

01-01. アクセス修飾子

◆ public

どのオブジェクトでも呼び出せる。

◆ protected

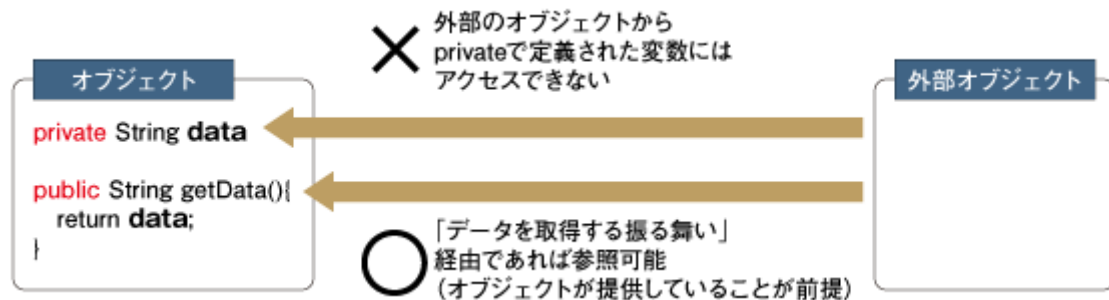
同じクラス内と、その親クラスまたは子クラスでのみ呼び出せる。

◆ private

同じオブジェクト内でのみ呼び出せる。

• Encapsulation (カプセル化)

カプセル化とは、システムの実装方法を外部から隠すこと。オブジェクト内のプロパティにアクセスするには、直接データを扱う事はできず、オブジェクト内のメソッドをコールし、アクセスしなければならない。



◆ static

別ファイルでのメソッドの呼び出しにはインスタンス化が必要である。しかし、static修飾子をつけることで、インスタンス化しなくともコールできる。プロパティ値は用いず、引数の値のみを用いて処理を行うメソッドに対して用いる。

【実装例】

// インスタンスを作成する集約メソッドは、プロパティ値にアクセスしないため、常に同一の処理を行う。

```
public static function aggregateDogToyEntity(Array $fetchData)
{
    return new DogToyEntity
    (
        new ColorVO($fetchData['dog_toy_type']),
        $fetchData['dog_toy_name'],
        $fetchData['number'],
        new PriceVO($fetchData['dog_toy_price']),
        new ColorVO($fetchData['color_value'])
    );
}
```

【実装例】

// 受け取ったOrderエンティティから値を取り出すだけで、プロパティ値は呼び出していない。

```
public static function computeExampleFee(Entity $order): Money
{
    $money = new Money($order->exampleFee);
    return $money;
}
```

01-02. メソッド

◆ メソッドの実装手順

1. その会社のシステムで使われているライブラリ
2. PHPのデフォルト関数（引用：PHP関数リファレンス, <https://www.php.net/manual/ja/function-f.php>）
3. 新しいライブラリ

◆ 値を取得するアクセサメソッドの実装

Getterでは、プロパティを取得するだけではなく、何かしらの処理を加えたうえで取得すること。

【実装例】

- Getter

```
class ABC {

    private $property;

    public function getEditProperty()
    {
        // 単なるGetterではなく、例外処理も加える。
        if(!isset($this->property)){
            throw new Exception('プロパティに値がセットされていません。')
        }
        return $this->property;
    }
}
```

```
}  
  
}
```

◆ 値を設定するアクセサメソッドの実装

- Setter

『Mutable』なオブジェクトを実現できる。

【実装例】

```
class Test01 {  
  
    private $property01;  
  
    // Setterで$property01に値を設定  
    public function setProperty($property01)  
    {  
        $this->property01 = $property01;  
    }  
  
}
```

- マジックメソッドの `__construct()`

Setterを持たせずに、`__construct()` だけを持たせれば、ValueObjectのような、『Immutable』なオブジェクトを実現できる。

【実装例】

```
class Test02 {  
  
    private $property02;  
  
    // コンストラクタで$property02に値を設定  
    public function __construct($property02)  
    {  
        $this->property02 = $property02;  
    }  
  
}
```

- 『Mutable』と『Immutable』を実現できる理由

Test01クラスインスタンスの `$property01` に値を設定するためには、インスタンスからSetterをコールする。Setterは何度でも呼び出せ、その度にプロパティの値を上書きできる。

```
$test01 = new Test01  
  
$test01->setProperty01("プロパティ01の値")  
  
$test01->setProperty01("新しいプロパティ01の値")
```

一方で、Test02クラスインスタンスの \$property02 に値を設定するためには、インスタンスを作り直さなければならない。つまり、以前に作ったインスタンスの \$property02 の値は上書きできない。Setterを持たせずに、`__construct()` だけを持たせれば、『Immutable』なオブジェクトとなる。

```
$test02 = new Test02("プロパティ02の値")

$test02 = new Test02("新しいプロパティ02の値")
```

◆ メソッドチェーン

以下のような、オブジェクトAを最外層とした関係が存在しているとする。

【オブジェクトA（オブジェクトBをプロパティに持つ）】

```
class Obj_A{
    private $objB;

    // 戻り値のデータ型を指定
    public function getObjB(): ObjB
    {
        return $this->objB;
    }
}
```

【オブジェクトB（オブジェクトCをプロパティに持つ）】

```
class Obj_B{
    private $objC;

    // 戻り値のデータ型を指定
    public function getObjC(): ObjC
    {
        return $this->objC;
    }
}
```

【オブジェクトC（オブジェクトDをプロパティに持つ）】

```
class Obj_C{
    private $objD;

    // 戻り値のデータ型を指定
    public function getObjD(): ObjD
    {
        return $this->objD;
    }
}
```

以下のように、戻り値のオブジェクトを用いて、より深い層に連続してアクセスしていく場合...

```
$ObjA = new Obj_A;

$ObjB = $ObjA->getObjB();

$ObjC = $B->getObjB();

$ObjD = $C->getObjD();
```

以下のように、メソッドチェーンという書き方が可能。

```
$D = getObjB()->getObjC()->getObjC();

// $D には ObjD が格納されている。
```

◆ マジックメソッド（Getter系）

オブジェクトに対して特定の操作が行われた時に自動的にコールされる特殊なメソッドのこと。自動的に呼び出される仕組みは謎。共通の処理を行うGetter（例えば、値を取得するだけのGetterなど）を無闇に増やしたくない場合に用いることで、コード量の肥大化を防ぐことができる。PHPには最初からマジックメソッドは組み込まれているが、自身で実装した場合、オーバーライドされてコールされる。

- `__get()`

定義されていないプロパティや、アクセス権のないプロパティを取得しようとした時に、代わりに呼び出される。メソッドは定義しているが、プロパティは定義していないような状況で用いる。

```
class Example
{

    private $example = [];

    // 引数と返り値のデータ型を指定
    public function __get(String $name): String
    {
        echo "{$name}プロパティは存在しないため、プロパティ値を取得できません。"
    }

}
```

```
// 存在しないプロパティを取得。
$example = new Example();
$example->hoge;

// 結果
hogeプロパティは存在しないため、値を呼び出せません。
```

- `__call()`

定義されていないメソッドや、アクセス権のないメソッドを取得しようとした時に、代わりにコールされる。プロパティは定義しているが、メソッドは定義していないような状況で用いる。

- `__callStatic()`

◆ マジックメソッド（Setter系）

定義されていないstaticメソッドや、アクセス権のないstaticメソッドを取得しようとした時に、代わりに呼び出される。自動的にコールされる仕組みは謎。共通の処理を行うSetter（例えば、値を設定するだけのSetterなど）を無闇に増やしたくない場合に用いることで、コード量の肥大化を防ぐことができる。PHPには最初からマジックメソッドは組み込まれているが、自身で実装した場合、オーバーライドされて呼び出される。

- `__set()`

定義されていないプロパティや、アクセス権のないプロパティに値を設定しようとした時に、代わりにコールされる。オブジェクトの不変性を実現するために使用される。（詳しくは、ドメイン駆動設計のノートを参照せよ）

```
class Example
{
    private $example = [];

    // 引数と戻り値のデータ型を指定
    public function __set(String $name, String $value): String
    {
        echo "{$name}プロパティは存在しないため、{$value}を設定できません。"
    }
}
```

```
// 存在しないプロパティに値をセット。
$example = new Example();
$example->hoge = "HOGE";

// 結果
hogeプロパティは存在しないため、HOGEを設定できません。
```

- マジックメソッドの `__construct()`

インスタンス化時に自動的に呼び出されるメソッド。インスタンス化時に実行したい処理を記述できる。Setterを持たせずに、`__construct()`でのみ値の設定を行えば、ValueObjectのような、『Immutable』なオブジェクトを実現できる。

【実装例】

```
class Test02 {
    private $property02;

    // コンストラクタで$property02に値を設定
    public function __construct($property02)
    {
        $this->property02 = $property02;
    }
}
```

- 【『Mutable』と『Immutable』を実現できる理由】

Test01クラスインスタンスの `$property01` に値を設定するためには、インスタンスからSetterをコールする。Setterは何度でもコールでき、その度にプロパティの値を上書きできる。

```
$test01 = new Test01

$test01->setProperty01("プロパティ01の値")
PHP
$test01->setProperty01("新しいプロパティ01の値")
```

一方で、Test02クラスインスタンスの `$property02` に値を設定するためには、インスタンスを作り直さなければならない。つまり、以前に作ったインスタンスの `$property02` の値は上書きできない。Setterを持たせずに、`__construct()` だけを持たせれば、『Immutable』なオブジェクトとなる。

```
$test02 = new Test02("プロパティ02の値")

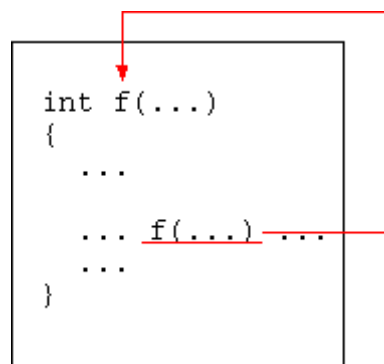
$test02 = new Test02("新しいプロパティ02の値")
```

◆ Recursive call : 再帰的プログラム

自プログラムから自身自身をコールし、実行できるプログラムのこと。

【具体例】

ある関数 `f` の定義の中に `f` 自身を呼び出している箇所がある。



```
int f(...)
{
    ...
    ... f(...) ...
    ...
}
```

【実装例】

クイックソートのアルゴリズム（※詳しくは、別ノートを参照）

1. 適当な値を基準値（Pivot）とする（※できれば中央値が望ましい）
2. Pivotより小さい数を前方、大きい数を後方に分割する。
3. 二分割された各々のデータを、それぞれソートする。
4. ソートを繰り返し実行する。

```
function quickSort(Array $array): Array
{
    // 配列の要素数が一つしかない場合、クイックソートする必要がないので、返却する。
    if (count($array) <= 1) {
        return $array;
    }
}
```

```
// 一番最初の値をPivotとする。
$pivot = array_shift($array);

// グループを定義
$left = $right = [];

foreach ($array as $value) {

    if ($value < $pivot) {

        // Pivotより小さい数は左グループに格納
        $left[] = $value;

    } else {

        // Pivotより大きい数は右グループに格納
        $right[] = $value;

    }

}

// 処理の周回ごとに、結果の配列を結合。
return array_merge
(
    // 左のグループを再帰的にクイックソート。
    quickSort($left),

    // Pivotを結果に組み込む。
    array($pivot),

    // 右のグループを再帰的にクイックソート。
    quickSort($right)
);
}
```

```
// 実際に使ってみる。
$array = array(6, 4, 3, 7, 8, 5, 2, 9, 1);
$result = quickSort($array);
var_dump($result);

// 昇順に並び替えられている。
1, 2, 3, 4, 5, 6, 7, 8
```

◆ 高階関数とClosure（無名関数）

関数を引数として受け取ったり、関数自体を返したりする関数を『高階関数』と呼ぶ。

- 無名関数を用いない場合

【実装例】

```
## 第一引数のみの場合
```



```
// 高階関数を定義
function test($callback)
{
    echo $callback();
}

// コールバックを定義
// 関数の中でコールされるため、「後で呼び出される」という意味合いから、コールバック関数といえる。
function callbackMethod() : String
{
    return "出力成功";
}

// 高階関数の引数として、コールバック関数を渡す
test("callbackMethod");

// 出力結果
出力成功
```

第一引数と第二引数の場合

```
// 高階関数を定義
public function higher-order($param, $callback)
{
    return $callback($param);
}

// コールバック関数を定義
public function callbackMethod($param)
{
    return $param."の出力成功";
}

// 高階関数の第一引数にコールバック関数の引数、第二引数にコールバック関数を渡す
higher-order("第一引数", "callbackMethod");

// 出力結果
第一引数の出力成功
```

- 無名関数を用いる場合

【実装例】

```
// 高階関数のように、関数を引数として渡す。
public function higher-order($param, $callback)
{
    $parentVar = "&親メソッドのスコープの変数"
    return $callback($param)
}

// 第二引数の無名関数。関数の中でコールされるため、「後でコールされる」という意味合いから、コールバック関数といえる。
// コールバック関数は再利用されないため、名前をつけずに無名関数とすることが多い。
// 親メソッドのスコープの変数を引数として用いることができる。
higher-order(第一引数,
    function($param) use($parentVar)
```

```

        {
            return $param.$parentVar."の出力成功";
        }
    )

// 出力結果
第一引数&親メソッドのスコープの変数の出力成功

```

◆ Closure（無名関数）と即時関数

定義したその場で実行される無名関数を『即時関数』と呼ぶ。

```

$item = new Item;
$item->getOption()->setName('オプションA')

// 即時関数を定義
// 無名関数の引数に、親メソッドのスコープの$itemを渡す。
$optionName = call_user_func(function() use($item){
    $item->hasOption()
    ? $item->getOption()->name()
    : '';
});

// 出力結果
echo $optionName //オプションA

```

01-03. 外部ファイルの読み込みとコール

◆ use文による読み込みとコール

PHPでは、use文によって、外部ファイルの名前空間、クラス、メソッド、定数を読み込み、コールできる。ただし、静的に読み込むことに注意。しかし、チームの各エンジニアが好きな物を読み込んでいたらスパゲッティコードになりかねない。そこで、チームでの開発では、記述ルールを設けて、use文で読み込んでよいものを制限するとよい。

【以下で読み込まれるクラスの実装例】

```

// 名前空間を定義。
namespace Domain\Entity1;

// 定数を定義。
const VALUE = "これは定数です。"

class E1
{
    public function className()
    {
        return "example1メソッドです。";
    }
}

```

- 名前空間の読み込み

```
// use文で名前空間を読み込む。
use Domain/Entity2

namespace Domain/Entity2;

class E2
{
    // 名前空間を読み込み、クラスまで辿り、インスタンス作成。
    $e1 = new Entity1/E1;
    echo $e1;
}
```

- クラスの読み込み

```
// use文でクラス名を読み込む。
use Domain/Entity1/E1;

namespace Domain/Entity2;

class E2
{
    // 名前空間を読み込み、クラスまで辿り、インスタンス作成。
    $e1 = new E1;
    echo $e1;
}
```

- メソッドの読み込み

```
// use文でメソッドを読み込む。
use Domain/Entity1/E1/className;

namespace Domain/Entity2;

class E2
{
    // E1クラスのclassName()をコール。
    echo className();
}
```

- 定数の読み込み

```
// use文で定数を読み込む。
use Domain/Entity1/E1/VALUE;

namespace Domain/Entity2;

class E2
{
    // E1クラスの定数を出力。
    echo VALUE;
}
```

◆ 親クラスの静的メソッドのコール

```
abstract class Example
{
    public function example()
    {
        // 処理内容;
    }
}
```

```
class SubExample
{
    public function subExample()
    {
        // 親メソッドの静的メソッドをコール
        $example = parent::example();
    }
}
```

02-01. データ型

プログラムを書く際にはどのような処理を行うのかを事前に考え、その処理にとって最適なデータ構造で記述する必要がある。そのためにも、それぞれのデータ構造の特徴（長所、短所）を知っておくことが重要である。

◆ Array型

『内部ポインタ』とは、配列において、参照したい要素を位置で指定するためのカーソルのこと。

● 内部ポインタを用いた配列要素の出力

```
$array = array("あ", "い", "う");

// 内部ポインタが現在指定している要素を出力。
echo current($array); // あ

// 内部ポインタを一つ進め、要素を出力。
echo next($array); // い

// 内部ポインタを一つ戻し、要素を出力。
echo prev($array); // あ

// 内部ポインタを最後まで進め、要素を出力。
echo end($array); // う

// 内部ポインタを最初まで戻し、要素を出力。
echo reset($array); // あ
```

● 多次元配列

中に配列をもつ配列のこと。配列の入れ子構造が2段の場合、『二次元配列』と呼ぶ。

```

Array
(
    [0] => Array
        (
            [0] => リンゴ
            [1] => イチゴ
            [2] => トマト
        )

    [1] => Array
        (
            [0] => メロン
            [1] => キュウリ
            [2] => ピーマン
        )
)

```

• 連想配列

中に配列をもち、キーに名前がついている（赤、緑、黄、果物、野菜）ような配列のこと。下の例は、二次元配列かつ連想配列である。

```

Array
(
    [赤] => Array
        (
            [果物] => リンゴ
            [果物] => イチゴ
            [野菜] => トマト
        )

    [緑] => Array
        (
            [果物] => メロン
            [野菜] => キュウリ
            [野菜] => ピーマン
        )
)

```

◆ List型

配列の要素一つ一つを変数に格納したい場合、List型を使わなければ、以下のように実装する必要がある。

```

$array = array("あ", "い", "う");
$a = $array[0];
$i = $array[1];
$u = $array[2];

echo $a.$i.$u; // あいう

```

しかし、以下の様にList型を使うことによって、複数の変数への格納を一行で実装することができる。

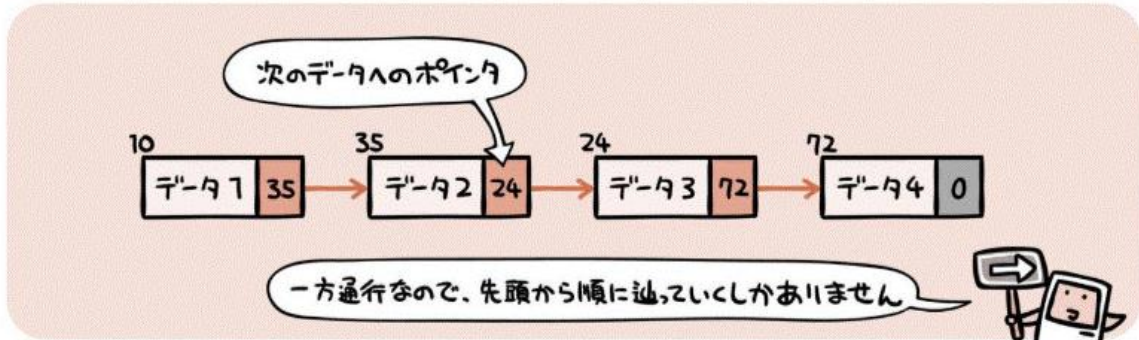
```
list($a, $i, $u) = array("あ", "い", "う")
```

```
echo $a.$i.$u; // あいう
```

- 単方向List

☞ 単方向リスト

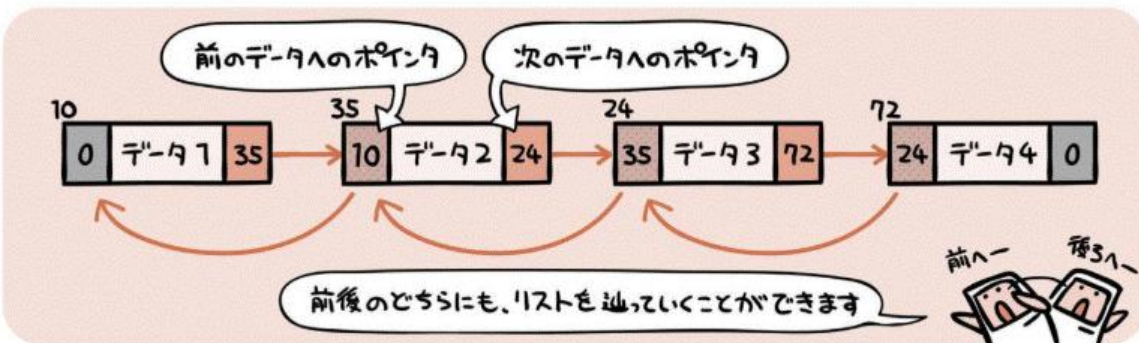
次のデータへのポインタを持つリストです。左ページの説明でも用いているようにリストといえばこれ。一番基本的な構造です。



- 双方向List

☜☞ 双方向リスト

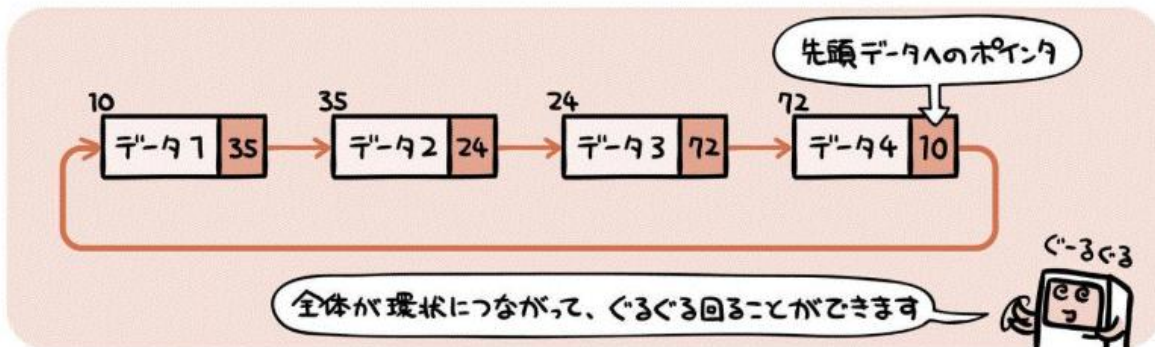
次のデータへのポインタと、前のデータへのポインタを持つリストです。



- 循環List

☞ 循環リスト

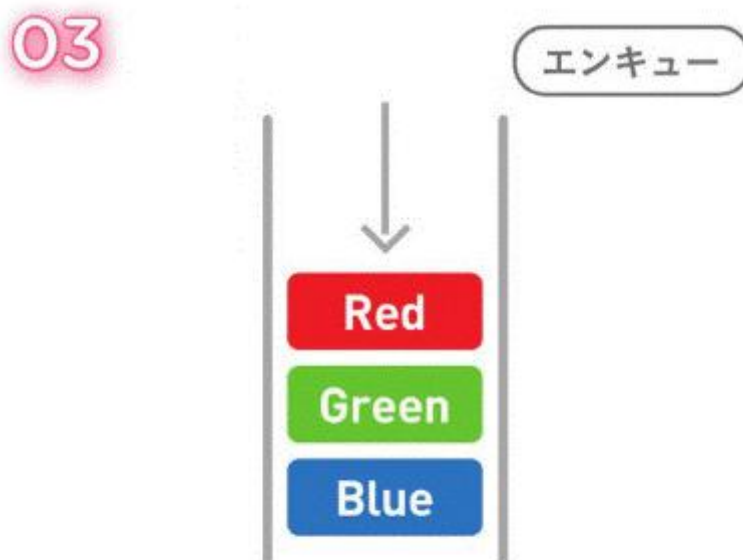
次のデータへのポインタを持つリスト。ただし、最後尾データは、先頭データへのポインタを持ちます。



◆ Object型

```
Fruit Object  
(  
  [id:private] => 1  
  [name:private] => リンゴ  
  [price:private] => 100  
)
```

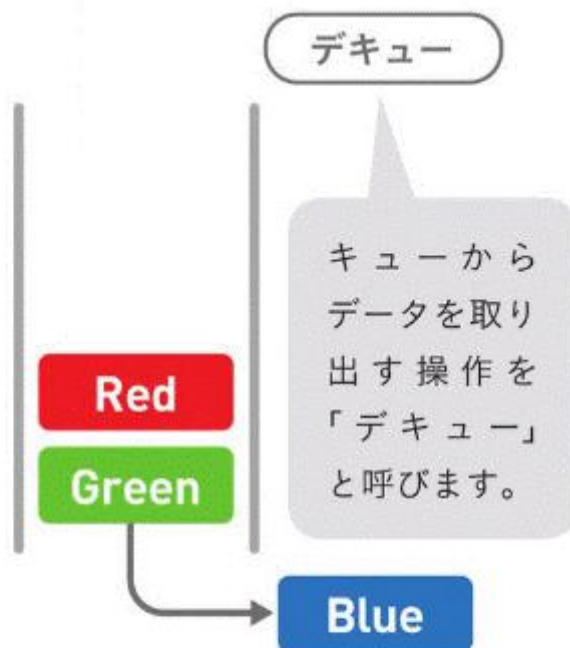
◆ Queue型



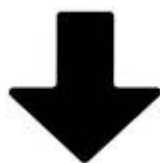
さらに「Red」というデータをエンキューしました。



04

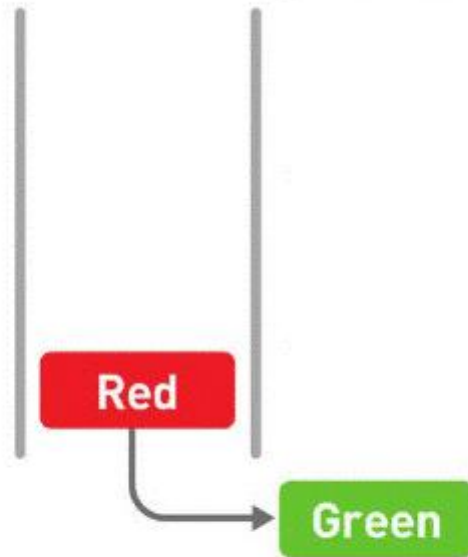


キューからデータを取り出す場合、一番下、すなわち最も古くに追加されたデータから取り出されます。この場合は「Blue」が取り出されます。



05

デキュー



もう一度デキュー操作をすると、今度は「Green」が取り出されます。

PHPでは、`array_push()` と `array_shift()` を組み合わせることで実装できる。

```
$array = array("Blue", "Green");

// 引数を、配列の最後に、要素として追加する。
array_push($array, "Red");
print_r($array);

// 出力結果
Array
(
    [0] => Blue
    [1] => Green
    [2] => Red
)

// 配列の最初の要素を取り出す。
$theFirst= array_shift($array);
print_r($array);

// 出力結果
Array
(
    [0] => Green
    [1] => Red
)

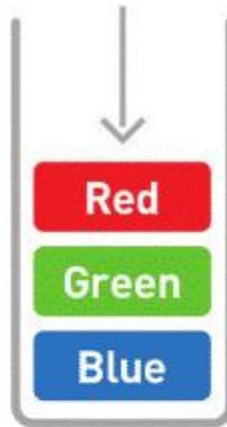
// 取り出された値の確認
echo $theFirst // Blue
```

◆ Stack型

PHPでは、`array_push()` と `array_pop()` で実装可能。

03

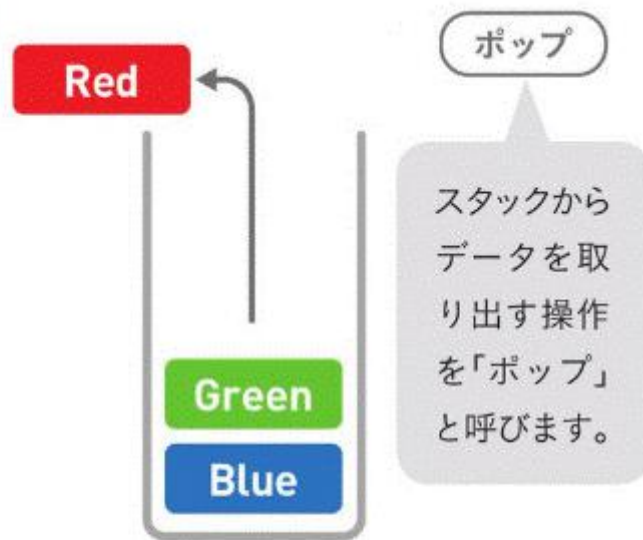
プッシュ



さらに「Red」というデータをプッシュしました。



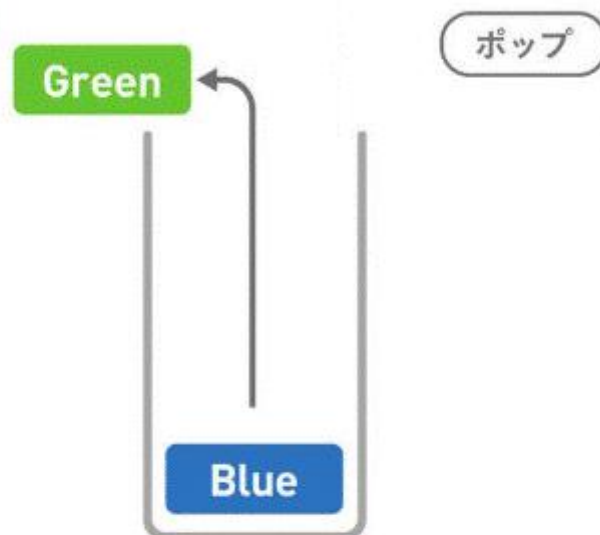
04



スタックからデータを取り出す場合、一番上、すなわち最も新しく追加されたデータから取り出されます。この場合は「Red」が取り出されます。



05



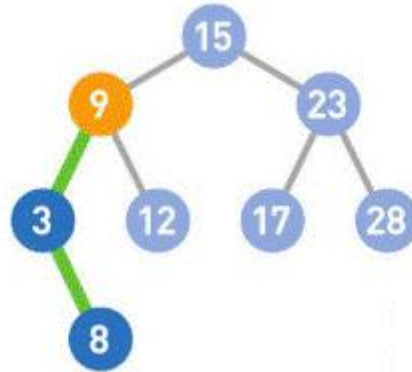
もう一度ポップ操作をすると、今度は「Green」が取り出されます。

◆ ツリー構造

- 二分探索木

各ノードにデータが格納されている。

02

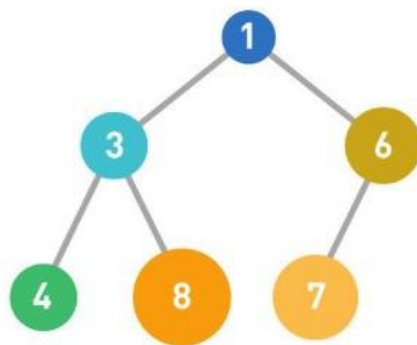


2分探索木には2つの性質があります。1つ目の性質として、すべてのノードは、そのノードの左部分木に含まれるどの数よりも大きくなります。例えば、ノード9はその左部分木のどの数よりも大きいです。

- ヒープ

Priority Queueを実現するときに用いられる。各ノードにデータが格納されている。

02

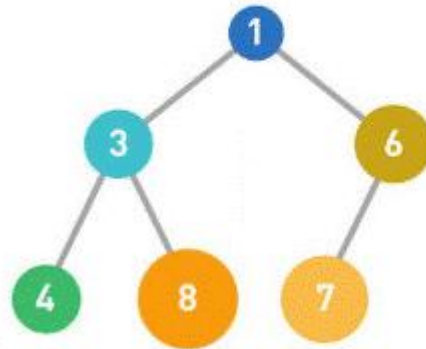


また、ヒープにおいてデータを格納する場所のルールとして、子の数字は必ず親の数字より大きくなっています。したがって、一番上（根）に最小の数字が入っています。データを追加するときは、ヒープの形のルールを守るために、一番下の段に左詰めします。一番下の段がすべて埋まっている場合は新しい段ができ、その一番左に追加されます。



03

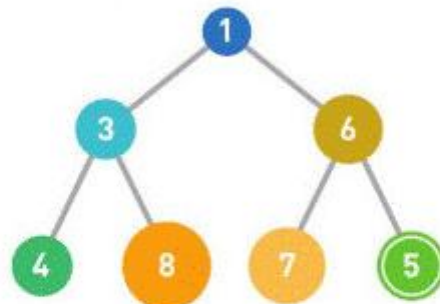
5



ヒープに数 (5) を追加してみましょう。

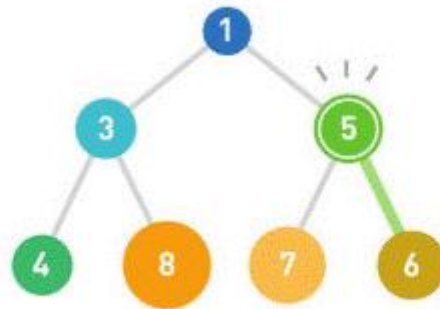


04



追加された数は、まず 02 で説明した位置に設置されます。この場合は一番下の段に空きが1つありましたから、そこに設置されることになります。





親6 > 子5であるため、数字を入れ替えました。
この操作を、入れ替えが発生しなくなるまで繰り返します。

02-02. 変数

◆ スーパーグローバル変数

スコープに関係なく、どのプログラムからでもアクセスできる連想配列変数

SGLOBALS (グローバル変数)	<ul style="list-style-type: none"> ・グローバルスコープで使用可能なすべての変数への参照 ・連想配列として使用
S_SERVER (サーバー変数)	<ul style="list-style-type: none"> ・サーバ情報および実行時の環境情報が格納される ・連想配列として使用
S_GET (ゲット変数)	<ul style="list-style-type: none"> ・HTTP GET で送信された変数が格納される ・連想配列として使用
S_POST (ポスト変数)	<ul style="list-style-type: none"> ・HTTP POST で送信された変数が格納される ・連想配列として使用
S_FILES (ファイル変数)	<ul style="list-style-type: none"> ・HTTP POSTでアップロードされた変数が格納される ・連想配列として使用
S_REQUEST (リクエスト変数)	<ul style="list-style-type: none"> ・HTTP リクエスト変数が格納される ・連想配列として使用
S_SESSION (セッション変数)	<ul style="list-style-type: none"> ・セッション変数が格納される ・連想配列として使用
S_ENV (環境変数)	<ul style="list-style-type: none"> ・環境変数が格納される ・連想配列として使用
S_COOKIE (クッキー変数)	<ul style="list-style-type: none"> ・HTTP クッキー変数が格納される ・連想配列として使用

- ・ `$_SERVER` に格納されている値

```
$_SERVER['SERVER_ADDR']
$_SERVER['SERVER_NAME']
$_SERVER['SERVER_PORT']
```

サーバのIPアドレス(例:192.168.0.1)
サーバの名前(例:www.example.com)
サーバのポート番号(例:80)

<code>\$_SERVER['SERVER_PROTOCOL']</code>	サーバプロトコル(例:HTTP/1.1)
<code>\$_SERVER['SERVER_ADMIN']</code>	サーバの管理者(例:root@localhost)
<code>\$_SERVER['SERVER_SIGNATURE']</code>	サーバのシグニチャ(例:Apache/2.2.15...)
<code>\$_SERVER['SERVER_SOFTWARE']</code>	サーバソフトウェア(例:Apache/2.2.15...)
<code>\$_SERVER['GATEWAY_INTERFACE']</code>	CGIバージョン(例:CGI/1.1)
<code>\$_SERVER['DOCUMENT_ROOT']</code>	ドキュメントルート(例:/var/www/html)
<code>\$_SERVER['PATH']</code>	環境変数PATHの値
(例:/sbin:/usr/sbin:/bin:/usr/bin)	
<code>\$_SERVER['PATH_TRANSLATED']</code>	スクリプトファイル名(例:/var/www/html/test.php)
<code>\$_SERVER['SCRIPT_FILENAME']</code>	スクリプトファイル名(例:/var/www/html/test.php)
<code>\$_SERVER['REQUEST_URI']</code>	リクエストのURI(例:/test.php)
<code>\$_SERVER['PHP_SELF']</code>	PHPスクリプト名(例:/test.php)
<code>\$_SERVER['SCRIPT_NAME']</code>	スクリプト名(例:/test.php)
<code>\$_SERVER['PATH_INFO']</code>	URLの引数に指定されたパス名(例:/test.php/aaa)
<code>\$_SERVER['ORIG_PATH_INFO']</code>	PHPで処理される前のPATH_INFO情報
<code>\$_SERVER['QUERY_STRING']</code>	URLの?以降に記述された引数(例:q=123)
<code>\$_SERVER['REMOTE_ADDR']</code>	クライアントのIPアドレス(例:192.168.0.123)
<code>\$_SERVER['REMOTE_HOST']</code>	クライアント名(例:client32.example.com)
<code>\$_SERVER['REMOTE_PORT']</code>	クライアントのポート番号(例:64799)
<code>\$_SERVER['REMOTE_USER']</code>	クライアントのユーザ名(例:tanaka)
<code>\$_SERVER['REQUEST_METHOD']</code>	リクエストメソッド(例:GET)
<code>\$_SERVER['REQUEST_TIME']</code>	リクエストのタイムスタンプ(例:1351987425)
<code>\$_SERVER['REQUEST_TIME_FLOAT']</code>	リクエストのタイムスタンプ(マイクロ秒)(PHP 5.1.0以降)
<code>\$_SERVER['REDIRECT_REMOTE_USER']</code>	リダイレクトされた場合の認証ユーザ(例:tanaka)
<code>\$_SERVER['HTTP_ACCEPT']</code>	リクエストのAccept:ヘッダの値(例:text/html)
<code>\$_SERVER['HTTP_ACCEPT_CHARSET']</code>	リクエストのAccept-Charset:ヘッダの値(例:utf-8)
<code>\$_SERVER['HTTP_ACCEPT_ENCODING']</code>	リクエストのAccept-Encoding:ヘッダの値(例:gzip)
<code>\$_SERVER['HTTP_ACCEPT_LANGUAGE']</code>	リクエストのAccept-Language:ヘッダの値(ja,en-US)
<code>\$_SERVER['HTTP_CACHE_CONTROL']</code>	リクエストのCache-Control:ヘッダの値(例:max-age=0)
<code>\$_SERVER['HTTP_CONNECTION']</code>	リクエストのConnection:ヘッダの値(例:keep-alive)
<code>\$_SERVER['HTTP_HOST']</code>	リクエストのHost:ヘッダの値(例:www.example.com)
<code>\$_SERVER['HTTP_REFERER']</code>	リンクの参照元URL(例:http://www.example.com/)
<code>\$_SERVER['HTTP_USER_AGENT']</code>	リクエストのUser-Agent:ヘッダの値
(例:Mozilla/5.0...)	
<code>\$_SERVER['HTTPS']</code>	HTTPSを利用しているか否か(例:on)
<code>\$_SERVER['PHP_AUTH_DIGEST']</code>	ダイジェスト認証時のAuthorization:ヘッダの値
<code>\$_SERVER['PHP_AUTH_USER']</code>	HTTP認証時のユーザ名
<code>\$_SERVER['PHP_AUTH_PW']</code>	HTTP認証時のパスワード
<code>\$_SERVER['AUTH_TYPE']</code>	HTTP認証時の認証形式

• スーパーグローバル変数からの値取得 (Symfony)

```
// $_GET['hoge']
$request->query->get('hoge');

// $_POST['hoge']
$request->request->get('hoge');

// ルーティングパラメータ / ex) @Route('/{hoge}')
$request->attributes->get('hoge');

// $_COOKIE['hoge']
$request->cookies->get('hoge');

// $_FILES['hoge']
$request->files->get('hoge');
```

```
// $_SERVER['SCRIPT_FILENAME']
$request->server->get('SCRIPT_FILENAME');

// $_SERVER['HTTP_USER_AGENT']
$request->headers->get('User-Agent');

// query > attribute > request の順で検索
$request->get('hoge');
```

◆ 変数展開

文字列の中で、変数の中身を取り出すことを『変数展開』と呼ぶ。

※Paizaで検証済み。

• シングルクォーテーションによる変数展開

シングルクォーテーションの中身は全て文字列として認識され、変数は展開されない。

```
$fruit = "リンゴ";

echo 'これは$fruitです。';

// 出力結果
これは、$fruitです。
```

• シングルクォーテーションと波括弧による変数展開

シングルクォーテーションの中身は全て文字列として認識され、変数は展開されない。

```
$fruit = "リンゴ";

echo 'これは{$fruit}です。';

// 出力結果
これは、{$fruit}です。
```

• ダブルクォーテーションによる変数展開

変数の前後に半角スペースを置いた場合にのみ、変数は展開される。（※半角スペースがないとエラーになる）

```
$fruit = "リンゴ";

echo "これは $fruit です。";

// 出力結果
これは リンゴ です。
```

• ダブルクォーテーションと波括弧による変数展開

波括弧を用いると、明示的に変数として扱うことができる。これによって、変数の前後に半角スペースを置かなくとも、変数は展開される。


```
$fruit = "リンゴ";

echo "これは${fruit}です。";

// 出力結果
これは、リンゴです。
```

◆ 参照渡しと値渡し

- 参照渡し

「参照渡し」とは、変数に代入した値の参照先（メモリアドレス）を渡すこと。

```
$value = 1;
$result = &$value; // 値の入れ物を参照先として代入
```

【実装例】 \$b には、\$a の参照によって10が格納される。

```
$a = 2;
$b = &$a; // 変数aを&をつけて代入
$a = 10;  // 変数aの値を変更
echo $b;

# 結果
10
```

- 値渡し

「値渡し」とは、変数に代入した値のコピーを渡すこと。

```
$value = 1;
$result = $value; // 1をコピーして代入
```

【実装例】 \$b には、\$a の一行目の格納によって2が格納される。

```
$a = 2;
$b = $a; // 変数aを代入
$a = 10; // 変数aの値を変更
echo $b;

# 結果
2
```

02-03. Bool値

◆ Falseの定義

- 表示なし
- キーワード FALSE false

- 整数 0
- 浮動小数点 0.0
- 空の文字列 ""
- 空の文字列 ''
- 文字列 "0" (文字列としての0)
- 要素数が 0 の配列 \$ary = array();
- プロパティーやメソッドを含まない空のオブジェクト
- NULL値

◆ Trueの定義

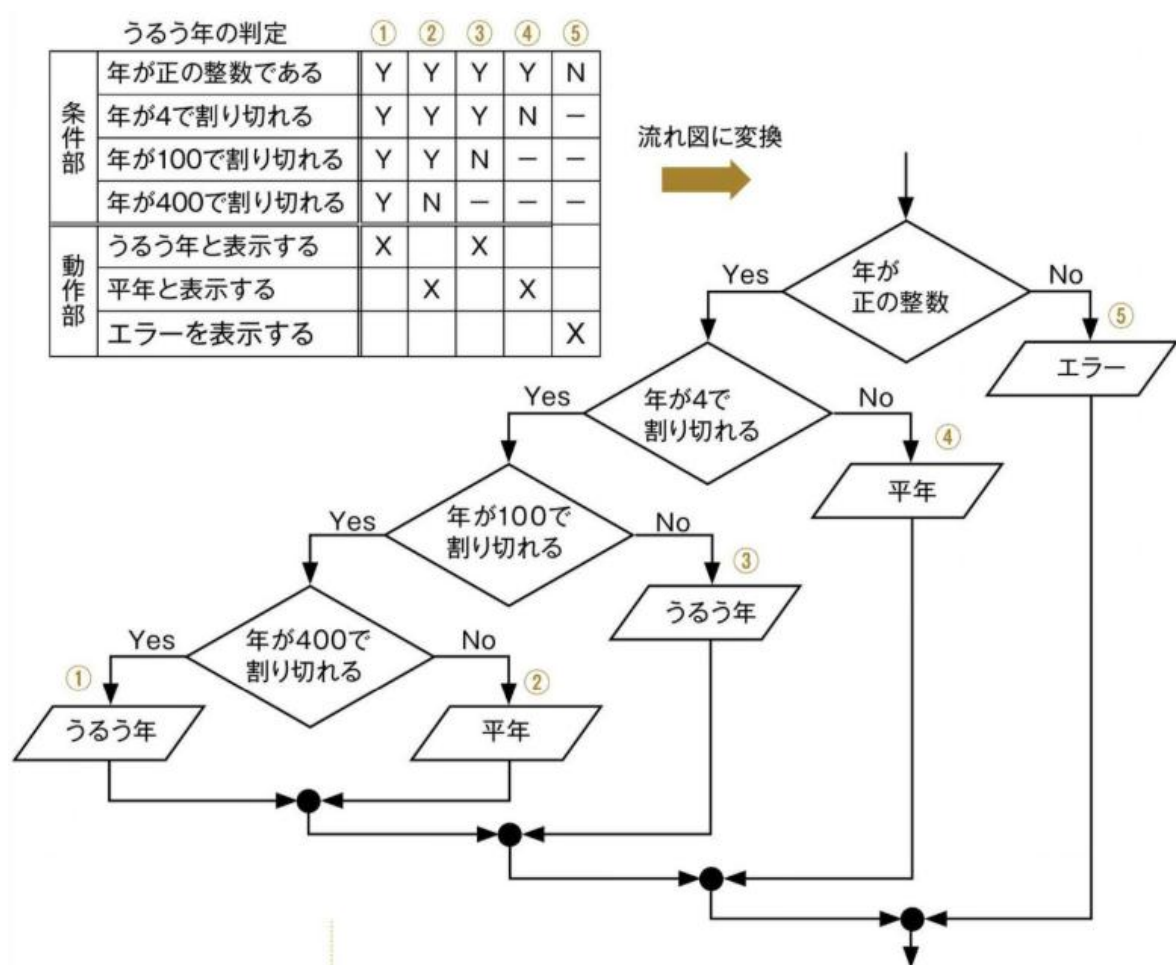
上記の値以外は、全て TRUEである。

02-04. 条件式

◆ 決定表

【作成例】

平年とうるう年を出力する処理を、決定表で表す。その後、流れ図として表す。



◆ If-Elseif と Switch-Case-Break

```
// 変数に Tue を格納
$weeks = 'Tue';

// if文でTueに該当したら'火曜日'と表示する。
if ($weeks == 'Mon') {
    echo '月曜日';
} else if($weeks == 'Tue'){
    echo '火曜日';
} else if($weeks == 'wed'){
    echo '水曜日';
} else if($weeks == 'Thu'){
    echo '木曜日';
} else if($weeks == 'Fri'){
    echo '金曜日';
} else if($weeks == 'Sat'){
    echo '土曜日';
} else if($weeks == 'Sun'){
    echo '日曜日';
}

// 実行結果
火曜日
```

```
// 変数に Tue を格納
$weeks = 'Tue';

// 条件分岐でTueに該当したら'火曜日'と表示する。breakでif文を抜けなければ、全て実行されてしま
う。
switch ($weeks) {
    case 'Mon':
        echo '月曜日';
        break;
    case 'Tue':
        echo '火曜日';
        break;
    case 'Wed':
        echo '水曜日';
        break;
    case 'Thu':
        echo '木曜日';
        break;
    case 'Fri':
        echo '金曜日';
        break;
    case 'Sat':
        echo '土曜日';
        break;
    case 'Sun':
        echo '日曜日';
        break;
}

// 実行結果
火曜日
```

◆ 『else』 はできるだけ用いない

- 『else』 を用いる場合

冗長になってしまう。

```
// マジックナンバーを使わずに、定数として定義
const noOptionItem = 0;

// RouteEntityからoptionsオブジェクトに格納されるoptionオブジェクト配列を取り出す。
if(!empty($routeEntity->options) {
    foreach ($routeEntity->options as $option) {

        // if文を通過した場合、メソッドの戻り値が格納される。
        // 通過しない場合、定数が格納される。
        if ($option->isOptionItemA()) {
            $result['optionItemA'] = $option->optionItemA();
        } else {
            $result['optionItemA'] = noOptionItem;
        }

        if ($option->isOptionItemB()) {
            $result['optionItemB'] = $option->optionItemB();
        } else {
            $result['optionItemB'] = noOptionItem;
        }

        if ($option->isOptionItemC()) {
            $result['optionItemC'] = $option->optionItemC();
        } else {
            $result['optionItemC'] = noOptionItem;
        }
    };
}

return $result;
```

- 初期値と上書きのロジックを用いる場合

よりすっきりした書き方になる。

```
// マジックナンバーを使わずに、定数として定義
const noOptionItem = 0;

// 初期値0を設定
$result['optionItemA'] = noOptionItem;
$result['optionItemB'] = noOptionItem;
$result['optionItemC'] = noOptionItem;

// RouteEntityからoptionsオブジェクトに格納されるoptionオブジェクト配列を取り出す。
if(!empty($routeEntity->options) {
    foreach ($routeEntity->options as $option) {

        // if文を通過した場合、メソッドの戻り値によって初期値0が上書きされる。
        // 通過しない場合、初期値0が用いられる。
```

```

        if ($option->isOptionItemA()) {
            $result['optionItemA'] = $option->optionItemA();
        }

        if ($option->isOptionItemB()) {
            $result['optionItemB'] = $option->optionItemB();
        }

        if ($option->isOptionItemC()) {
            $result['optionItemC'] = $option->optionItemC();
        }
    };
}

return $result;

```

◆ オブジェクトごとにプロパティの値の有無が異なる時の出力

```

// 全てのオブジェクトが必ず持っているわけではなく、
$csv['オリコ払い'] = $order->oricoCondition ? $order->oricoCondition->

// 全てのオブジェクトが必ず持っているプロパティの場合には不要
$csv['ID'] = $order->id;

```

02-05. 例外処理

データベースから取得した後に直接表示する値の場合、データベースでNullにならないように制約をかけられるため、変数の中身に例外判定を行う必要はない。しかし、データベースとは別に新しく作られる値の場合、例外判定が必要になる。

◆ 例外処理前の条件分岐

	if(\$var)	isset(\$var)	! empty(\$var)	! is_null(\$var)
0	×	○	×	○
1	○	○	○	○
""	×	○	×	○
"あ"	×	○	×	○
NULL	×	×	×	×
array()	×	○	×	○
array(1)	○	○	○	○

右辺には、上記に当てはまらない状態『TRUE』が置かれている。

```
if($this->$var == TRUE){  
    // 処理A;  
}
```

ただし、基本的に右辺は省略すべき。

```
if($this->$var){  
    // 処理A;  
}
```

◆ Exceptionクラスを継承した独自例外クラス

```
// HttpRequestに対処する例外クラス  
class HttpRequestException extends Exception  
{  
    // インスタンスが作成された時に実行される処理  
    public function __construct()  
    {  
        parent::__construct("HTTPリクエストに失敗しました", 400);  
    }  
  
    // 新しいメソッドを定義  
    public function example()  
    {  
        // なんらかの処理;  
    }  
}
```

◆ If-Throw文

特定の処理の中に、想定できる例外があり、それをエラー文として出力するために用いる。ここでは、全ての例外クラスの親クラスであるExceptionクラスのインスタンスを投げている。

```
if (empty($value)) {  
    throw new Exception('Variable is empty');  
}  
  
return $value;
```

◆ Try-Catch文

特定の処理の中に、想定できない例外があり、それをエラー文として出力するために用いる。定義されたエラー文は、デバック画面に表示される。

```
// Exceptionを投げる  
try{  
    // WebAPIの接続に失敗した場合
```

```

    if(...){
        throw new WebAPIException();
    }

    if(...){
        throw new HttpRequestException();
    }

    // try文で指定のExceptionが投げられた時に、指定のcatch文に入る
    // あらかじめ出力するエラーが設定されている独自例外クラス（以下参照）
}catch(WebAPIException $e){
    // エラー文を出力。
    print $e->getMessage();

}catch(HttpRequestException $e){
    // エラー文を出力。
    print $e->getMessage();

}

// Exceptionクラスはtry文で生じた全ての例外をキャッチしてしまうため、最後に記述するべき。
}catch(Exception $e){
    // 特定できなかったことを示すエラーを出力
    throw new Exception("なんらかの例外が発生しました。")

}

// 正常と例外にかかわらず、必ず実行される。
}finally{
    // 正常と例外にかかわらず、必ずファイルを閉じるための処理
}

```

以下、上記で使用した独自の例外クラス。

```

// HttpRequestに対処する例外クラス
class HttpRequestException extends Exception
{
    // インスタンスが作成された時に実行される処理
    public function __construct()
    {
        parent::__construct("HTTPリクエストに失敗しました", 400);
    }
}

```

```

// HttpRequestに対処する例外クラス
class HttpRequestException extends Exception
{
    // インスタンスが作成された時に実行される処理
    public function __construct()
    {
        parent::__construct("HTTPリクエストに失敗しました", 400)
    }
}

```

02-06. 反復処理

◆ Foreachの用途

- 配列を返却したい場合

- いずれかの配列の要素を返却する場合

- エンティティごとに、プロパティの値を持つか否かが異なる場合

◆ 無限ループ

反復処理では、何らかの状態になった時に反復処理を終えなければならない。しかし、終わることができないと、無限ループが発生してしまう。

- **break**

```
// 初期化
$i = 0;
while($i < 4){

    echo $i;

    // 改行
    echo PHP_EOL;
}
```

- **continue**

02-07. 演算子

◆ 等価演算子を用いたオブジェクトの比較

- イコールが2つの場合

同じオブジェクトから別々に作られたインスタンスであっても、『同じもの』として認識される。

```
class Example {}

if(new Example == new Example){
    echo '同じです';
} else { echo '異なります' }

// 実行結果
同じです
```


- **イコールが3つの場合**

同じオブジェクトから別々に作られたインスタンスであっても、『異なるもの』として認識される。

```
class Example {};  
  
if(new Example === new Example){  
    echo '同じです';  
} else { echo '異なります' }  
  
// 実行結果  
異なります
```

同一のインスタンスの場合のみ、『同じもの』として認識される。

```
class Example {};  
  
$a = $b = new Example;  
  
if($a === $b){  
    echo '同じです';  
} else { echo '異なります' }  
  
// 実行結果  
同じです
```

◆ キャスト演算子

【実装例】

```
$var = 10; // $varはInt型。  
  
// キャスト演算子でデータ型を変換  
$var = (string) $var; // $varはString型
```

【その他のキャスト演算子】

```
// Int型  
$var = (int) $var  
(integer)  
  
// Boolean型  
$var = (bool) $var  
(boolean)  
  
// Float型  
$var = (float) $var  
(double)  
(real)  
  
// Array型  
$var = (array) $var
```

```
// Object型
$var = (object) $var
```

04. 実装のモジュール化

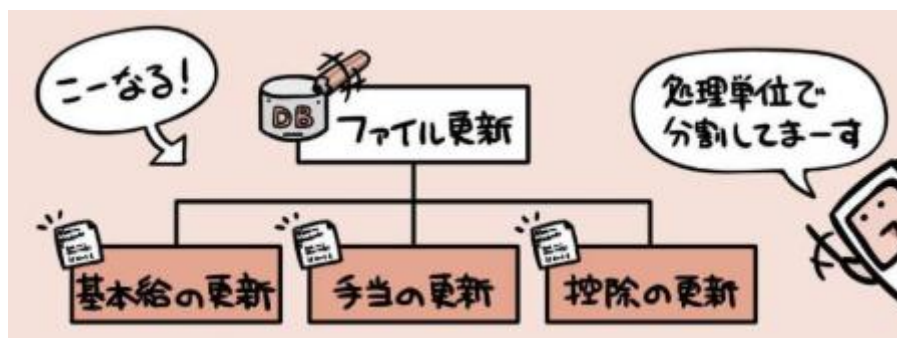
◆ STS分割法

プログラムを、『Source（入力処理）→ Transform（変換処理）→ Sink（出力処理）』のデータの流れに則って、入力モジュール、処理モジュール、出力モジュール、の3つ分割する方法。（リクエスト → DB → レスポンス）



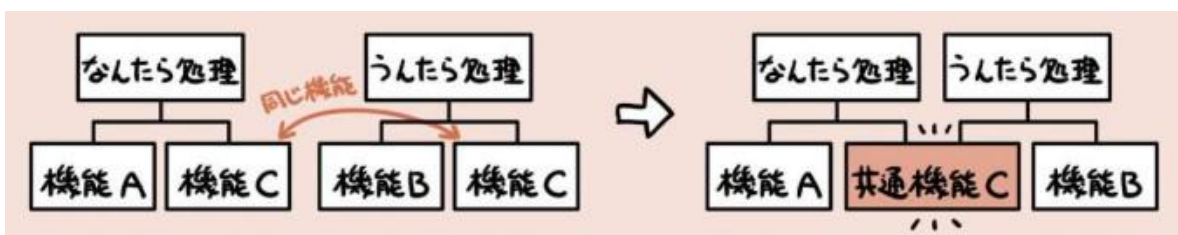
◆ Transaction分割法

データの種類によってTransaction（処理）の種類が決まるような場合に、プログラムを処理の種類ごとに分割する方法。



◆ 共通機能分割法

プログラムを、共通の機能ごとに分割する方法



◆ MVC

ドメイン駆動設計のノートを参照せよ。

◆ ドメイン駆動設計

ドメイン駆動設計のノートを参照せよ。

◆ デザインパターン

デザインパターンのノートを参照せよ。

05. ファイルパス

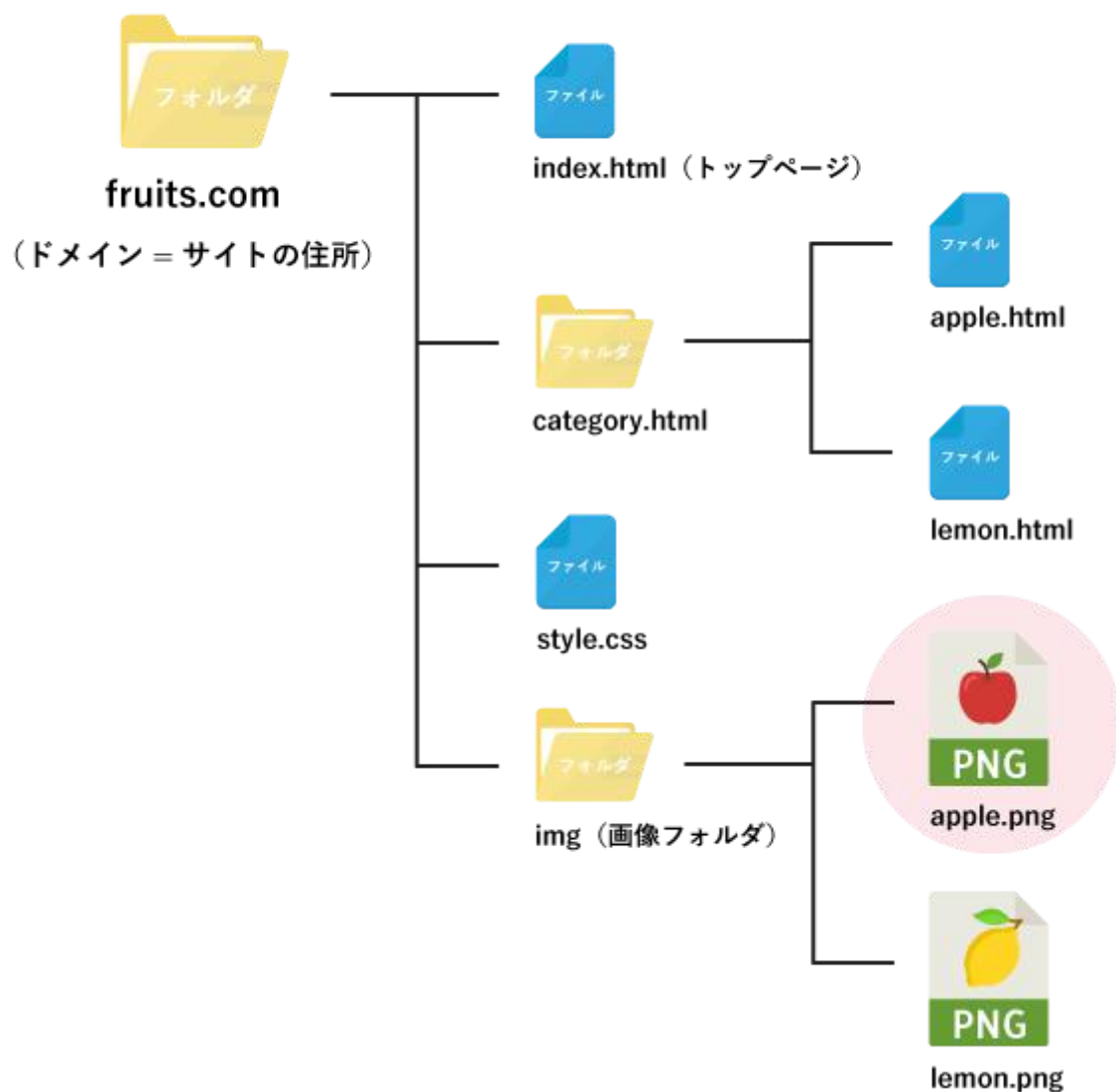
◆ 絶対パス

ルートディレクトリ（fruit.com）から、指定のファイル（apple.png）までのパス。

```

```

絶対パスでファイルを読み込む



◆ 相対パス

起点となる場所（apple.html）から、指定のディレクトリやファイル（apple.png）の場所までを辿るパス。例えば、apple.htmlのページでapple.pngを使用したいとする。この時、『..』を用いて一つ上の階層に行き、青の後、imgフォルダを指定する。

```

```

相対パスでファイルを読み込む

