「オブジェクト指向を強く意識する必要がある」ということです。

純粋なオブジェクト指向開発は今も難しいのが現状です。オブジェクト指向言語や GUI を使ったとしても、結局コンピュータの内部は手続き指向に則っているためです。オブジェクト指向に“違反”しようと思えば、いくらでもできてしまいます。例えば，システムの処理性能を優先して、プログラムからいきなりメモリー・アドレスを指定して直接アクセスすることも可能です。データベースやミドルウエアなど、システム構築に欠かせない IT 関連製品の中には、オブジェクト指向とは異なる旧来の構造を持つものも依然として多いです。オブジェクト指向開発を成功に導くカギは、知識や技術の習得に加えて“強い意志”を持つことも必要といえます。

- 抽象メソッド

抽象メソッドとは、メソッド定義の先頭に abstract 修飾子を付加したメソッドであり、抽象メソッド自体は動作を定義しない空のメソッドです。抽象メソッドは、サブクラスで必ずオーバーライドしなければいけません。クラス内の抽象メソッドがサブクラスで必ずオーバーライドされていなければそのクラスはインスタンス化できません。これにより、同名のメソッドがサブクラスで再定義されることを保証します。抽象メソッドの定義では、中身が空であっても { } のようなブロックを定義しません。

また、抽象メソッドを含むクラスのことを抽象クラスといい、class キーワードの前に abstract 修飾子を付加する必要があります。抽象クラスのサブクラスが抽象メソッドをすべてオーバーライドしていない場合、エラーが出ます。

・abstract.php

```
<?php
abstract class Figure {
    protected $width;
    protected $height;

    public function _construct($width, $height){
        $this->width = $width;
        $this->height = $height;
    }
    protected abstract function getArea();
}
class Triangle extends Figure {
    public final function getArea() {
        return $this->width * $this->height / 2;
    }
}
class Square extends Figure {
    public final function getArea() {
        return $this->width * $this->height;
    }
}
$tri = new Triangle(10, 5);
$sqr = new Square(10, 5);
print('三角形の面積は'.$tri->getArea().'です。<br>');
print('四角形の面積は'.$sqr->getArea().'です。<br>');
?>
```

- インターフェイス

抽象クラスによるポリモーフィズムの実現には問題もあります。それは、PHP では多重継承、つまり、サブクラスが同時に複数のスーパークラスを継承できないという点です。この性質を単一継承といいます。多重継承ができないということはポリモーフィズムを実現したいすべての機能（メソッド）をひとつの抽象クラスに含めなければいけないことを意味します。これは必ずしもサブクラスがスーパークラスの機能を必要としない場合でも、機能をオーバーライドしなければ ならなくなり、当然、コードは冗長になります。コードが冗長になるということはサブクラスの役割が分かりにくくする一因をつくることにつながります。そこで、PHP では、インターフェイスを利用します。インターフェイスとは配下のメソッドがすべて抽象メソッドである特別なクラスを言います。インターフェイ スを定義する場合には、class キーワードの代わりに interfece キーワードを使用します。また、配下のメソッドが抽象メソッドであることは interfece キーワードから明らかなので個々のメソッドには abstract 修飾子は必要ありません。

インターフェイスとクラスとの決定的な違いはいわゆる多重継承が可能になるということです。インターフェイスを採用することにより、サブクラス側では必要なメソッドを含むインターフェイス を選択し継承することが可能となります。このとき、インターフェイスの機能を受け継ぐことは「継承する」ではなく「実装する」と言います。インターフェイ スを実装する場合には、extends キーワードの代わりに implements キーワードを使用します。インターフェイスを実装したクラスのことを「実装クラス」といいます。また、複数のインターフェイスを実装する必要がある場合には、インターフェイス名をカンマ区切りで記述します。

※ PHP または、Java では言語仕様的に多重継承が許されていません。これは、メソッド名が重複した場合の処置など重複チェックが大変なことや、多重継承が様々な問題を引き起こしやすいからといわれています（C++では仕様的には認められています）。ただ、多重継承をサポートすることにより言語の構造が複雑になりすぎることや、本当に多重継承を使わざる得ない場合はそれほど多くありません。Java でも多重継承ができない代わりにインターフェイスが使えるようになっています。多重継承、インターフェイスともにそれ相応の高レベルのプログラム制作に使用される技術なので、通常の業務ではあまり使用されることはありません。ただ、こういった考え方もオブジェクト指向という分野に入るので概要だけでも頭に入れといてください。

・Figure.class.php

```
<?php
interface Figure {
    public function getArea();
}
?>
```

・interface.php

```
<?php
require_once('Figure.class.php');
class Triangle implements Figure {
    private $width;
    private $height;
    public function _construct($width, $height){
        $this->width = $width;
        $this->height = $height;
    }
    public final function getArea(){
        return $this->width * $this->height / 2;
    }
}
class Square implements Figure {
    private $width;
    private $height;
    public function _construct($width, $height){
        $this->width = $width;
        $this->height = $height;
    }
    public final function getArea(){
        return $this->width * $this->height;
    }
}
$tri = new Triangle(10, 5);
$sqr = new Square(10, 5);
print('三角形の面積は'.$tri->getArea().'です。<br>');
print('四角形の面積は'.$sqr->getArea().'です。<br>');
?>
```

3.PHP5独自のオブジェクト構文規則

- _autoload 関数

PHP5では、オブジェクト指向構文が強化され、限りなく JAVA ライクなオブジェクト指向プログラミングが可能になりました。さらに、PHP5では独自で使える構文規則も追加されています。そのひとつが autoload 関数です。もともとクラスは複数のスクリプトから再利用されるというその性質上、1クラス1ファイルで管理することが望ましいとされています（ただ、短いプログラムなどは可読性を考慮して設計されていることもあるので例外はあります）。1 クラス1ファイルなら、無駄なクラスを読み込む必要もありませんし、ファイル管理という観点からも整理しやすくなります。ただ、1クラス1ファイル方式で 作成していくと扱うクラスが増えてきた場合にひとつひとつのクラスファイルについて include_once 関数や require_once 関数を記述す ることが大きな手間になりますし、記述漏れや誤りの原因にもなります。そこで役に立つのが_autoload 関数です。

_autoload 関数は、未定義クラスを呼び出したタイミングで自動的に呼び出される特別な関数であり、呼び出した未定義のクラスの名前が引数として渡されます。_autoload 関数は、それ自体はなんら実装を持たない名前だけ予約された関数です。一般的には、「クラス名.class.php」のような形式であらかじめファイル名に一定の規則性を設けた上で、_autoload 関数内で require_once 関数を呼び出し、クラスが呼び出さ れたタイミングで自動的に同名のクラスファイルをインクルードするような用途で使用します。

_autoload 関数を利用すると、クラスごとに require_once 関数を呼び出す必要がなくなるので、コードがシンプルになります。また、スクリプト内で使用するかどうか分からないクラスをインクルードしなくても済むので処理上のオーバーヘッドが軽減します。なお、_autoload 関数を定義したスクリプトは、その性質上、アプリケーション内のすべてのスクリプトで有効にしておくことをお勧めします。アプリケーション配下のファイルに対して autoload.php を自動的に読み込むためには、.htaccess ファイルを利用します。

・autoload.php

```
<?php
function _autoload($name){
    require_once($name . '.class.php');
}
?>
```

・auto_load.php

```
<?php
require_once('autoload.php');
$obj = new MyClass();
$obj -> width = 10;
$obj -> height = 5;
print('三角形の面積は'.$obj->triangle().'です。<br>');
?>
```

- _call メソッド

_call メソッドは、未定義のメソッドが呼び出された場合の挙動を定義するためのメソッドです。_autoload 関数と同様 PHP5独自のもので あり、それ自体はなんら実装を持たない名前だけ予約されたメソッドですので必要に応じて中身を実装する必要があります。_call メソッドは引数として、メソッド名と引数値の配列を受け取ることができます。変わり種の機能ではありますが、メソッド名に応じて共通的な処理を分岐したいなどのケースで利用できるでしょう。

```
<?php
class MyClass{
    public function _call($name,$args){
        print('メソッド名 : '.$name.'<br>');
        print_r($args);
    }
}
$obj = new MyClass();
$obj -> nothing('x','y');
?>
```

[出力]
メソッド名 : nothing
Array ([0] => x [1] => y)

- _get,_set メソッド

_get,_set メソッドは、未定義のプロパティを取得・設定しようとしたタイミングで呼び出されるメソッドです。_autoload 関数と同様 PHP5独自のものであり、それ自体はなんら実装を持たない名前だけ予約されたメソッドですので必要に応じて中身を実装する必要があります。_get メ ソッドは、取得しようとした未定義のプロパティの名前を引数として受け取ります。_set メソッドは、設定しようとした未定義のプロパティの名前と値を引数として受け取ります。これらのメソッドを使用して、未定義のプロパティが参照されたときの処理を記述します。

```
<?php
class MyClass{
    public function _get($name){
        print($name . 'プロパティが参照されました。<br>');
    }
    public function _set($name, $value){
        print($name . 'プロパティ値' . $value . 'がセットされました。<br>');
    }
}
$obj = new MyClass();
$obj -> nobody = 1;
print($obj->nobody);
?>
```

[出力]
nobody プロパティ値 1 がセットされました。
nobody プロパティが参照されました。

4.Apache との連携

- .htaccess

.htaccess ファイル（先頭にドットがつく。UNIX 系列では先頭にドットがつくファイルは隠しファイル属性の意味になる）は、Apache 標準の設定ファイルで、ドキュメントルート配下の任意のディレクトリに配置することで、該当するディレクトリと、その配下のサブディレクトリに対してだけ適用されるパラメータを設定することができます。php.ini を利用するとサーバー全体に対して設定が適用されてしまいますが、このように特定のアプリケーションに対して設定を適用したいという場合は、.htaccess を利用します。書式は以下のとおりです。

```
<IfModule mod_php5.c>
   php_value パラメータ名 パラメータ値
   php_flag パラメータ名 on | off
<IfModule>
```

パラメータ値が on または off で表される場合には、php_flag を、それ以外の値である場合には php_value を使用します。 php_value,php_flag は必要な数だけ記述できます。また、.htaccess で PHP の設定を行う場合には、次の２点について注意する必要があります。

- http.conf の AllowOverride ディレクティブを"All"に設定

AllowOverride ディレクティブは、.htaccess ファイルによるパラメータの上書きを可能にするかどうかを決めるためのパラメータです。 AllowOverride ディレクティブが"None"に設定されている場合は.htaccess ファイルは無視されます。

- .htaccess ファイルで設定可能なパラメータは決まっている

.htaccess ファイルでは、すべての php.ini のパラメータを設定できるわけではありません。パラメータによっては、php.ini でしか設定できないものがあるので注意してください。どのパラメータが.htaccess ファイルで設定可能かは、ini_get_all 関数の access 情報を参照することで確認できます。アクセスレベルが6または7である場合、そのパラメータは.htaccess で設定可能です。

```
<?
print '<PRE>';
print_r(ini_get_all());
print '</PRE>';
?>

[出力]
Array
(
   [allow_call_time_pass_reference] => Array
      (
         [global_value] => 1
         [local_value] => 1
         [access] => 6
      )
   中略
)
```