Money Documentation

Release 3.0.0

Mathias Verraes

October 03, 2017

1 Why a Money library for PHP? 3

2 The goal 5
2.1 Getting started . . . 5
2.2 Concept . . . 6
2.3 Inspiration . . . 7
2.4 Operation . . . 7
2.5 Comparison . . . 9
2.6 Allocation . . . 10
2.7 Parsing . . . 11
2.8 Formatting . . . 12
2.9 Currencies . . . 13
2.10 Currency Conversion . . . 15
2.11 Bitcoin . . . 16

Bibliography 19

This library intends to provide tools for storing and using monetary values in an easy, yet powerful way.

CHAPTER 1

Why a Money library for PHP?

Also see http://blog.verraes.net/2011/04/fowler-money-pattern-in-php/

This is a PHP implementation of the Money pattern, as described in *[Fowler2002]* :

> A large proportion of the computers in this world manipulate money, so it's always puzzled me that money isn't actually a first class data type in any mainstream programming language. The lack of a type causes problems, the most obvious surrounding currencies. If all your calculations are done in a single currency, this isn't a huge problem, but once you involve multiple currencies you want to avoid adding your dollars to your yen without taking the currency differences into account. The more subtle problem is with rounding. Monetary calculations are often rounded to the smallest currency unit. When you do this it's easy to lose pennies (or your local equivalent) because of rounding errors.

CHAPTER 2

The goal

Implement a reusable Money class in PHP, using all the best practices and taking care of all the subtle intricacies of handling money.

Getting started

Instantiation

All amounts are represented in the smallest unit (eg. cents), so USD 5.00 is written as

```
use Money\Currency;
use Money\Money;

$fiver = new Money(500, new Currency('USD'));
// or shorter:
$fiver = Money::USD(500);
```

See Parsing for additional ways to instantiate a Money object from strings.

Accepted integer values

The Money object only supports integer(ish) values on instantiation. The following is (not) supported. When a non-supported value is passed a *InvalidArgumentException* will be thrown.

```
use Money\Currency;
use Money\Money;

// int is accepted
$fiver = new Money(500, new Currency('USD'));

// string is accepted if integer
$fiver = new Money('500', new Currency('USD'));

// string is accepted if fractional part is zero
$fiver = new Money('500.00', new Currency('USD'));

// leading zero's are not accepted
$fiver = new Money('00500', new Currency('USD'));

// multiple zero's are not accepted
```

```
$fiver = new Money('000', new Currency('USD'));

// plus sign is not accepted
$fiver = new Money('+500', new Currency('USD'));
```

Installation

Install the library using composer. Execute the following command in your command line.

```
$ composer require moneyphp/money
```

Concept

This section introduces the concept and basic features of the library

Immutability

Jim and Hannah both want to buy a copy of book priced at EUR 25.

```
use Money\Money;

$jimPrice = $hannahPrice = Money::EUR(2500);
```

Jim has a coupon for EUR 5.

```
$coupon = Money::EUR(500);
$jimPrice->subtract($coupon);
```

Because `$jimPrice` and `$hannahPrice` are the same object, you'd expect Hannah to now have the reduced price as well. To prevent this problem, Money objects are **immutable**. With the code above, both `$jimPrice` and `$hannahPrice` are still EUR 25:

```
$jimPrice->equals($hannahPrice); // true
```

The correct way of doing operations is:

```
$jimPrice = $jimPrice->subtract($coupon);
$jimPrice->lessThan($hannahPrice); // true
$jimPrice->equals(Money::EUR(2000)); // true
```

Integer Limit

Although in real life it is highly unprobable, you might have to deal with money values greater than the integer limit of your system (`PHP_INT_MAX` constant represents the maximum integer value).

In order to bypass this limit, we introduced *Calculators*. Based on your environment, Money automatically picks the best internally and globally. The following implementations are available:

- BC Math (requires *bcmath* extension)
- GMP (requires *gmp* extension)
- Plain integer

Calculators are checked for availability in the order above. If no suitable Calculator is found Money silently falls back to the integer implementation.

Because of PHP's integer limit, money values are stored as string internally and `Money::getAmount` also returns string.

```
use Money\Currency;
use Money\Money;

$hugeAmount = new Money('12345678901234567890', new Currency('USD'));
```

Note: Remember, because of the integer limit in PHP, you should inject a string that represents your huge amount.

JSON

If you want to serialize a money object into a JSON, you can just use the PHP method `json_encode` for that. Please find below example of how to achieve this.

```
use Money\Money;

$money = Money::USD(350);
$json = json_encode($money);
echo $json; // outputs '{"amount":"350","currency":"USD"}'
```

Inspiration

- https://github.com/RubyMoney/money
- http://css.dzone.com/books/practical-php-patterns/basic/practical-php-patterns-value
- http://www.codeproject.com/KB/recipes/MoneyTypeForCLR.aspx
- http://stackoverflow.com/questions/1679292/proof-that-fowlers-money-allocation-algorithm-is-correct
- http://timeandmoney.sourceforge.net/
- http://www.joda.org/joda-money/
- http://en.wikipedia.org/wiki/Currency_pair
- https://github.com/RubyMoney/eu_central_bank
- http://en.wikipedia.org/wiki/ISO_4217

Operation

Attention: Operations with Money objects are always immutable. See *Immutability*.

Addition & Subtraction

Additions can be performed using `add()`.

```
$value1 = Money::EUR(800);          // €8.00
$value2 = Money::EUR(500);          // €5.00

$result = $value1->add($value2);    // €13.00
```

Subtractions can be performed using `subtract()`.

```
$value1 = Money::EUR(800);               // €8.00
$value2 = Money::EUR(500);               // €5.00

$result = $value1->subtract($value2);    // €3.00
```

Multiplication & Division

Multiplications can be performed using `multiply()`.

```
$value = Money::EUR(800);        // €8.00

$result = $value->multiply(2);   // €16.00
```

Divisions can be performed using `divide()`.

```
$value = Money::EUR(800);        // €8.00

$result = $value->divide(2);     // €4.00
```

Rounding Modes

A number of rounding modes are available for *Multiplication & Division* above.

- `Money::ROUND_HALF_DOWN`
- `Money::ROUND_HALF_EVEN`
- `Money::ROUND_HALF_ODD`
- `Money::ROUND_HALF_UP`
- `Money::ROUND_UP`
- `Money::ROUND_DOWN`
- `Money::ROUND_HALF_POSITIVE_INFINITY`
- `Money::ROUND_HALF_NEGATIVE_INFINITY`

Absolute Value

`absolute()` provides the absolute value of a Money object.

```
$value = Money::EUR(-800);        // -€8.00

$result = $value->absolute();     // €8.00
```

Comparison

A number of built in methods are available for comparing Money objects.

Same Currency

`isSameCurrency()` compares whether two Money objects have the same currency.

```
$value1 = Money::USD(800);                          // $8.00
$value2 = Money::USD(100);                          // $1.00
$value3 = Money::EUR(800);                          // €8.00

$result = $value1->isSameCurrency($value2);     // true
$result = $value1->isSameCurrency($value3);     // false
```

Equality

`equals()` compares whether two Money objects are equal in value and currency.

```
$value1 = Money::USD(800);                  // $8.00
$value2 = Money::USD(800);                  // $8.00
$value3 = Money::EUR(800);                  // €8.00

$result = $value1->equals($value2);      // true
$result = $value1->equals($value3);      // false
```

Greater Than

`greaterThan()` compares whether the first Money object is larger than the second.

```
$value1 = Money::USD(800);                      // $8.00
$value2 = Money::USD(700);                      // $7.00

$result = $value1->greaterThan($value2);     // true
```

You can also use `greaterThanOrEqual()` to additionally check for equality.

```
$value1 = Money::USD(800);                               // $8.00
$value2 = Money::USD(800);                               // $8.00

$result = $value1->greaterThanOrEqual($value2);      // true
```

Less Than

`lessThan()` compares whether the first Money object is less than the second.

```
$value1 = Money::USD(800);                  // $8.00
$value2 = Money::USD(700);                  // $7.00

$result = $value1->lessThan($value2);    // false
```

You can also use `lessThanOrEqual()` to additionally check for equality.

```
$value1 = Money::USD(800);                          // $8.00
$value2 = Money::USD(800);                          // $8.00

$result = $value1->lessThanOrEqual($value2);     // true
```

Value Sign

You may determine the sign of Money object using the following methods.

- `isZero()`
- `isPositive()`
- `isNegative()`

```
Money::USD(100)->isZero();           // false
Money::USD(0)->isZero();             // true
Money::USD(-100)->isZero();          // false

Money::USD(100)->isPositive();       // true
Money::USD(0)->isPositive();         // false
Money::USD(-100)->isPositive();      // false

Money::USD(100)->isNegative();       // false
Money::USD(0)->isNegative();         // false
Money::USD(-100)->isNegative();      // true
```

Allocation

Allocate by Ratios

My company made a whopping profit of 5 cents, which has to be divided amongst myself (70%) and my investor (30%). Cents can't be divided, so I can't give 3.5 and 1.5 cents. If I round up, I get 4 cents, the investor gets 2, which means I need to conjure up an additional cent. Rounding down to 3 and 1 cent leaves me 1 cent. Apart from re-investing that cent in the company, the best solution is to keep handing out the remainder until all money is spent. In other words:

```
use Money\Money;

$profit = Money::EUR(5);
list($my_cut, $investors_cut) = $profit->allocate([70, 30]);
// $my_cut is 4 cents, $investors_cut is 1 cent

// The order is important:
list($investors_cut, $my_cut) = $profit->allocate([30, 70]);
// $my_cut is 3 cents, $investors_cut is 2 cents
```

Allocate to N targets

An amount of money can be allocated to N targets using `allocateTo()`.

```
$value = Money::EUR(800);            // $8.00

$result = $value->allocateTo(3);     // $result = [$2.67, $2.67, $2.66]
```

Parsing

In an earlier version of Money there was a `Money::stringToUnits` method which parsed strings and created money objects. When the library started to move away from the ISO-only concept, we realized that there might be other cases when parsing from string is necessary. This led us creating parsers and moving the `stringToUnits` to `StringToUnitsParser` (later replaced by `DecimalMoneyParser`).

Money comes with the following implementations out of the box:

Intl Parser

As its name says, this parser requires the *intl* extension and uses `NumberFormatter`. In order to provide the correct subunit for the specific currency, you should also provide the specific currency repository.

> **Warning:** Please be aware that using the *intl* extension can give different results in different environments.

```
use Money\Currencies\ISOCurrencies;
use Money\Parser\IntlMoneyParser;

$currencies = new ISOCurrencies();

$numberFormatter = new \NumberFormatter('en_US', \NumberFormatter::CURRENCY);
$moneyParser = new IntlMoneyParser($numberFormatter, $currencies);

$money = $moneyParser->parse('$1.00');

echo $money->getAmount(); // outputs 100
```

Decimal Parser

This parser takes a simple decimal string which is always in a consistent format independent of locale. In order to provide the correct subunit for the specific currency, you should provide the specific currency repository.

```
use Money\Currencies\ISOCurrencies;
use Money\Parser\DecimalMoneyParser;

$currencies = new ISOCurrencies();

$moneyParser = new DecimalMoneyParser($currencies);

$money = $moneyParser->parse('1000', 'USD');

echo $money->getAmount(); // outputs 100000
```

Aggregate Parser

This parser collects multiple parsers and chooses the most appropriate one based on success to parse. Most parsers throw an exception when the string's format is not supported.

```
use Money\Parser\AggregateMoneyParser;
use Money\Parser\BitcoinMoneyParser;
use Money\Parser\IntlMoneyParser;
```

```
$numberFormatter = new \NumberFormatter('en_US', \NumberFormatter::CURRENCY);
$intlParser = new IntlMoneyParser($numberFormatter, 2);
$bitcoinParser = new BitcoinMoneyParser(2);

$moneyParser = new AggregateParser([
    $intlParser,
    $bitcoinParser,
]);

$dollars = $moneyParser->parse('1 USD');
$bitcoin = $moneyParser->parse("\0xC9\0x831.00");
```

This is very useful if you want to use one parser as a service in DI context.

Bitcoin Parser

See *Bitcoin*.

Formatting

It is often necessary that you display the money value somewhere, probably in a specific format. This is where formatters help you. You can turn a money object into a human readable string.

Money comes with the following implementations out of the box:

Intl Formatter

As its name says, this formatter requires the *intl* extension and uses `NumberFormatter`. In order to provide the correct subunit for the specific currency, you should also provide the specific currency repository.

> **Warning:** Please be aware that using the *intl* extension can give different results in different environments.

```
use Money\Currencies\ISOCurrencies;
use Money\Currency;
use Money\Formatter\IntlMoneyFormatter;
use Money\Money;

$money = new Money(100, new Currency('USD'));
$currencies = new ISOCurrencies();

$numberFormatter = new \NumberFormatter('en_US', \NumberFormatter::CURRENCY);
$moneyFormatter = new IntlMoneyFormatter($numberFormatter, $currencies);

echo $moneyFormatter->format($money); // outputs $1.00
```

Decimal Formatter

This formatter outputs a simple decimal string which is always in a consistent format independent of locale. In order to provide the correct subunit for the specific currency, you should provide the specific currency repository.

```php
use Money\Currencies\ISOCurrencies;
use Money\Currency;
use Money\Formatter\DecimalMoneyFormatter;
use Money\Money;

$money = new Money(100, new Currency('USD'));
$currencies = new ISOCurrencies();

$moneyFormatter = new DecimalMoneyFormatter($currencies);

echo $moneyFormatter->format($money); // outputs 1.00
```

Aggregate Formatter

This formatter collects multiple formatters and chooses the most appropriate one based on currency code.

```php
use Money\Currency;
use Money\Formatter\AggregateMoneyFormatter;
use Money\Formatter\BitcoinMoneyFormatter;
use Money\Formatter\IntlMoneyFormatter;
use Money\Money;

$dollars = new Money(100, new Currency('USD'));
$bitcoin = new Money(100, new Currency('XBT'));

$numberFormatter = new \NumberFormatter('en_US', \NumberFormatter::CURRENCY);
$intlFormatter = new IntlMoneyFormatter($numberFormatter);
$bitcoinFormatter = new BitcoinMoneyFormatter(2);

$moneyFormatter = new AggregateFormatter([
    'USD' => $intlFormatter,
    'XBT' => $bitcoinFormatter,
]);

echo $moneyFormatter->format($dollars); // outputs $1.00
echo $moneyFormatter->format($bitcoin); // outputs 1.00
```

This is very useful if you want to use one formatter as a service in DI context and want to support multiple currencies.

Bitcoin Formatter

See *Bitcoin*.

Currencies

Applications often a certain subset of currencies. Those currencies can from different data sources. Therefore you can implement the *Currencies* interface. The interface provides a list of available currencies and the subunit for the currency.

Money comes with the following implementations out of the box:

ISOCurrencies

As it's name says, the ISO currencies implementation provides all available ISO4217 currencies. It uses the official ISO 4217 Maintenance Agency as source for the data.

```
use Money\Currencies\ISOCurrencies;
use Money\Currency;

$currencies = new ISOCurrencies();
foreach ($currencies as $currency) {
    echo $currency->getCode(); // prints an available currency code within the repository
}

$currencies->contains(new Currency('USD')); // returns boolean whether USD is available in this repos
$currencies->subunitFor(new Currency('USD')); // returns the subunit for the dollar (2)
```

BitcoinCurrencies

The Bitcoin currencies provides a single currency: the Bitcoin. It uses XBT as its code and has a subunit of 8.

```
use Money\Currencies\BitcoinCurrencies;
use Money\Currency;

$currencies = new BitcoinCurrencies();
foreach ($currencies as $currency) {
    echo $currency->getCode(); // prints XBT
}

$currencies->contains(new Currency('XBT')); // returns boolean whether XBT is available in this repos
$currencies->contains(new Currency('USD')); // returns boolean whether USD is available in this repos
$currencies->subunitFor(new Currency('XBT')); // returns the subunit for the Bitcoin (8)
```

Aggregate Currencies

This formatter collects multiple currencies.

```
use Money\Currency;
use Money\Currencies\AggregateCurrencies;
use Money\Currencies\BitcoinCurrencies;
use Money\Currencies\ISOCurrencies;

$currencies = new AggregateCurrencies([
    new BitcoinCurrencies(),
    new ISOCurrencies()
]);

foreach ($currencies as $currency) {
    echo $currency->getCode(); // prints XBT or any ISO currency code
}

$currencies->contains(new Currency('XBT')); // returns boolean whether XBT is available in this repos
$currencies->contains(new Currency('USD')); // returns boolean whether USD is available in this repos
$currencies->subunitFor(new Currency('XBT')); // returns the subunit for the Bitcoin (8)
```

This is very useful if you want to support multiple currencies data sources.

Currency Conversion

To convert a Money instance from one Currency to another, you need the Converter. This class depends on Currencies and Exchange. Exchange returns a *CurrencyPair*, which is the combination of the base currency, counter currency and the conversion ratio.

Fixed Exchange

You can use a fixed exchange to convert *Money* into another Currency.

```
use Money\Converter;
use Money\Currency;
use Money\Exchange\FixedExchange;

$exchange = new FixedExchange([
    'EUR' => [
        'USD' => 1.25
    ]
]);

$converter = new Converter(new ISOCurrencies(), $exchange);

$eur100 = Money::EUR(100);
$usd125 = $converter->convert($eur100, new Currency('USD'));
```

Reversed Currencies Exchange

In some cases you might want the Exchange to resolve the reverse of the Currency Pair as well if the original cannot be found. To add this behaviour to any Exchange you need to wrap it in in a *ReversedCurrenciesExchange*. If a reverse Currency Pair can be found, it's simply used as a divisor of 1 to calculate the reverse conversion ratio.

For example this can be useful if you use a *FixedExchange* and you don't want to define the currency pairs in both directions.

```
use Money\Converter;
use Money\Currency;
use Money\Exchange\FixedExchange;
use Money\Exchange\ReversedCurrenciesExchange;

$exchange = new ReversedCurrenciesExchange(new FixedExchange([
    'EUR' => [
        'USD' => 1.25
    ]
]));

$converter = new Converter(new ISOCurrencies(), $exchange);

$usd125 = Money::USD(125);
$eur100 = $converter->convert($usd125, new Currency('EUR'));
```

Third Party Exchange

We also provide a way to integrate external sources of conversion rates by implementing the `Money\Exchange` interface. There is a default one in the core using Swap which you can install via Composer:

```
$ composer require florianv/swap
```

Then conversion is quite simple:

```
use Money\Money;
use Money\Converter;

// $swap = Implementation of \Swap\SwapInterface
$exchange = new SwapExchange($swap);

$converter = new Converter(new ISOCurrencies(), $exchange);
$eur100 = Money::EUR(100);
$usd125 = $converter->convert($eur100, new Currency('USD'));
```

CurrencyPair

A CurrencyPair is returned by the Exchange. If you want to implement your own Exchange, you can use the OOP notation to define a pair:

```
use Money\Currency;
use Money\CurrencyPair;

$pair = new CurrencyPair(new Currency('EUR'), new Currency('USD'), 1.2500);
```

But you can also parse ISO notations. For example, the quotation `EUR/USD 1.2500` means that one euro is exchanged for 1.2500 US dollars.

```
use Money\CurrencyPair;

$pair = CurrencyPair::createFromIso('EUR/USD 1.2500');
```

You could also create a pair using a third party. There is a default one in the core using Swap which you can install via Composer.

```
use Money\Currency;
use Money\Exchange\SwapExchange;

$eur = new Currency('EUR');
$usd = new Currency('USD');

// $swap = Implementation of \Swap\SwapInterface
$exchange = new SwapExchange($swap);

$pair = $exchange->quote($eur, $usd);
```

Bitcoin

Since Money is not ISO currency specific, you can construct a currency object by using the code XBT. For Bitcoin there is also a formatter and a parser available. The subunit is 8 for a Bitcoin.

Please see the example below how to use the Bitcoin currency:

```
use Money\Currencies\BitcoinCurrencies;
use Money\Currency;
use Money\Formatter\BitcoinMoneyFormatter;
```

```php
use Money\Money;
use Money\Parser\BitcoinMoneyParser;

// construct bitcoin (subunit of 8)
$money = new Money(100000000000, new Currency('XBT'));

// construct bitcoin currencies
$currencies = new BitcoinCurrencies();

// format bitcoin
$formatter = new BitcoinMoneyFormatter(2, $currencies);
echo $formatter->format($money); // prints 1000.00

// parse bitcoin
$parser = new BitcoinMoneyParser(2);
$money = $parser->parse("\0xC9\0x831000.00", 'XBT');
echo $money->getAmount(); // outputs 100000000000
```

In most cases you probably don't know the exact currency you are going to format or parse. For such scenarios, we have an aggregate formatter and a parser which lets you configure multiple parsers and then choose the best based on the value. See more in *Formatting* and *Parsing* section.

[Fowler2002] Fowler, M., D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002. http://martinfowler.com/books.html#eaa