本格的な単体テストを書くハンズオン　問題の解説

1問目
1問目は引数が複数のパターンを持つメソッドに対してテストを書く問題です。
問題のように引数は1個ですが、指定できる値は複数のパターンが指定可能となっています。

具体的には/src/authentic/q001というフォルダ内のWarekiクラスのgetWarekiメソッドに対して単体テストを書く問題となります。
getWarekiメソッドは引数に年月日としてyyyymmddを指定し、
戻り値として明治ならM、大正ならTのように元号の頭文字を返します。
なお、江戸以前や引数不正はエラーとしてERRORという文字列を、
新元号はNを返します。

問題ソースコード(/src/authentic/q001)
```
<?php

/**
 * Created by PhpStorm.
 * User: fortegp05
 */
namespace LaravelJpCon¥q001;
use DateTime;

/**
 * Class Wareki
 * @package LaravelJpCon¥Question¥q001
 */
class Wareki
{
    const MEIJI = 'M';
    const TAISYO = 'T';
    const SYOWA = 'S';
    const HEISEI = 'H';
    const NEW_GENGO = 'N';
    const DATE_FORMAT = 'Ymd';
    const ERROR_MSG = 'ERROR';
    private $wareki_start_days = [
        self::MEIJI => '18680125',
        self::TAISYO => '19120730',
        self::SYOWA => '19261225',
        self::HEISEI => '19890108',
        self::NEW_GENGO => '20190501'
    ];

    /** yyyymmddを入力すると和暦を返す
     * 入力値異常、江戸以前はERROR、新年号はNとして返す
     * @param string $ymd
     * @return string
     */
    public function getWareki(string $ymd): string
    {
        if (!$this->isDate($ymd)) {
            return self::ERROR_MSG;
        }
        switch (true) {
```

```
            case $ymd < $this->wareki_start_days[self::MEIJI]:
                return self::ERROR_MSG;
            case $this->wareki_start_days[self::MEIJI] <= $ymd && $ymd < $this->wareki_start_days[self::TAISYO]:
                return self::MEIJI;
            case $this->wareki_start_days[self::TAISYO] <= $ymd && $ymd < $this->wareki_start_days[self::SYOWA]:
                return self::TAISYO;
            case $this->wareki_start_days[self::SYOWA] <= $ymd && $ymd < $this->wareki_start_days[self::HEISEI]:
                return self::SYOWA;
            case $this->wareki_start_days[self::HEISEI] <= $ymd && $ymd <
$this->wareki_start_days[self::NEW_GENGO]:
                return self::HEISEI;
            case $this->wareki_start_days[self::NEW_GENGO] <= $ymd:
                return self::NEW_GENGO;
        }
        return self::ERROR_MSG;
    }

    /** 引数チェックyyyymmddの妥当性チェック
     * @param string $ymd
     * @return bool
     */
    private function isDate(string $ymd): bool
    {
        $date = DateTime::createFromFormat(self::DATE_FORMAT, $ymd);
        return $date && $date->format(self::DATE_FORMAT) == $ymd;
    }
}
```

回答例

```
<?php
/**
 * Created by PhpStorm.
 * User: fortegp05
 */

use LaravelJpCon¥q001¥Wareki;
use PHPUnit¥Framework¥TestCase;

/**
 * Class WarekiTest
 */
class WarekiTest extends TestCase
{
    private $wareki;
    const RESULT_MSG = 'RESULT';
    const INPUT = 'INPUT';
    private $testData = [
        '日付じゃない(文字列)' => [self::RESULT_MSG => Wareki::ERROR_MSG, self::INPUT => 'HOGEHOGE'],
        '日付じゃない(数値)' => [self::RESULT_MSG => Wareki::ERROR_MSG, self::INPUT => '00000000'],
        'あり得ない日付' => [self::RESULT_MSG => Wareki::ERROR_MSG, self::INPUT => '20180229'],
        '江戸終了境界' => [self::RESULT_MSG => Wareki::ERROR_MSG, self::INPUT => '18680124'],
        '明治開始境界' => [self::RESULT_MSG => Wareki::MEIJI, self::INPUT => '18680125'],
        '明治終了境界' => [self::RESULT_MSG => Wareki::MEIJI, self::INPUT => '19120729'],
        '大正開始境界' => [self::RESULT_MSG => Wareki::TAISYO, self::INPUT => '19120730'],
        '大正終了境界' => [self::RESULT_MSG => Wareki::TAISYO, self::INPUT => '19261224'],
        '昭和開始境界' => [self::RESULT_MSG => Wareki::SYOWA, self::INPUT => '19261225'],
        '昭和終了境界' => [self::RESULT_MSG => Wareki::SYOWA, self::INPUT => '19890107'],
        '平成開始境界' => [self::RESULT_MSG => Wareki::HEISEI, self::INPUT => '19890108'],
        '平成終了境界' => [self::RESULT_MSG => Wareki::HEISEI, self::INPUT => '20190420'],
        '新年号開始境界' => [self::RESULT_MSG => Wareki::NEW_GENGO, self::INPUT => '20190501'],
    ];

    /**
     * 初期設定
     */
    public function setUp()
    {
        $this->wareki = new Wareki();
    }

    /**
     * dataProvider未使用版。
     *
     */
    public function test_getWareki_no_dataProvider()
    {
        foreach ($this->testData as $title => $data) {
            $this->assertSame($data[self::RESULT_MSG], $this->wareki->getWareki($data[self::INPUT]), $title);
        }
    }

    /**
     * dataProvider使用版。
     * 入力しているデータはkey=>array()となっているが、引数で入力されるのはarray()のみ
     * keyは失敗時にテスト名として表示される
     * @dataProvider dataProvider
     * @param $expected
     * @param $input
     */
    public function test_getWareki_dataProvider($expected, $input)
```

```
    {
        $this->assertSame($expected, $this->wareki->getWareki($input));
    }

    /**
     * @return array
     */
    public function dataProvider()
    {
        return $this->testData;
    }
}
```

解説
この問題の目的は以下の通りです。
・正常、異常、境界のパターンを書けること
・setUpについて
・dataProviderについて学ぶこと

まずこのgetWarekiメソッドは引数に年月日(yyyymmdd)を取ります。
そしてこの引数は一致する元号の頭文字(明治ならM)を返します。
そのため、引数が1868/01/25～1912/07/29の間は戻り値がMとなることをテストする必要があります。
まず異常テストとして年月日以外の入力と、江戸時代(1868/01/24)があり、
明治時代の境界テストとして1868/01/25、1912/07/29があります。
そして大正のテスト項目として1912/07/30をテストすれば正常(兼境界)、異常の各パターンが満たせます。
あとは元号分同じ考えで行います。
なお、最後の新元号は終わりが決まっていないため開始のみテストを行います。

次にsetUpメソッドについて説明します。
setUpメソッドはtestメソッド(assertionを実行するメソッド)を実行する前に実行されるメソッドです。
このメソッドではファイルのオープンとか、DB接続などのテストしたい処理の下準備を行うためのメソッドます。
そのため、テスト対象のクラスのインスタンス化処理は多くの場合でsetUpメソッドに記述されます。
同様に後始末用のメソッドとして、tearDownメソッドというものもあります。
こちらはtestメソッドの実行後に実行されます。
ファイルやDBのクローズ、不要な変数の解放などに使用されます。

最後にdataProviderについて説明します。
今回のテストの様に引数の組み合わせが多い場合に有効なのが、dataProviderです。
引数のパターンが多い場合は回答例のように配列に期待値(expected)、結果(actual)、パターン名を定義してテストすることが多いです。
このとき、通常は配列なのでforループなどループ処理を書きます。
しかし、dataProviderを使うことで以下のメリットが得られます。
・setUp、tearDownが使える
・例外処理に対応

setUp、tearDownが使えることで、1ループごとの事前準備や後始末がシンプルに記述可能です。
次に例外が発生する場合でも途中で止まらずにテスト可能です。
通常のループ処理では、例外が発生しても全パターンテストするためにはループ処理の中にtry/catchを書く必要がありますが、
dataProviderなら例外が発生しても次のパターンに進むためtry/catchは不要です。

2問目
2問目は入力値以外の要素で出力が変わる場合です。
引数が無い場合でもその他の要因、例えばDB接続に失敗した場合、
どこかで例外が発生した場合などに対するテストの書き方を学ぶ問題です。

今回は山手線を例に、いま電車が山手線のどの駅にいるのか?
つまりカーソルをクラスのprivateなプロパティとして持っている場合となっています。
このクラスには内回り、外回りで電車を進める、つまりカーソルを進める処理が実装されていたり、単純に駅名を取り出す処理があります。
これらを組み合わせて山手線に対するテストを書いてください。

問題ソースコード(/src/authentic/q002)
```
<?php
namespace LaravelJpCon¥q002;

/**
 * Class YamanoteLine
 * @package LaravelJpCon¥Question¥q002
 */
class YamanoteLine
{
    // 東京から内回りで定義した駅名配列
    private $stations = [
        '東京駅',
        '神田駅',
        '秋葉原駅',
        '御徒町駅',
        '上野駅',
        '鶯谷駅',
        '日暮里駅',
        '西日暮里駅',
        '田端駅',
        '駒込駅',
        '巣鴨駅',
        '大塚駅',
        '池袋駅',
        '目白駅',
        '高田馬場駅',
        '新大久保駅',
        '新宿駅',
        '代々木駅',
        '原宿駅',
        '渋谷駅',
        '恵比寿駅',
        '目黒駅',
        '五反田駅',
        '大崎駅',
        '品川駅',
        '高輪ゲートウェイ駅',
        '田町駅',
        '浜松町駅',
        '新橋駅',
        '有楽町駅'
    ];
    private $cursor = 0;
```

```
    /**
     * 現在の駅を取得する(東京から始まる)
     * @return string
     */
    public function nowStation(): string
    {
        return $this->stations[$this->cursor];
    }

    /**
     * @return string
     */
    public function moveCursorUchimawari(): void
    {
        $this->cursor++;
        if ($this->cursor > (count($this->stations) - 1)) {
            $this->cursor = 0;
        }
    }

    /**
     * 外回りで駅を取得する(東京->神田->秋葉原->御徒町->上野…)
     * @return string
     */
    public function nextStationUchimawari(): string
    {
        // カーソル移動して駅名を取得する
        $this->moveCursorUchimawari();
        return $this->nowStation();
    }

    /**
     * @return string
     */
    public function moveCursorSotomawari(): void
    {
        $this->cursor--;
        if ($this->cursor < 0) {
            $this->cursor = count($this->stations) - 1;
        }
    }

    /**
     * 外回りで駅を取得する(東京->有楽町->新橋->浜松町…)
     * @return string
     */
    public function nextStationSotomawari(): string
    {
        // カーソル移動して駅名を取得する
        $this->moveCursorSotomawari();
        return $this->nowStation();
    }

    /**
     * 東京に戻す
     */
    public function reset(): void
    {
        $this->cursor = 0;
    }
}
```

解答例

```
<?php
/**
 * Created by PhpStorm.
 * User: fortegp05
 */

use LaravelJpCon¥q002¥YamanoteLine;
use PHPUnit¥Framework¥TestCase;

/**
 * Class YamanoteLineTest
 */
class YamanoteLineTest extends TestCase
{
    private $yamanoteLine;

    /**
     * 初期設定
     */
    public function setUp()
    {
        $this->yamanoteLine = new YamanoteLine();
    }

    /**
     * 開始駅テスト
     */
    public function test_startStationIsTokyo()
    {
        $this->assertSame('東京駅', $this->yamanoteLine->nowStation(), '東京から始まっているか?');
    }

    /**
     * 東京からの内回りテスト
     */
    public function test_uchimawari()
    {
        $this->assertSame('神田駅', $this->yamanoteLine->nextStationUchimawari(), '内回りの駅確認');
    }

    /**
     * 東京からの外回りテスト
     */
    public function test_sotomawari()
    {
        $this->assertSame('有楽町駅', $this->yamanoteLine->nextStationSotomawari(), '外回りの駅確認');
    }

    /**
     * 内回りのループ確認
     */
    public function test_loopUchimawari()
    {
        for ($i = 0; $i < 29; $i++) {
            $this->yamanoteLine->moveCursorUchimawari();
        }
        $this->assertSame('東京駅', $this->yamanoteLine->nextStationUchimawari(), '内回りのループ確認');
    }
```

```
    /**
     * 外回りのループ確認
     */
    public function test_loopSotomawari()
    {
        for ($i = 0; $i < 29; $i++) {
            $this->yamanoteLine->moveCursorSotomawari();
        }
        $this->assertSame('東京駅', $this->yamanoteLine->nextStationSotomawari(), '外回りのループ確認');
    }

    /**
     * リセット後の駅確認テスト
     */
    public function test_resetCursorIsTokyo()
    {
        $this->yamanoteLine->moveCursorUchimawari();
        $this->yamanoteLine->reset();
        $this->assertSame('東京駅', $this->yamanoteLine->nowStation(), '東京にリセットされるか?');
    }
}
```

解説
この問題の目的は、
内部で持っているprivateなプロパティで出力結果が変わる場合のテストの書き方を学ぶことです。
privateなプロパティであればリフレクションでアクセス可能にしてまえば良い、という考えもありますが、
今回のように処理(仕様)が単純明快であり、
メソッド分割がなされている場合はシンプルに書いた方がわかりやすいと思います。

テストパターンとしては以下の2点を確認する必要があります。
- 内回り、外回りで正しい駅が取得できること
- 環状線(山手線)なので1週したら元に戻ること

ひとつ目は単純に内回りであれば神田、秋葉原、
外回りであれば有楽町、新橋と順番に取得できることをテストします。

ふたつ目は環状線である山手線は、東京から始まって1週したらまた東京に戻ってくるということをテストする必要がある点です。
そのため、内回り外回りともに31個目はまた東京になっていることを確認します。
より丁寧にテストするならば、その次がそれぞれ神田、有楽町になっていることをテストしても良いかと思います。

3問目
バブルソートに対するテストを書く問題です。
バブルソートについてはこちらを参照してください。
https://ja.wikipedia.org/wiki/%E3%83%90%E3%83%96%E3%83%AB%E3%82%BD%E3%83%BC%E3%83%88

この問題は配列が入出力になる場合のメソッドに対してテストを書きます。
このメソッドは与えられた引数の数字配列を昇順でソートするので、
引数に配列を与え、戻り値も配列です。
戻り値が配列の場合にテストコードを書く問題となります。

ヒント：assertArraySubset(array $subset, array $array[, bool $strict = false, string $message = ''])

問題ソースコード
```
<?php
namespace LaravelJpCon¥q004;

/**
 * Created by PhpStorm.
 * User: fortegp05
 */
class BubbleSort
{
    /**
     * @param array $array
     * @return array
     */
    public function sort(array $array): array
    {
        $size = count($array);
        for ($i = 0; $i < $size; $i++) {
            for ($j = ($size - 1); $i < $j; $j--) {
                if ($array[$j] < $array[$j - 1]) {
                    $temp = $array[$j - 1];
                    $array[$j - 1] = $array[$j];
                    $array[$j] = $temp;
                }
            }
        }
        return $array;
    }
}
```

解答例

```
<?php
use LaravelJpCon¥q004¥BubbleSort;
use PHPUnit¥Framework¥TestCase;
/**
 * Created by PhpStorm.
 * User: fortegp05
 */
class BubbleSortTest extends TestCase
{
    private $bubbleSort;
    /**
     * 初期設定
     */
    public function setUp()
    {
        $this->bubbleSort = new BubbleSort();
    }
    /**
     * 正常にソートされる
     */
    public function test_normalSort()
    {
        $expectArray = [1, 2, 3, 4, 5];
        $array = [2, 4, 5, 3, 1];
        $this->assertArraySubset($expectArray, $this->bubbleSort->sort($array));
    }
    /**
     * 1個だけの配列
     */
    public function test_oneData()
    {
        $expectArray = [1];
        $array = [1];
        $this->assertArraySubset($expectArray, $this->bubbleSort->sort($array));
    }
    /**
     * 空の配列
     */
    public function test_blankArray()
    {
        $array = [];
        $this->assertArraySubset($array, $this->bubbleSort->sort($array));
    }
}
```

解説

この問題の目的は入出力が配列であるメソッドに対するテストの書き方を学ぶことです。
ヒントにあるとおり、assertArraySubset()を使用してテストします。
実装上、nullなどarray以外の引数は実行時エラーになるので異常系のテストは行いません。
そのため、境界として1個だけの配列、空の配列を引数にした場合のテストを行います。

assertArraySubset()を用いて、期待値にも入力値にも配列を指定します。
期待値としてソートされた後の数値配列を指定し、
1個だけの配列、空の配列はそれぞれ同じものを指定します。

4問目
例外が発生する実装になっているソースコードに対してテストコードを書く問題です。
引数不足時の例外、戻り値の型不正など、実際にIDEなどを使っているとこういったことはないかもしれませんが、
意図的に例外を返す実装を想定してテストを書く問題となっています。

ヒント：TestCase.expectException(Exception.class) // 期待される例外を指定する
TestCase.expectExceptionMessage(string message) // 期待される例外メッセージを指定する

問題ソースコード
```
<?php

// メソッド呼び出し時に例外が出るように定義
declare(strict_types=1);
namespace LaravelJpCon¥q006;

/**
 * Class ThrowException
 * @package LaravelJpCon¥q006
 */
class ThrowException
{
    const MESSAGE_ARGUMENT_IS_TOO_SHORT = 'can not explode because $str is too short';
    /**
     * 入力された文字列を文字列で分割しようとはしているが...？
     */
    public function callExplodeThrowsException(string $str): array
    {
        if (strlen($str) <= 1) {
            // 一文字しかないので引数が不正として例外にする
            throw new ¥Exception(self::MESSAGE_ARGUMENT_IS_TOO_SHORT);
        }
        // explode ( string $delimiter , string $string [, int $limit = PHP_INT_MAX ] ) : array
        // これでは明らかに引数が足らない
        // 実際には ArgumentCountError が発生する
        return explode($str);
    }

    /**
     * @param string $str
     * @return string
     * @throws ¥Exception
     * なお、正しくは戻り値の型がarrayですが、テスト用にstringにしています。
     */
    public function callExplodeReturnTypeIsBad(string $str): string
    {
        if (strlen($str) <= 1) {
            // 一文字しかないので引数が不正として例外にする
            throw new ¥Exception(self::MESSAGE_ARGUMENT_IS_TOO_SHORT);
        }
        return explode($str, ",");
    }

    /**
     * @param string $str
```

```
     * @return array
     * @throws ¥Exception
     */
    public function callExplodeNormally(string $str): array
    {
        if (strlen($str) <= 1) {
            // 一文字しかないので引数が不正として例外にする
            throw new ¥Exception(self::MESSAGE_ARGUMENT_IS_TOO_SHORT);
        }
        return explode(",", $str);
    }
}
```

解答例

```
<?php

use LaravelJpCon¥q006¥ThrowException;
use PHPUnit¥Framework¥TestCase;

/**
 * Class ThrowExceptionTest
 */
class ThrowExceptionTest extends TestCase
{
    private $obj;
    public function setUp()
    {
        $this->obj = new ThrowException();
    }

    public function testExplodeThrowsException()
    {
        $this->expectException(¥ArgumentCountError::class);
        $this->obj->callExplodeThrowsException("test");
    }

    public function testExplodeThrowsException_argumentStrIsTooShort()
    {
        $this->expectException(¥Exception::class);
        $this->expectExceptionMessage(ThrowException::MESSAGE_ARGUMENT_IS_TOO_SHORT);
        $this->obj->callExplodeThrowsException("t");
    }

    public function testExplodeReturnTypeIsBad_argumentStrIsTooShort()
    {
        $this->expectException(¥Exception::class);
        $this->expectExceptionMessage(ThrowException::MESSAGE_ARGUMENT_IS_TOO_SHORT);
        $this->obj->callExplodeReturnTypeIsBad("t");
    }

    public function testExplodeReturnTypeIsBad_SignatureIsBad()
    {
        $expected = ['a', 'b'];
        $this->expectException(¥TypeError::class);
        $this->assertSame($expected, $this->obj->callExplodeReturnTypeIsBad("a,b"));
    }

    public function testExplodeNormally_argumentStrIsTooShort()
    {
        $this->expectException(¥Exception::class);
        $this->expectExceptionMessage(ThrowException::MESSAGE_ARGUMENT_IS_TOO_SHORT);
        $this->obj->callExplodeNormally("t");
    }

    public function testExplodeNormally()
    {
        $expected = ['a', 'b', 'c'];
        $this->assertSame($expected, $this->obj->callExplodeNormally("a,b,c"));
    }
}
```

解説
expectExceptionメソッドを用いて、テストしたい例外を指定します。
これにより例外が発生するメソッドでもテストが可能になります。
また、expectExceptionMessageで例外のメッセージを指定することで、
独自の例外もテストが可能となります。
これにより引数エラーのような例外もテストが可能となります。

5問目
number_formatに対するテストパターンを考えるテストになります。
このnumber_formatは数値を桁区切りしてくれるメソッドですが、
このメソッドは少し特殊な挙動をするので、それを学ぶ目的もあります。

問題ソースコード
```
<?php

namespace LaravelJpCon¥q005;

/**
 * Created by PhpStorm.
 * User: tetsunosuke
 */
class NumFormat
{
    /**
     * @param string $str
     * @return string
     */
    public function format(string $str): string
    {
        // number_format ( float $number [, int $decimals = 0 ] ) : string
        // この変換はやや想定外の挙動をするので、テストしておきたい
        $f = floatval($str);
        return number_format($f);
    }
}
```

解答例

```
<?php

use LaravelJpCon¥q005¥NumFormat;
use PHPUnit¥Framework¥TestCase;

/**
 * Created by PhpStorm.
 * User: fortegp05
 */
class NumFormatTest extends TestCase
{
    private $numFormat;
    /**
     * 初期設定
     */
    public function setUp()
    {
        $this->numFormat = new NumFormat();
    }

    /**
     * フォーマットテスト 通常テスト
     */
    public function test_formatNormalNum()
    {
        $this->assertEquals("10,000,000", $this->numFormat->format("10000000"));
    }

    /**
     * フォーマットテスト 少数テスト
     */
    public function test_formatDecimal()
    {
        $this->assertEquals("0", $this->numFormat->format("0.05"));
    }

    /**
     * フォーマットテスト 引数が数字
     */
    public function test_formatParamNum()
    {
        // タイプヒントの挙動は設定次第
        $this->assertEquals("1,000", $this->numFormat->format(1000));
    }
```

```
    /**
     * フォーマットテスト 引数が文字列
     */
    public function test_formatParamText()
    {
        // 文字列は0として扱われる
        $this->assertEquals("0", $this->numFormat->format("test"));
    }

    /**
     * フォーマットテスト 引数が16進数文字列
     */
    public function test_formatParamTextNearHexadecimal()
    {
        // 16進ぽくても同じ
        $this->assertEquals("0", $this->numFormat->format("beef"));
    }

    /**
     * フォーマットテスト 引数が数字っぽい文字列
     */
    public function test_formatParamNumTextAfter()
    {
        // 後半の文字列が捨てられる
        $this->assertEquals("1,234", $this->numFormat->format("1234test"));
    }

    /**
     * フォーマットテスト 引数が数字っぽい文字列
     */
    public function test_formatParamNumTextMix()
    {
        // 後半の文字列が捨てられる
        $this->assertEquals("1", $this->numFormat->format("1-2-3-4"));
    }
}
```

解説
number_formatは数値を引数に文字列を返すメソッドです。
そのため、まずテストパターンとしては以下を考えると思います。
- ・数値を入れてみる(正常系)
- ・数値以外(文字列)を入れてみる(異常系)

さらにここから踏み込んでみることが大切です。
ここでいう数値は1種類しかないように見えますが、
PHPで扱える(プログラムで扱える)数値にはたくさんの種類があります。
正数、負数、整数、小数、n進数など。
この中で正数、整数は数値だけなのでテストが不要と考えます。
つぎに負数は符号としての-が付いているだけなので、丸まりのように値が変わることはないことからこちらもテスト不要と考えます。
ですが少数、n進数は小数点以降の丸まりや文字列に見える数値の扱いを確認するためにテストを行う必要があります。

次に異常系として文字列を入れた場合の挙動です。
"ABC"のような文字列は0を返してきますが、
"1000"のような数値文字列の場合には"1,000"を返してきます。
さらに”1ABC”の場合は0ではなく、"1"を返してきます。
頭から数値として扱える範囲だけをフォーマットして返し、
それ以降の文字列は切り捨てられます。
こういった挙動が特殊な挙動になります。

6問目
簡易的なブラックジャックゲームをテストする問題です。
乱数で結果が変わる処理の場合、結果が標準出力の場合に対するテストを学びます。
昔のバッチ処理などに使えるかと思います。
ぜひ、ヒントを見ながら解いてみてください。

ヒント：
リフレクションとアクセス権限のセット
```
$class = new ¥ReflectionClass(ClassName)
$class->getMethod // メソッドを取得
$class->getProperty // プロパティを取得
setAccessible // メソッド、プロパティのアクセス権限をセット
invoke($class) // メソッド実行
setValue($class, value) // プロパティに値をセット
```

標準出力の取得。
```
ob_start();
  ～ここで標準出力する～
$var = ob_get_clean();
```

問題ソースコード
```
<?php
namespace LaravelJpCon¥q003;

/**
 * Class Blackjack
 * ルール：カードを引いていき、21を超えたら負け。
 * 21以下で勝負した時にディーラー(COM)との比較で21に近い方が勝ち。
 * 同じ場合は引き分け。
 * それ以外のルールはなし。
 */
class Blackjack
{
    const SPADE = 'spade';
    const HEART = 'heart';
    const DIAMOND = 'diamond';
    const CLUB = 'club';
    private $myPoint;
    private $dealerPoint;
    private $cards = [
        self::SPADE => [],
        self::HEART => [],
        self::DIAMOND => [],
        self::CLUB => []
    ];
    private $suits = [self::SPADE, self::HEART, self::DIAMOND, self::CLUB];

    /**
     *
     */
    public function gameStart(): void
    {
```

```
    $this->initGame();
    $this->stdout('簡易ブラックジャック開始!');
    $this->playGame();
}

/**
 *
 */
private function playGame(): void
{
    while ($this->showOrder()) {
        // ゲーム中…
    }
    $this->gameSet();
}

private function initGame(): void
{
    $this->initCard();
    $this->initHand();
}

/**
 *
 */
private function initCard()
{
    foreach ($this->cards as $suit => $suitCards) {
        $this->cards[$suit] = [];
        for ($i = 0; $i < 13; $i++) {
            $this->cards[$suit][] = ($i + 1);
        }
    }
}

/**
 *
 */
private function initHand()
{
    $this->myPoint = 0;
    $this->dealerPoint = 0;
}

/**
 * @return bool
 */
private function showOrder()
{
    $this->stdout('コマンド?');
    $this->stdout('1.カードを引く');
    $this->stdout('2.カードオープン');
    $stdin = trim(fgets(STDIN));
    switch (true) {
        case $stdin === '1':
            return $this->dealCards();
        case $stdin === '2':
            $this->cardOpen();
            return false;
        default:
            $this->stdout('正しいコマンドを選択してください。');
            return true;
```

```
    }
}

/**
 *
 */
private function cardOpen()
{
    $resultMessage = 'あなたは' . $this->myPoint . '点、ディーラーは' . $this->dealerPoint . '点で、';
    switch (true) {
        case $this->myPoint === $this->dealerPoint:
            $resultMessage .= '引き分けです!';
            break;
        case $this->myPoint === 21:
            $resultMessage .= 'あなたの勝ちです!';
            break;
        case $this->dealerPoint === 21:
            $resultMessage .= 'ディーラーの勝ちです!';
            break;
        case $this->myPoint < 21 && $this->dealerPoint < $this->myPoint:
            $resultMessage .= 'あなたの勝ちです!';
            break;
        case $this->dealerPoint < 21 && $this->myPoint < $this->dealerPoint:
            $resultMessage .= 'ディーラーの勝ちです!';
            break;
        case 21 < $this->myPoint && 21 < $this->dealerPoint:
            $resultMessage .= '引き分けです!';
            break;
        default:
            $resultMessage .= '想定外です!';
            break;
    }
    $this->stdout($resultMessage);
}

/**
 *
 */
private function dealCards(): bool
{
    $this->myPoint += $this->pullCard();
    $this->stdout('あなたの合計値は' . $this->myPoint . 'です。');
    $this->dealerPoint += $this->pullCard();
    switch (true) {
        case $this->myPoint > 21:
            $this->stdout('あなたはバーストしました!');
            return false;
        case $this->dealerPoint > 21:
            $this->stdout('ディーラーはバーストしました!');
            return false;
        default:
            return true;
    }
}

/**
 * @return mixed
 */
private function pullCard()
{
    $suit = $this->suits[rand(0, 3)];
    return array_splice($this->cards[$suit], rand(0, (count($this->cards[$suit]) - 1)), 1)[0];
```

```
    }

    /**
     *
     */
    private function gameSet()
    {
        $this->stdout('ゲーム終了!');
        $this->stdout('お疲れ様でした!');
    }

    /**
     * @param string $text
     */
    private function stdout(string $text)
    {
        echo $text . PHP_EOL;
    }
}
```

解答例

```
<?php

use LaravelJpCon¥q003¥Blackjack;
use PHPUnit¥Framework¥TestCase;

/**
 * Created by PhpStorm.
 * User: fortegp05
 */
class BlackjackTest extends TestCase
{
    private $blackjack;
    private $methodCardOpen;
    private $propertyMyPoint;
    private $propertyDealerPoint;

    /**
     * 初期設定
     */
    public function setUp()
    {
        $this->blackjack = new Blackjack();
        $blackjackReflection = new ¥ReflectionClass($this->blackjack);
        $this->methodCardOpen = $blackjackReflection->getMethod('cardOpen');
        $this->methodCardOpen->setAccessible(true);
        $this->propertyMyPoint = $blackjackReflection->getProperty('myPoint');
        $this->propertyMyPoint->setAccessible(true);
        $this->propertyDealerPoint = $blackjackReflection->getProperty('dealerPoint');
        $this->propertyDealerPoint->setAccessible(true);
    }

    /**
     * 引き分けテスト
     */
    public function test_cardOpenDraw()
    {
        $playerPoint = 21;
        $dealerPoint = 21;
        $this->propertyMyPoint->setValue($this->blackjack, $playerPoint);
        $this->propertyDealerPoint->setValue($this->blackjack, $dealerPoint);
        ob_start();
        $this->methodCardOpen->invoke($this->blackjack);
        $actual = ob_get_clean();
        $this->assertSame('あなたは' . $playerPoint . '点、ディーラーは' . $dealerPoint . '点で、引き分けです!'
. PHP_EOL, $actual, '引き分け');
    }

    /**
     * プレイヤー勝利テスト ブラックジャック
     */
    public function test_cardOpenPlayerWin()
    {
        $playerPoint = 21;
        $dealerPoint = 10;
        $this->propertyMyPoint->setValue($this->blackjack, $playerPoint);
        $this->propertyDealerPoint->setValue($this->blackjack, $dealerPoint);
        ob_start();
        $this->methodCardOpen->invoke($this->blackjack);
        $actual = ob_get_clean();
```

```
        $this->assertSame('あなたは' . $playerPoint . '点、ディーラーは' . $dealerPoint . '点で、あなたの勝ちです!' . PHP_EOL, $actual, '引き分け');
    }

    /**
     * プレイヤー勝利テスト 21に近い
     */
    public function test_cardOpenPlayerWin()
    {
        $playerPoint = 20;
        $dealerPoint = 10;
        $this->propertyMyPoint->setValue($this->blackjack, $playerPoint);
        $this->propertyDealerPoint->setValue($this->blackjack, $dealerPoint);
        ob_start();
        $this->methodCardOpen->invoke($this->blackjack);
        $actual = ob_get_clean();
        $this->assertSame('あなたは' . $playerPoint . '点、ディーラーは' . $dealerPoint . '点で、あなたの勝ちです!' . PHP_EOL, $actual, '引き分け');
    }

    /**
     * ディーラー勝利テスト ブラックジャック
     */
    public function test_cardOpenDealerWin()
    {
        $playerPoint = 10;
        $dealerPoint = 21;
        $this->propertyMyPoint->setValue($this->blackjack, $playerPoint);
        $this->propertyDealerPoint->setValue($this->blackjack, $dealerPoint);
        ob_start();
        $this->methodCardOpen->invoke($this->blackjack);
        $actual = ob_get_clean();
        $this->assertSame('あなたは' . $playerPoint . '点、ディーラーは' . $dealerPoint . '点で、ディーラーの勝ちです!' . PHP_EOL, $actual, '引き分け');
    }

    /**
     * ディーラー勝利テスト 21に近い
     */
    public function test_cardOpenDealerWin()
    {
        $playerPoint = 10;
        $dealerPoint = 20;
        $this->propertyMyPoint->setValue($this->blackjack, $playerPoint);
        $this->propertyDealerPoint->setValue($this->blackjack, $dealerPoint);
        ob_start();
        $this->methodCardOpen->invoke($this->blackjack);
        $actual = ob_get_clean();
        $this->assertSame('あなたは' . $playerPoint . '点、ディーラーは' . $dealerPoint . '点で、ディーラーの勝ちです!' . PHP_EOL, $actual, '引き分け');
    }

}
```

解説
この問題はpublicなメソッドが一つしか無く、
テストしたいプロパティも乱数が入るためテストが書きづらいコードになっています。
こういった場合、ユニットテストを書くことに悩むのではなく、
ユニットテストを描く対象を精査するところから始めましょう。

まず、このブラックジャックゲームは標準入力と標準出力を使用しているため、
すべての入出力をユニットテストで行うのはちょっと面倒です。
そのため、解答例では標準入力は手動テストで行うと割り切って実施していません。
単体テストの問題なのにそれは間違いではないのか?と思われる方がいるかも知れません
が、テストに絶対的な正解はありません。
自分たちのチーム、組織、プロダクトの状態に応じて適切な対応を提案、実施できるのが、
ITエンジニアとしての力量だと思います。
今回は標準入力でテストしたい項目は3つのみ(1、2、それ以外)だったこと、
このゲーム仕様上入力が増えることは無さそうということ、
これらから手動でやると割り切りました。
もちろん、ユニットテストで行うのもOKです。
大事なのは自分たちにあった対応を選択できるか?ということになります。

次に悩むのはprivateのメソッドが多いことです。
全てリフレクションでアクセスする可能にしてテストすることは可能ですが、
その必要があるでしょうか?
また、標準入力を受け付けるために無限ループしているメソッドもあります。
やはりこれもユニットテストとして行うのは面倒そうです。
というわけで、こちらも大半のメソッドを手動テストで行うことで割り切ります。

よって、テストしたい項目としては最小限の出力をテストすることになります。
このゲームの結果は以下のパターンがあります。
- ・引き分け
- ・プレイヤーの勝ち
- ・ディーラーの勝ち

また勝ちのパターンとしても21に到達した場合と、
より21に近いから勝ちという2パターンがあると思います。
よって、以下の5パターンのテストをすれば良さそうです。
- ・引き分け
- ・プレイヤーの勝ち　ブラックジャック
- ・プレイヤーの勝ち　より21に近い
- ・ディーラーの勝ち　ブラックジャック
- ・ディーラーの勝ち　より21に近い

コードとしてはこれ以外のパターンとして異常系も実装していますが、
ゲーム全体でみると通らないパスなので今回はテストしていません。
ここも実装の要否について賛否両論が分かれるところだと思いますが、
今回は意図を明確に伝わりやすくするためにあえて実装しました。

次にテストコードの内容について解説します。
まず、privateなメソッドやプロパティにアクセスするために、
リフレクションでアクセス可能にしましょう。
new ¥ReflectionClass(ClassName)で対象のクラスを取得し、
このインスタンスのgetMethod()、getProperty()で対象のメソッドやプロパティを取得します。
次にsetAccessible(true)でアクセス権限を得ればアクセス可能になります。
メソッドに対してはinvoke($class)で実行できます。
プロパティに対してはsetValue($class, value)で値をセットすることが可能です。

結果は戻り値ではなく標準出力されます。
これを変数として受け取るには、ob_start();と$var = ob_get_clean();で対象の処理を囲んであげる必要があります。
すると$varに標準出力の文字列が代入されます。
その変数に対してassertでテストを書いていきます。