Datetime

Philip Michael Raab

Version 0.4.0, 2025-05-27

Table of Contents

version: Id ($Date$)

Things to help you bend space and time to your will. Or at least do some calculations with it.

Install

```
$ composer require inanepain/datetime
```

Datetime

This is meant as more of an introduction to the various classes, interfaces, enums and anything else found in the `Inane\Datetime` namespace and not an in-depth guide.

Interfaces

- TimeWrapper

Traits

- Timescale

Enums

- TimeTrait

Classes

- Timespan
- Timestamp

.1. TimeWrapper

The `TimeWrapper` **interface** implemented by `Timespan` and `Timestamp` (possibly more to come), started more as a convenience than actually any real requirement. Soon after implementing this interface the usage of the `Datetime` classes increased, thanks to the greater interoperability they had acquired. This in turn resulted in them quickly evolving from the skinny classes they were to what they are now, skinny but with some meat on them. Enough rambling and there's nothing left to say about it anyway.

.1.1. Interface Methods

There are *three* **interface** methods shared by `Datetime` classes:

interface methods

- public function getSeconds(): int
- public function format(string $format = 'Y-m-d H:i:s'): string
- public function absoluteCopy(): Timestamp

getSeconds: int

At the core it's all numbers and for these numbers it's the seconds returned by this method.

method: getSeconds

```
public function getSeconds(): int;
```

---

format: string

Used to get the object as a string using a custom format pattern. The actual implementations may use different format options best suited to the type of data they contain. Where possible patterns mimic existing php patterns.

- `Timestamp`: `\DateTimeInterface::format()`
- `Timespan`: `\DateTimeInterface::format()`

The method uses the same formatting pattern as `date` and `DateTime`, as well as numerous other php functions, to return a string representation of the object.

method: format

```
public function format(string $format = ''): string;
```

---

absoluteCopy: TimeWrapper

Get a copy of the object with an absolute value.

method: absoluteCopy

```
public function absoluteCopy(): ((TimeWrapper));
```

.1.2. Format

Table 1.A subset of format options commonly used by `Datetime` classes.

Format character	Description	Example returned values
Day	---	---
`d`	Day of the month, 2 digits with leading zeros	`01 to 31`
`D`	A textual representation of a day, 3 letters	`Mon through Sun`
`j`	Day of the month no leading zeros	`1 to 31`
`l (lowercase 'L')`	A full textual representation of the day of the week	`Sunday through Saturday`
Month	---	---
`F`	A full textual representation of a month	`January through December`
`m`	Numeric representation of a month, with leading zeros	`01 through 12`
`M`	A 3 letter textual representation of a month	`Jan through Dec`
`n`	Numeric representation of a month, no leading zeros	`1 through 12`
Year	---	---
`Y`	Full numeric year, at least 4 digits, - for BCE	`Examples: -0055, 0787, 1999, 2003, 10191`
Time	---	---
`a`	Lowercase Ante meridiem and Post meridiem	`am or pm`
`A`	Uppercase Ante meridiem and Post meridiem	`AM or PM`
`g`	12-hr format of an hour no leading zeros	`1 through 12`
`G`	24-hr format of an hour no leading zeros	`0 through 23`
`h`	12-hr format of an hour with leading zeros	`01 through 12`
`H`	24-hr format of an hour with leading zeros	`00 through 23`
`i`	Minutes with leading zeros	`00 to 59`

Format character	Description	Example returned values
s	Seconds with leading zeros	00 through 59
Full date/time	---	---
c	ISO 8601 date	2004-02-12T15:19:21+00:00
r	» RFC 2822/» RFC 5322 formatted date	Example: Thu, 21 Dec 2000 16:01:07 +0200

.2. TimeTrait

The `TimeTrait` **trait** simple adds some properties for the three timestamps: **seconds**, **milliseconds** and **microseconds**.

.2.1. Properties

There are *three* **properties** added by `TimeTrait`:

trait properties

- public private(set) int $seconds
- public private(set) int $milliseconds
- public private(set) int $microseconds

Only `$microseconds` actually stores a value while the `$milliseconds` and `$seconds` calculate their value from it.

$seconds (int)

Returns a timestamp the has 10 digits.

example

```
1746045170
```

$milliseconds (int)

Returns a timestamp the has 13 digits.

example

```
1746045170733
```

$microseconds (int)

Returns a timestamp the has 16 digits.

example

```
1746045170733444
```

.3. Timescale

The `Timescale` **enum** simply defines three **timestamps** of various length: **seconds** (10 digits), **milliseconds** (13 digits) and **microseconds** (16 digits).

.3.1. Cases

- SECOND
  - 10 digits
- MILLISECOND
  - 13 digits
- MICROSECOND
  - 16 digits

.3.2. Methods

class methods

- public static function **tryFromName**(string $name, bool $ignoreCase = false): ?static
- public static function **tryFromTimestamp**(int|Timestamp $timestamp): ?Timescale

instance methods

- public function **timestamp**(bool $asObject = false): int|Timestamp
- public function **unit**(): string

.4. Timespan

The `Timespan` **class** represents a *timespan* or *duration*(fn-duration) which is essentially a number of seconds. As with most things, seconds can be expressed in various formats to suite the situation, for instance `5400` and `2years 3d 7secs`, `Timespan` makes this easy for you.

Quickly get a *moment*[1] in the *future* or *past* by **adding** or **subtracting** a `Timespan` from a `Timestamp`. Working together these two classes have you covered in the *past*, *present*, *future* or anytime in *between*.

The Timespan as is a period of time or duration and a Timestamp[2] is a moment or point in time.

.4.1. Notes on Methods

The methods `format` and `getDuration` are similar but there are bigger differences between the two than that `format` allows for a far great level of customising the output string. The main difference is a big one with far-reaching consequences if not understood correctly. The `format` method is a pure display method that will only substitute existing values into the template string. Hence, all it takes as a parameter is the template string. The `duration` method calculates the duration using the supplied unit of measurement. The only formatting options it has is weather to show the unit of time: character, abbreviated, word.

.4.2. Moments and Durations

This covers a basic overview of different `Timespan` formats and types.

Time Durations

A length of time, like **5 minutes** which is the same as a **timespan 300**.

Table 2. Supported Time Durations

Name	Type	Example
timespan	int (seconds)	`5400`
duration	string	`2years 3d 7secs`
DateInterval	class	`new DateInterval('PT300S')`

Moments in Time

A specific point in time, like **01 January 1970 02:00:00 AM SAST** which is the same as the unix timestamp **0**.

A **unix timestamp** or **epoc time** while being a moment in time as also a **duration**, the number of seconds since the **epoch** (**0**).

Table 3. Supported Date Times

Name	Type	Example
timestamp	seconds	`236988000`
formatted date	string	`06 July 1977 12:00:00 AM SAST`
DateTime(Immutable)	class	`new DateTime('NOW')`

.4.3.Examples

example

```
// Create with current seconds since epoch
$epoch = new \Inane\Datetime\Timespan(time());
echo("Time since epoch: $epoch\n"); // 52y 32w 1d 6h 13i 1s (not what you got! lol.)

// Create from duration
$ts1 = \Inane\Datetime\Timespan::fromDuration('3 yrs 2w 2days 4 min');

// Automatic string conversion
echo("$ts1");                    // 3y 2w 2d 4i

echo($ts1->getSeconds());       // 96053496 (seconds)
// Timestamp uses now as it's point of reference
echo($ts1->getTimestamp());      // 1756477381 (added timespan to now)
echo($ts1->getTimestamp(false)); // 1564370389 (subtracted timespan from now)

// default format: Y-m-d H:i:s
echo($ts1->format());                    // 2025-08-29 16:23:01
echo($ts1->format('', false) . "\n");   // 2019-07-29 05:19:49

// Another Timespan
$ts2 = new \Inane\Datetime\Timespan(8600);
echo("Symbol Format:");
echo("Default:\t" . $ts2->getDuration());
// 2hrs 23mins 20secs
echo("Char:\t\t" . $ts2->getDuration(\Inane\Datetime\Timespan::SYMBOL_CHAR));
// 2h 23i 20s (not the `i` for min char.)
echo("Abbreviated:\t" . $ts2-
>getDuration(\Inane\Datetime\Timespan::SYMBOL_ABBREVIATED));   // 2hrs 23mins 20secs
echo("Word:\t\t" . $ts2->getDuration(\Inane\Datetime\Timespan::SYMBOL_WORD) . "\n");
// 2hours 23minutes 20seconds

// Static conversion functions
$s1 = \Inane\Datetime\Timespan::dur2ts('1hr 30min');    // 10800
echo("1hr 30min == $s1");

$d1 = \Inane\Datetime\Timespan::ts2dur(10800);          // 1hr 30min
echo("10800 == $d1");
// OR
echo "$s1 == $d1";                                      // 10800 == 1hr 30min
```

.5. Timestamp

The `Timestamp` **class** is a truly simple wrapper for an **epoch timestamp**. It's mainly useful when used in combination with a `Timespan` to do chained date calculations and then displaying the formatted result. Essentially it saves typing a few lines of code here and there and I think it looks a tad neater too. The original bit of code was primarily used as a convenience class to easily switch between various date and time structures.

Enough chatter, here comes it's breakdown.

.5.1. Properties

There are *three* **properties** added by `TimeTrait`:

trait properties

- public private(set) int $seconds
- public private(set) int $milliseconds
- public private(set) int $microseconds

Only `$microseconds` actually stores a value while the `$milliseconds` and `$seconds` calculate their value from it.

.5.2. Methods

There are *three* **interface** methods shared by `Datetime`` classes:

interface methods

- public function getSeconds(): int
- public function format(string $format = 'Y-m-d H:i:s'): string
- public function absoluteCopy(): Timestamp

And *six* class methods.

class methods

- public function __construct(?int $timestamp = null)
- public static function createFromFormat(string $format, string $datetime): static|false
- public static function now(): int
- public function getDateTime(bool $immutable = false): DateTime|DateTimeImmutable
- public function adjust(int|Timespan $timespan): self
- public function diff(int|Timestamp $timestamp): Timespan

Create / New

In addition to instantiating a `Timestamp` using `new` the static method `createFromFormat` can also be used to create a `Timestamp` instance. The constructor only takes a unix timestamp, for all other values the create method is used.

Creating Timestamps: default and custom values.

```
$now = new Timestamp(); ①
$then = Timestamp::createFromFormat('g:ia \o\n l jS F Y', '12:00am on Wednesday 6th
July 1977');
```

① creating a new Timestamp without any parameters uses the current unix time

Get / Show

Once you have a Timestamp it's easy to retrieve the date as various types that best suite the situation.

datetime types

- string
- int
- DateTime/DateTimeImmutable

Getting values to work with or display.

```
$user->setAnniversary($then->getSeconds()); ①
echo "The time between $now and $then.", PHP_EOL; ②
$tokyoTime = $now->getDateTime()->setTimezone($timeZone); ③
```

① Add to entity as int to store in database

② as a string to show on screen

③ and as a DateTime to work with a DateTimeZone

Calculate

A Timestamp can also work directly on DateTime to do calculations. When getting the difference between two Timestamps the result will be positive when the end Timestamp is greater than the source/initial Timestamp. Using `$now→diff($then)`, we can read it as; how much time needs to be added to `$now` to get to `$then`. If `$now` is today and `$then` is yesterday, we need to add a negative day; `-1`. But if `$then` were tomorrow, it would be the same distance `1`, but this time positive.

If it's the just size of the gap between Timestamps that's needed the `absoluteCopy` method can be used on the result or use the format method and ignore sign format symbol.

Time Manipulating

```
$between = $now->diff($then); ①
echo $between, PHP_EOL; ②
echo $between->format('Formatted time between: %r%y years and %m months'), PHP_EOL; ③
```

① using the variables we created before

② when this document was drafted, the result was: - 45yrs 3months 3weeks 6hrs 14mins 29secs

③ Formatted time between: -45 years and 3 months

Comparing Timestamps

Figuring out what came first, the chicken or the egg, might be an unending debate. But with a `Timestamp` is just as easy as a **timestamp**, simply use the usual *great* (>), *less* (<) or *equals* (==) comparators.

What came first, chicken or egg?

```
$chicken = new \Inane\Datetime\Timestamp(-27106883520);
$egg = \Inane\Datetime\Timestamp::createFromFormat('d F Y g:i:s', '01 January 1111
00:00:00');

if ($chicken < $egg) echo "The chicken preceded the egg!", PHP_EOL; ①
else if ($egg < $chicken) echo "The egg preceded the chicken!", PHP_EOL; ①
else if ($egg == $chicken) echo "Apparently the chicken and the egg arrived
together!", PHP_EOL; ②
else echo "Something impossible has happened!!!", PHP_EOL;
```

① greater or less than works as normal, using the seconds for comparison

② same goes for equality

[1] **moment:** exact point in time.

[2] **timestamp:** a.k.a. unix time, epoch time, posix time and various combinations there of.

Glossary

Timespan

a duration of time, a period of time, a length of time

Timestamp

a moment in time, a point in time, a date and time

Timescale

a unit of time, a scale of time, a measurement of time

seconds

10 digits

milliseconds

13 digits

microseconds

16 digits

Index