Chapter 6. Dates and Times

Manipulating dates and times is one of the most complicated tasks you can do in any language, let alone in PHP. This is simply because time is relative—*now* will differ from one user to the next and potentially trigger different behavior in your application.

Object Orientation

PHP developers will work primarily with `DateTime` objects in code. These objects work by wrapping a particular instance in time and provide a wide variety of functionality. You can take the differences between two `DateTime` objects, convert between arbitrary time zones, or add/subtract windows of time from an otherwise static object.

Additionally, PHP supports a `DateTimeImmutable` object which is functionally identical to `DateTime` but cannot be modified directly. Most methods on a `DateTime` object will both return the same object and mutate its internal state. The same methods on `DateTimeImmutable` leave the internal state in place but return *new instances* representing the result of the change.

NOTE

Both date/time classes extend an abstract `DateTimeInterface` base class, making the two classes nearly interchangeable within PHP's date and time functionality. Everywhere you see `DateTime` in this chapter you could use a `DateTimeImmutable` instance instead and achieve similar if not identical functionality.

Time Zones

One of the most challenging problems any developer will face is working with time zones, particularly when daylight saving time is involved. On the one

hand, it's easy to simplify and assume every timestamp within an application is referencing the same time zone. This is rarely true.

Luckily, PHP makes handling time zones remarkably easy. Every `DateTime` has a time zone embedded automatically, usually based on the default defined within the system on which PHP is running. You can also explicitly set a time zone whenever you create a `DateTime` making the region and time you're referencing entirely unambiguous. Converting between time zones is also simple and powerful and covered at length in Recipe 6.9.

Unix Timestamps

Many computer systems use Unix timestamps internally to represent dates and times. These timestamps represent the number of seconds that have occurred between the Unix Epoch (January 1, 1970 at 00:00:00 GMT) and a given time. They are memory-efficient and frequently used by databases and programmatic APIs. However, counting the number of seconds since a fixed date/time isn't exactly user-friendly, so you need a reliable way to convert between Unix timestamps and human-readable date/time representations within your applications.

PHP's native formatting capabilities make this straightforward. Additional functions, like `time()`, produce Unix timestamps directly as well.

The following recipes cover these topics at length, in addition to several other common date/time-related tasks.

6.1 Finding the Current Date and Time

Problem

You want to know the current date and time.

Solution

To print the current date and time following a particular format, use `date()`. For example:

```
$now = date('r');
```

The output of `date()` depends on the system it's being run on and the current actual time. Using `r` as a format string, this function would return something like the following:

```
Wed, 09 Nov 2022 14:15:12 -0800
```

Similarly, a newly instantiated `DateTime` object will also represent the current date and time. The `::format()` method on this object exhibits the same behavior as `date()`, meaning the following two statements are functionally identical:

```
$now = date('r');

$now = (new DateTime())->format('r');
```

Discussion

PHP's `date()` function, as well as a `DateTime` object instantiated with no parameters, will automatically inherit the current date and time of the system on which they're run. The additional `r` passed into both is a format character that defines how to convert the given date/time information into a string—in this case, specifically as a date formatted according to RFC 2822. You can learn more about date formatting in Recipe 6.2.

A powerful alternative is to leverage PHP's `getdate()` function to retrieve an associative array of all of the parts of the current system date and time. This array will contain the keys and values in Table 6-1.

Table 6-1. Key elements returned by `getdate()`

Key	Description of value	Example
`seconds`	Seconds	`0` to `59`
`minutes`	Minutes	`0` to `59`

Key	Description of value	Example
`hours`	Hours	`0` to `23`
`mday`	Day of the month	`1` through `31`
`wday`	Day of the week	`0` (Sunday) through `6` (Saturday)
`mon`	Month	`1` through `12`
`year`	Full, four-digit year	`2023`
`yday`	Day of the year	`0` through 365
`weekday`	Day of the week	`Sunday` through `Saturday`
`month`	Month of the year	`January` through `December`
`0`	Unix timestamp	`0` to `2147483647`

In some applications, you might only need the day of the week rather than a fully operational `DateTime` object. Consider Example 6-1, which illustrates how you might achieve this with either `DateTime` or `getdate()`.

Example 6-1. Comparing `DateTime` with `getdate()`

```
print (new DateTime())->format('l') . PHP_EOL;

print getdate()['weekday'] . PHP_EOL;
```

These two lines of code are functionally equivalent. For a simple task such as "print today's date," either would be adequate for the job. The `DateTime` object provides functionality for converting time zones or forecasting future dates (both of which are covered further in other recipes). The associative array returned by `getdate()` lacks this functionality but makes up for that shortcoming through its simple, easy-to-recognize array keys.

See Also

PHP documentation on date and time functions, the `DateTime` class, and the `getdate()` function.

6.2 Formatting Dates and Times

Problem

You want to print a date to a string in a particular format.

Solution

Use the `::format()` method on a given `DateTime` object to specify the format of the returned string, as shown in Example 6-2.

Example 6-2. Date and time format examples

```
$birthday = new DateTime('2017-08-01');

print $birthday->format('l, F j, Y') . PHP_EOL;
                                    ❶

print $birthday->format('n/j/y') . PHP_EOL;
                                    ❷

print $birthday->format(DateTime::RSS) . PHP_EOL;
                                    ❸
```

```
Tuesday, August 1, 2017 ❶
8/1/17                  ❷
Tue, 01 Aug 2017 00:00:00 +0000
```

Discussion

Both the `date()` function and the `DateTime` object's `::format()` method accept a variety of input strings that ultimately define the final structure of the string produced by PHP. Each format string is composed of individual characters that represent specific parts of a date or time value, as you can see in Table 6-2.

Table 6-2. PHP format characters

Character	Description	Example values
Day		
`d`	Day of the month, two digits with leading 0	`01` to `31`
`D`	A textual representation of a day, three letters	`Mon` through `Sun`
`j`	Day of the month without leading 0	`1` to `31`
`l`	The name of the day of the week	`Sunday` through `Saturday`
`N`	ISO 8601 numeric representation of the day of the week	`1` (for Monday) through `7` (for Sunday)
`S`	English ordinal suffix for the day of the month, two characters	`st`, `nd`, `rd`, or `th`. Works well with `j`
`w`	Numeric representation of the day of the week	`0` (for Sunday) through `6` (for Saturday)
`z`	The day of the year (starting from 0)	`0` through `365`
Month		
`F`	The full name of the month	`January` through `December`
`m`	Numeric representation of a month, with leading 0	`01` through `12`

Character	Description	Example values
`M`	A textual representation of a month, three letters	`Jan` through `Dec`
`n`	Numeric representation of a month, without leading 0	`1` through `12`
`t`	Number of days in the given month	`28` through `31`
Year		
`L`	Whether it's a leap year	`1` if it is a leap year, `0` otherwise.
`o`	ISO 8601 week-numbering year. This has the same value as `Y`, except that if the ISO week belongs to the previous or next year, that year is used instead	`1999` or `2003`
`Y`	A full numeric representation of a year, four digits	`1999` or `2003`
`y`	A two-digit representation of a year	`99` or `03`
Time		
`a`	Lowercase ante meridiem or post meridiem	`am` or `pm`
`A`	Uppercase ante meridiem or post meridiem	`AM` or `PM`

Character	Description	Example values
`g`	12-hour format of an hour without leading 0	`1` through `12`
`G`	24-hour format of an hour without leading 0	`0` through `23`
`h`	12-hour format of an hour with leading 0	`01` through `12`
`H`	24-hour format of an hour with leading 0	`00` through `23`
`i`	Minutes with leading 0	`00` to `59`
`s`	Seconds with leading 0	`00` through `59`
`u`	Microseconds	`654321`
`v`	Milliseconds	`654`
Time zone		
`e`	Time zone identifier	`UTC`, `GMT`, `Atlantic/Azores`
`I`	Whether the date is in daylight saving time	`1` if daylight saving time, `0` otherwise.
`O`	Difference from Greenwich time (GMT) without colon between hours and minutes	`+0200`
`P`	Difference from Greenwich time (GMT) with colon between hours and minutes	`+02:00`

Character	Description	Example values
p	The same as P, but returns Z instead of +00:00	+02:00
T	Time zone abbreviation, if known; otherwise the GMT offset.	EST , MDT , +05
Z	Time zone offset in seconds	-43200 through 50400
Other		
U	Unix timestamp	0 through 2147483647

Combining these characters into a format string determines exactly how PHP will convert a given date/time construct into a string.

Similarly, PHP defines several predefined constants representing well-known and widely used formats. Table 6-3 shows some of the most useful.

Table 6-3. Predefined date constants

Constant	Class constant	Format characters
DATE_ATOM	DateTime::ATOM	Y-m-d\TH:i:sP
DATE_COOKIE	DateTime::COOKIE	l, d-M-Y H:i:s T
DATE_ISO8601`footnote: [Unfortunately, `DATE_ISO8601 isn't compatible with the ISO 8601 standard. If you need that	DateTime::ISO8601	Y-m-d\TH:i:sO

Constant	Class constant	Format characters
level of compatibility, use `DATE_ATOM` instead.]		
`DATE_RSS`	`DateTime::RSS`	`D, d M Y H:i:s O`

See Also

Full documentation on format characters and predefined `DateTime` constants.

6.3 Converting Dates and Times to Unix Timestamps

Problem

You want to convert a particular date or time to a Unix timestamp and convert a given Unix timestamp into a local date or time.

Solution

To convert a given date/time into a timestamp, use the `U` format character (see Table 6-2) with `DateTime::format()` as follows:

```
$date = '2023-11-09T13:15:00-0700';
$dateObj = new DateTime($date);

echo $dateObj->format('U');
// 1699560900
```

To convert a given timestamp into a `DateTime` object, also use the `U` format character but instead with `DateTime::createFromFormat()` as follows:

```
$timestamp = '1648241792';
$dateObj = DateTime::createFromFormat('U', $timestamp);

echo $dateObj->format(DateTime::ISO8601);
// 2022-03-25T20:56:32+0000
```

Discussion

The `::createFromFormat()` method is a static inverse of `DateTime`'s `::format()` method. Both functions use identical format strings to specify the format being used[1] but represent opposite transformations between a formatted string and the underlying state of a `DateTime` object. The Solution example explicitly leverages the `U` format character to tell PHP that the input data is a Unix timestamp.

If the input string doesn't actually match your format, PHP will return a literal `false` as in the following example:

```
$timestamp = '2023-07-23';
$dateObj = DateTime::createFromFormat('U', $timestamp);

echo gettype($dateObj); // false
```

When parsing user input, it is a good idea to explicitly check the return of `::createFromFormat()` to ensure that the date input was valid. For more on validating dates, see Recipe 6.7.

Rather than work with a full `DateTime` object, you can work with *parts* of a date/time directly. PHP's `mktime()` function will always return a Unix timestamp, and the only required parameter is the hour.

For example, assume you want the Unix timestamp representing July 4, 2023 at noon in GMT (no time zone offset). You could do this in two ways, as demonstrated in Example 6-3.

Example 6-3. Creating a timestamp directly

```
$date = new DateTime('2023-07-04T12:00:00');
$timestamp = $date->format('U');
```

❶

```
$timestamp = mktime(month: 7, day: 4, year: 2023, hour:
```

❷

This output will be exactly `1688472000` .

This output will be *close* to `1688472000` but will vary in the last three digits.

While this simpler example appears elegant and avoids instantiating an object only to turn it back into a number, it has an important problem. Failing to specify a parameter (in this case, minutes or seconds) will cause `mktime()` to use the current system values for those parameters by default. If you were to run this example code at 3:05 in the afternoon, the output might be `1688472300` .

This Unix timestamp translates to 12:05:00 rather than 12:00:00 when converted back to a `DateTime` , representing a (potentially negligible) difference from what the application expects.

It's important to remember that, if you choose to leverage the functional interface of `mktime()` , you either provide a value for *every* component of the date/time or you build your application in such a way that slight deviations are expected and handled.

See Also

Documentation on `DateTime::createFromFormat()` .

6.4 Converting from Unix Timestamps to Date and Time Parts

Problem

You want to extract a particular date or time part (day or hour) from a Unix timestamp.

Solution

Pass the Unix timestamp as a parameter to `getdate()` and reference the required keys in the resulting associative array. For example:

```
$date = 1688472300;
$time_parts = getdate($date);

print $time_parts['weekday'] . PHP_EOL; // Tuesday
print $time_parts['hours'] . PHP_EOL;   // 12
```

Discussion

The only parameter you can provide to `getdate()` is a Unix timestamp. If this parameter is omitted, PHP will leverage the current system date and time. When you provide a timestamp, PHP parses that timestamp internally and allows for the extraction of all expected date and time elements.

Alternatively, you can pass a timestamp into the constructor for a `DateTime` instance in two ways to build a full object from it:

1. Prefixing the timestamp with an `@` character tells PHP to interpret the entry as a Unix timestamp—for example, `new DateTime('@1688472300')`.
2. You can use the `U` format character when importing a timestamp into a `DateTime` object—for example, `DateTime::createFromFormat('U', '1688472300')`.

In any case, once your timestamp is properly parsed and loaded into a `DateTime` object, you can use its `::format()` method to extract any component you desire. Example 6-4 is an alternative implementation of the Solution example that leverages `DateTime` rather than `getdate()`.

Example 6-4. Extracting date and time parts from Unix timestamps

```
$date = '1688472300';

$parsed = new DateTime("@{$date}");
print $parsed->format('l') . PHP_EOL;
print $parsed->format('g') . PHP_EOL;
```

```
$parsed2 = DateTime::createFromFormat('U', $date);
print $parsed2->format('l') . PHP_EOL;
print $parsed2->format('g') . PHP_EOL;
```

Either of the approaches in Example 6-4 is a valid replacement of `getdate()` that also provides the benefit of giving you a fully functional `DateTime` instance. You could print the date (or time) in any format, manipulate the underlying value directly, or even convert between time zones if necessary. Each of these potential further uses for `DateTime` is covered in further recipes.

See Also

Recipe 6.1 for further discussion of `getdate()`. Read ahead in Recipe 6.8 to learn how to manipulate `DateTime` objects and in Recipe 6.9 to see how time zones can be managed directly.

6.5 Computing the Difference Between Two Dates

Problem

You want to find out how much time has passed between two dates or times.

Solution

Encapsulate each date/time in a `DateTime` object. Leverage the `::diff()` method on one to calculate the relative difference between it and the other `DateTime`. The result will be a `DateInterval` object as follows:

```
$firstDate = new DateTime('2002-06-14');
$secondDate = new DateTime('2022-11-09');

$interval = $secondDate->diff($firstDate);

print $interval->format('%y years %d days %m months');
// 20 years 25 days 4 months
```

Discussion

The `::diff()` method of the `DateTime` object effectively subtracts one date/time (the argument passed into the method) from another (the one represented by the object itself). The result is a representation of the relative duration of time between the two objects.

WARNING

The `::diff()` method ignores daylight saving time. To properly account for the potential one-hour difference intrinsic to that system, converting both date/time objects into UTC first is a good idea.

It is also important to note that, while it might appear similar in the Solution example, the `::format()` method of the `DateInterval` object takes a whole different set of format characters than those used by `DateTime`. Every format character must be prefixed by a literal `%` character, but the format string itself can include nonformatting characters (like *years* and *months* in the Solution example).

Available format characters are enumerated in Table 6-4. In every case except for the format characters of `a` and `r`, using the lowercase for a format character will return a numeric value without any leading 0. The enumerated uppercase format characters return at least two digits with a leading 0. Remember, every format character must be prefixed with a literal `%`.

Table 6-4. `DateInterval` format characters

Character	Description	Example
`%`	Literal `%`	`%`
`Y`	Years	`03`
`M`	Months	`02`
`D`	Days	`09`

Character	Description	Example
`H`	Hours	`08`
`I`	Minutes	`01`
`S`	Seconds	`04`
`F`	Microseconds	`007705`
`R`	Sign “-” when negative, “+” when positive	`-` or `+`
`r`	Sign “-” when negative, empty when positive	`-`
`a`	Total number of days	`548`

See Also

Full documentation on the `DateInterval` class.

6.6 Parsing Dates and Times from Arbitrary Strings

Problem

You need to convert an arbitrary, user-defined string into a valid `DateTime` object for further use or manipulation.

Solution

Use PHP’s powerful `strtotime()` function to convert the text entry into a Unix timestamp, and then pass that into the constructor of a new `DateTime` object. For example:

```
$entry = strtotime('last Wednesday');
$parsed = new DateTime("@{$entry}");

$entry = strtotime('now + 2 days');
$parsed = new DateTime("@{$entry}");

$entry = strtotime('June 23, 2023');
$parsed = new DateTime("@{$entry}");
```

Discussion

The power of `strtotime()` comes from the underlying date and time import formats supported by the language. These include the kinds of formats you might expect computers to use (like YYYY-MM-DD for a year, month, and day). But it extends to *relative* specifiers and complex, compound formats as well.

NOTE

The convention of prefixing a Unix timestamp with a literal `@` character when passing it into a `DateTime` constructor itself comes from the compound date/time formats supported by PHP.

The relative formats are the most powerful, supporting human-readable strings like these:

- `yesterday`
- `first day of`
- `now`
- `ago`

Armed with these formats, you can parse almost any string imaginable with PHP. However, there are some limits. In the Solution example, I used `now + 2 days` to specify "2 days from now." Example 6-5 demonstrates that the latter results in a parser error in PHP, even though it reads well in English.

Example 6-5. Limitations in `strtotime()` parsing

```
$date = strtotime('2 days from now');
```

```
if ($date === false) {
    throw new InvalidArgumentException('Error parsing t
}
```

It should always be noted that, no matter how clever you can make a computer, you are always limited by the quality of input provided by end users. There is no way you can foresee every possible way of specifying a date or time; `strtotime()` gets close, but you'll need to handle input errors as well.

Another potential way to parse user-provided dates is PHP's `date_parse()` function. Unlike `strtotime()`, this function expects a reasonably well-formatted input string. It also doesn't handle relative time quite the same way. Example 6-6 illustrates several strings that can be parsed by `date_parse()`.

Example 6-6. `date_parse()` **examples**

```
$first = date_parse('January 4, 2022');
```
❶

```
$second = date_parse('Feb 14');
```
❷

```
$third = date_parse('2022-11-12 5:00PM');
```
❸

```
$fourth = date_parse('1-1-2001 + 12 years');
```
❹

❶ Parses January 4, 2022

❷ Parses February 14, but with a `null` year

❸ Parses both the date and the time, but with no time zone

❹ Parses the date and stores an additional relative field

Rather than return a timestamp, `date_parse()` will extract the relevant date/time parts from the input string and store them in an associative array with keys for the following:

- `year`

- `month`
- `day`
- `hour`
- `minute`
- `second`
- `fraction`

In addition, passing a time-relative specification in the string (like the `+ 12 years` in Example 6-6) will add a `relative` key to the array with information about the relative offset.

All of this is useful in determining whether a user-provided date is useful as an actual date. The `date_parse()` function will also return warnings and errors if it encounters any parsing issues, making it even easier to check whether a date is valid. For more on checking date validity, read Recipe 6.7.

Revisiting Example 6-5 and leveraging `date_parse()` shows a little more about why PHP has trouble parsing `2 days from now` as a relative date. Consider the following example:

```
$date = date_parse('2 days from now');

if ($date['error_count'] > 0) {
    foreach ($date['errors'] as $error) {
        print $error . PHP_EOL;
    }
}
```

The preceding code will print `The time zone could not be found in the database`, which suggests PHP is *trying* to parse the date but is failing to identify what `from` really means in the statement `from now`. In fact, inspecting the `$date` array itself will show it returns a `relative` key. This relative offset properly represents the specified two days, meaning `date_parse()` (and even `strtotime()`) was able to read the relative date offset (`2 days`) but choked on the last part.

This additional error provides further context for debugging and could, perhaps, inform some kind of error message that the application should provide to the end user. In any case, it's more helpful than the mere `false` return of `strtotime()` on its own.

See Also

Documentation on `date_parse()` and `strtotime()`.

6.7 Validating a Date

Problem

You want to ensure that a date is valid. For example, you want to ensure that a user-defined birthdate is a real date on the calendar and not something like November 31, 2022.

Solution

Use PHP's `checkdate()` function as follows:

```
$entry = 'November 31, 2022';
$parsed = date_parse($entry);

if (!checkdate($parsed['month'], $parsed['day'], $parse
    throw new InvalidArgumentException('Specified date
}
```

Discussion

The `date_parse()` function was already covered in Recipe 6.6, but using it with `checkdate()` is new. This second function attempts to validate that the date is valid according to the calendar.

It checks that the month (first parameter) is between 1 and 12, that the year (third parameter) is between 1 and 32,767 (the maximum value of a 2-byte integer in PHP), and that the number of days is valid for that given month and year.

The `checkdate()` function properly handles months with 28, 30, or 31 days. Example 6-7 shows it also accounts for leap year, validating that February 29 exists in the appropriate years.

Example 6-7. Validating leap year

```
$valid = checkdate(2, 29, 2024); // true

$invalid = checkdate(2, 29, 2023); // false
```

See Also

PHP documentation on `checkdate()`.

6.8 Adding to or Subtracting from a Date

Problem

You want to apply a specific offset (either additive or subtractive) against a fixed date. For example, you want to calculate a future date by adding days to today's date.

Solution

Use the `::add()` or `::sub()` methods of a given `DateTime` object to add or subtract a `DateInterval`, respectively, as shown in Example 6-8.

Example 6-8. Simple `DateTime` addition

```
$date = new DateTime('December 25, 2023');

// When do the 12 days of Christmas end?
$twelve_days = new DateInterval('P12D');
$date->add($twelve_days);

print 'The holidays end on ' . $date->format('F j, Y');

// The holidays end on January 6, 2024
```

Discussion

Both the `::add()` and `::sub()` methods on a `DateTime` object modify the object itself by either adding or subtracting the given interval. Intervals are specified using a period designation that identifies the amount of time that interval represents. Table 6-5 illustrates the format characters used to denote an interval.

Table 6-5. Period designations used by `DateInterval`

Character	Description
Period designators	
`Y`	Years
`M`	Months
`D`	Days
`W`	Weeks
Time designators	
`H`	Hours
`M`	Minutes
`S`	Seconds

Every formatted date interval period starts with the letter `P`. This is followed by the number of years/months/days/weeks in that period. Any time elements in a duration are prefixed with the letter `T`.

WARNING

The period designations for months and minutes are both the letter `M`. This can lead to confusion when trying to identify 15 *minutes* versus 15 *months* in a time designation. If you intend to use minutes, ensure that your duration has properly used the `T` prefix to avoid a frustrating error in your application.

For example, a period of 3 weeks and 2 days would be represented as `P3W2D`. A period of 4 months, 2 hours, and 10 seconds would be represented as `P4MT2H10S`. Similarly, a period of 1 month, 2 hours, and 30 minutes would be represented as `P1MT2H30M`.

Mutability

Note that, in Example 6-8, the original `DateTime` object is itself modified when you call `::add()`. In a simple example, this is fine. If you're attempting to calculate *multiple* dates offset from the same starting date, the mutability of the `DateTime` object causes problems.

Instead, you can leverage the nearly identical `DateTimeImmutable` object. This class implements the same interface as `DateTime`, but the `::add()` and `::sub()` methods will instead return *new instances* of the class rather than mutating the internal state of the object itself.

Consider the comparison between both object types in Example 6-9.

Example 6-9. Comparing `DateTime` and `DateTimeImmutable`

```
$date = new DateTime('December 25, 2023');
$christmas = new DateTimeImmutable('December 25, 2023');

// When do the 12 days of Christmas end?
$twelve_days = new DateInterval('P12D');
$date->add($twelve_days);
                          ❶

$end = $christmas->add($twelve_days);
                          ❷


print 'The holidays end on ' . $date->format('F j, Y')
print 'The holidays end on ' . $end->format('F j, Y') .
```

```
// When is next Christmas?
$next_year = new DateInterval('P1Y');
$date->add($next_year);
$next_christmas = $christmas->add($next_year);

print 'Next Christmas is on ' . $date->format('F j, Y')
print 'Next Christmas is on ' . $next_christmas->format
                              ❸


print 'This Christmas is on ' . $christmas->format('F j
                              ❹
```

❶ Since `$date` is a mutable object, invoking its `::add()` method will modify the object directly.

❷ As `$christmas` is immutable, invoking `::add()` will return a new object that must be stored in a variable.

❸ Printing data from the resulting object from adding time to a `DateTimeImmutable` will present the correct data, as the *new* object was created with the right date and time.

❹ Even after invoking `:add()`, a `DateTimeImmutable` object will always contain the same data as it is, in fact, immutable.

The advantage of immutable objects is that you can treat them as constant and rest assured that no one is going to rewrite the calendar when you're not looking. The only disadvantage is with memory utilization. Since `DateTime` modifies a single object, memory doesn't necessarily increase as you keep making changes. Every time you "modify" a `DateTimeImmutable` object, however, PHP creates a new object and consumes additional memory.

In a typical web application, the memory overhead here will be negligible. There is no reason *not* to use a `DateTimeImmutable` object.

Simpler modification

In a similar track, both `DateTime` and `DateTimeImmutable` implement a `::modify()` method that works with human-readable strings rather than interval objects. This allows you to find relative dates like "last Friday" or "next week" from a given object.

A good example is Thanksgiving which, in the US, falls on the fourth Thursday in November. You can easily calculate the exact date in a given year

with the function defined in Example 6-10.

Example 6-10. Finding Thanksgiving with `DateTime`

```
function findThanksgiving(int $year): DateTime
{
    $november = new DateTime("November 1, {$year}");
    $november->modify('fourth Thursday');

    return $november;
}
```

The same functionality can be implemented using immutable date objects, as shown in Example 6-11.

Example 6-11. Finding Thanksgiving with `DateTimeImmutable`

```
function findThanksgiving(int $year): DateTimeImmutable
{
    $november = new DateTimeImmutable("November 1, {$ye
    return $november->modify('fourth Thursday');
}
```

See Also

Documentation on `DateInterval` .

6.9 Calculating Times Across Time Zones

Problem

You want to determine a specific time across more than one time zone.

Solution

Use the `::setTimezone()` method of the `DateTime` class to change a time zone as follows:

```
$now = new DateTime();
$now->setTimezone(new DateTimeZone('America/Los_Angeles

print $now->format(DATE_RSS) . PHP_EOL;

$now->setTimezone(new DateTimeZone('Europe/Paris'));

print $now->format(DATE_RSS) . PHP_EOL;
```

Discussion

Time zones are among the most frustrating things application developers need to worry about. Thankfully, PHP allows for converting from one time zone to another relatively easily. The `::setTimezone()` method used in the Solution example illustrates how an arbitrary `DateTime` can be converted from one time zone to another merely by specifying the desired time zone.

NOTE

Keep in mind that both `DateTime` and `DateTimeImmutable` implement a `::setTimezone()` method. The difference between their implementations is that `DateTime` will modify the state of the underlying object, while `DateTimeImmutable` will always return a *new* object instead.

It is important to know which time zones are available for use in code. The list is too long to enumerate, but developers can leverage `DateTimeZone::listIdentifiers()` to list all available named time zones. If your application only cares about a specific region, you can further pare down the list by using one of the predefined group constants that ship with the class.

For example, `DateTimeZone::listIdentifiers(DateTimeZone::AMERICA)` returns an array that lists all time zones available across the Americas. On a particular test system, this array has a list of 145 time zones, each pointing to a major local city to help identify the time zone they represent. You can generate a list of possible time zone identifiers for each of the following regional constants:

- `DateTimeZone::AFRICA`
- `DateTimeZone::AMERICA`
- `DateTimeZone::ANTARCTICA`
- `DateTimeZone::ARCTIC`
- `DateTimeZone::ASIA`
- `DateTimeZone::ATLANTIC`
- `DateTimeZone::AUSTRALIA`
- `DateTimeZone::EUROPE`
- `DateTimeZone::INDIAN`
- `DateTimeZone::PACIFIC`
- `DateTimeZone::UTC`
- `DateTimeZone::ALL`

Similarly, you can use bitwise operators to construct unions from these constants to retrieve lists of all time zones across two or more regions. For example, `DateTimeZone::ANTARCTICA | DateTimeZone::ARCTIC` would represent all time zones near either the South or North Pole.

The base `DateTime` class empowers you to instantiate an object with a specific time zone as opposed to accepting the system defaults. Merely pass a `DateTimeZone` instance as an optional second parameter to the constructor, and the new object will be set to the correct time zone automatically.

For example, the datetime `2022-12-15T17:35:53`, formatted according to ISO 8601, represents 5:35 p.m. on December 15, 2022, but does not reflect a specific time zone. When instantiating a `DateTime` object, you can easily specify this is a time in Tokyo, Japan, as follows:

```
$date = new DateTime('2022-12-15T17:35:53', new DateTim

echo $date->format(DateTime::ATOM);
// 2022-12-15T17:35:53+09:00
```

If time zone information is missing in the datetime string being parsed, providing that time zone makes things explicit. Had you *not* added a time zone identifier in the preceding example, PHP would have assumed the system's configured time zone instead.[2]

If time zone information *is* present in the datetime string, PHP will ignore any explicit time zone specified in the second parameter and parse the string as

provided.

See Also

Documentation on the `::setTimezone()` method and the `DateTimeZone` class.

[1] Format strings and available format characters are covered in Recipe 6.2.

[2] You can check the current time zone setting for your system with `date_default_timezone_get()`.