CHAPTER 12

Date and Time

Temporal matters play a role in practically every conceivable aspect of programming and are often crucial to representing data in a fashion of interest to users. When was a tutorial published? Is the pricing information for a particular product recent? What time did the office assistant log into the accounting system? At what hour of the day does the corporate Web site see the most visitor traffic? These and countless other time-oriented questions come about on a regular basis, making the proper accounting of such matters absolutely crucial to the success of your programming efforts.

This chapter introduces PHP's powerful date and time manipulation capabilities. After offering some preliminary information regarding how Unix deals with date and time values, you'll learn about several of the more commonly used functions found in PHP's date and time library. Next, we'll engage in a bout of Date Fu, where you'll learn how to use the date functions together to produce deadly (okay, useful) combinations, young grasshopper. We'll also create grid calendars using the aptly named PEAR package `Calendar`. Finally, the vastly improved date and time manipulation functions available as of PHP 5.1 are introduced.

The Unix Timestamp

Fitting the oft-incongruous aspects of our world into the rigorous constraints of a programming environment can be a tedious affair. Such problems are particularly prominent when dealing with dates and times. For example, suppose you were tasked with calculating the difference in days between two points in time, but the dates were provided in the formats July 4, 2005 3:45pm and 7th of December, 2005 18:17. As you might imagine, figuring out how to do this programmatically would be a daunting affair. What you would need is a standard format, some sort of agreement regarding how all dates and times will be presented. Preferably, the information would be provided in some sort of numerical format, 20050704154500 and 20051207181700, for example. Date and time values formatted in such a manner are commonly referred to as *timestamps*.

However, even this improved situation has its problems. For instance, this proposed solution still doesn't resolve challenges presented by time zones, matters pertinent to time adjustment due to daylight savings, or cultural date format variances. What we need is to standardize according to a single time zone, and to devise an agnostic format that could easily be converted to any desired format. What about representing temporal values in seconds, and basing everything on Coordinated Universal Time (UTC)? In fact, this strategy was embraced by the early Unix development team, using 00:00:00 UTC January 1, 1970 as the base from which all dates

are calculated. This date is commonly referred to as the *Unix epoch*. Therefore, the incongruously formatted dates in the previous example would actually be represented as 1120491900 and 1133979420, respectively.

■ **Caution** You may be wondering whether it's possible to work with dates prior to the Unix epoch (00:00:00 UTC January 1, 1970). Indeed it is, at least if you're using a Unix-based system. On Windows, due to an integer overflow issue, an error will occur if you attempt to use the timestamp-oriented functions in this chapter in conjunction with dates prior to the epoch definition.

PHP's Date and Time Library

Even the simplest of PHP applications often involve at least a few of PHP's date- and time-related functions. Whether validating a date, formatting a timestamp in some particular arrangement, or converting a human-readable date value to its corresponding timestamp, these functions can prove immensely useful in tackling otherwise quite complex tasks.

checkdate()

```
boolean checkdate (int month, int day, int year)
```

Although most readers could distinctly recall learning the "Thirty Days Hath September" poem[1] back in grade school, it's unlikely many of us could recite it, present company included. Thankfully, the `checkdate()` function accomplishes the task of validating dates quite nicely, returning `TRUE` if the date specified by `month`, `day`, and `year` is valid, and `FALSE` otherwise. Let's consider a few examples:

```
echo checkdate(4, 31, 2005);
// returns false

echo checkdate(03, 29, 2004);
// returns true, because 2004 was a leap yearf

echo checkdate(03, 29, 2005);
// returns false, because 2005 is not a leap year
```

date()

```
string date (string format [, int timestamp])
```

The `date()` function returns a string representation of the present time and/or date formatted according to the instructions specified by `format`. Table 12-1 includes an almost complete

1. "Thirty days hath September, April, June, and November; February has twenty-eight alone, All the rest have thirty-one, Excepting leap year, that's the time When February's days are twenty-nine."

breakdown of all available `date()` format parameters. Forgive the decision to forego inclusion of the parameter for Swatch Internet time[2].

Including the optional `timestamp` parameter, represented in Unix timestamp format, prompts `date()` to produce a string representation according to that designation. The `timestamp` parameter must be formatted in accordance with the rules of GNU's date syntax. If `timestamp` isn't provided, the current Unix timestamp will be used in its place.

Table 12-1. *The date() Function's Format Parameters*

Parameter	Description	Example
`a`	Lowercase ante meridiem and post meridiem	am or pm
`A`	Uppercase ante meridiem and post meridiem	AM or PM
`d`	Day of the month, with leading zero	01 to 31
`D`	Three-letter text representation of day	Mon through Sun
`F`	Complete text representation of month	January through December
`g`	12-hour format of hour, sans zeros	1 through 12
`G`	24-hour format, sans zeros	1 through 24
`h`	12-hour format of hour, with zeros	01 through 24
`H`	24-hour format, with zeros	01 through 24
`i`	Minutes, with zeros	01 through 60
`I`	Daylight saving time	0 if no, 1 if yes
`j`	Day of month, sans zeros	1 through 31
`l`	Text representation of day	Monday through Sunday
`L`	Leap year	0 if no, 1 if yes
`m`	Numeric representation of month, with zeros	01 through 12
`M`	Three-letter text representation of month	Jan through Dec
`n`	Numeric representation of month, sans zeros	1 through 12
`O`	Difference to Greenwich Mean Time (GMT)	–0500
`r`	Date formatted according to RFC 2822	Tue, 19 Apr 2005 22:37:00 –0500
`s`	Seconds, with zeros	01 through 59
`S`	Ordinal suffix of day	st, nd, rd, th

2. Created in the midst of the dotcom insanity, the watchmaker Swatch (`http://www.swatch.com/`) came up with the concept of *Swatch time,* which intended to do away with the stodgy old concept of time zones, instead setting time according to "Swatch beats." Not surprisingly, the universal reference for maintaining Swatch time was established via a meridian residing at the Swatch corporate office.

Table 12-1. *The date() Function's Format Parameters (Continued)*

Parameter	Description	Example
t	Number of days in month	28 through 31
T	Timezone setting of executing machine	PST, MST, CST, EST, etc.
U	Seconds since Unix epoch	1114646885
w	Numeric representation of weekday	0 for Sunday through 6 for Saturday
W	ISO-8601 week number of year	1 through 53
Y	Four-digit representation of year	1901 through 2038 (Unix); 1970 through 2038 (Windows)
z	The day of year	0 through 365
Z	Timezone offset in seconds	–43200 through 43200

Despite having regularly used PHP for years, many PHP programmers still need to visit the PHP documentation to refresh their memory about the list of parameters provided in Table 12-1. Therefore, although you likely won't be able to remember how to use this function simply by reviewing a few examples, let's look at a few examples just to give you a clearer understanding of what exactly `date()` is capable of accomplishing.

The first example demonstrates one of the most commonplace uses for `date()`, which is simply to output a standard date to the browser:

```
echo "Today is ".date("F d, Y");
// Today is April 27, 2005
```

The next example demonstrates how to output the weekday:

```
echo "Today is ".date("l");
// Today is Wednesday
```

Let's try a more verbose presentation of the present date:

```
$weekday = date("l");
$daynumber = date("dS");
$monthyear = date("F Y");

printf("Today is %s the %s day of %s", $weekday, $daynumber, $monthyear);
```

This returns the following output:

```
Today is Wednesday the 27th day of April 2005
```

You might be tempted to insert the nonparameter-related strings directly into the `date()` function, like this:

```
echo date("Today is l the ds day of F Y");
```

Indeed, this does work in some cases; however, the results can be quite unpredictable. For instance, executing the preceding code produces:

```
EDTo27pmO5 O351 Wednesday 3OO8e 2751 27pmO5 of April 2OO5
```

However, because punctuation doesn't conflict with any of the parameters, feel free to insert it as necessary. For example, to format a date as mm-dd-yyyy, use the following:

```
echo date("m-d-Y");
// O4-26-2OO5
```

Working with Time

The `date()` function can also produce time-related values. Let's run through a few examples, starting with simply outputting the present time:

```
echo "The time is ".date("h:i:s");
// The time is O7:44:53
```

But is it morning or evening? Just add the a parameter:

```
echo "The time is ".date("h:i:sa");
// The time is O7:44:53pm
```

getdate()

```
array getdate ([int timestamp])
```

The `getdate()` function returns an associative array consisting of timestamp components. This function returns these components based on the present date and time unless a Unix-format timestamp is provided. In total, 11 array elements are returned, including:

- `hours`: Numeric representation of the hours. The range is 0 through 23.
- `mday`: Numeric representation of the day of the month. The range is 1 through 31.
- `minutes`: Numeric representation of the minutes. The range is 0 through 59.
- `mon`: Numeric representation of the month. The range is 1 through 12.
- `month`: Complete text representation of the month, e.g. July.
- `seconds`: Numeric representation of seconds. The range is 0 through 59.
- `wday`: Numeric representation of the day of the week, e.g. 0 for Sunday.
- `weekday`: Complete text representation of the day of the week, e.g. Friday.
- `yday`: Numeric offset of the day of the year. The range is 0 through 365.

- year: Four-digit numeric representation of the year, e.g. 2005.
- 0: Number of seconds since the Unix epoch. While the range is system-dependent, on Unix-based systems, it's generally –2147483648 through 2147483647, and on Windows, the range is 0 through 2147483648.

■ **Caution** The Windows operating system doesn't support negative timestamp values, so the earliest date you could parse with this function on Windows is midnight, January 1, 1970.

Consider the timestamp 1114284300 (April 23, 2005 15:25:00 EDT). Let's pass it to getdate() and review the array elements:

```
Array (
 [seconds] => 0
 [minutes] => 25
 [hours] => 15
 [mday] => 23
 [wday] => 6
 [mon] => 4
 [year] => 2005
 [yday] => 112
 [weekday] => Saturday
 [month] => April
 [0] => 1114284300
)
```

gettimeofday()

```
mixed gettimeofday ([bool return_float])
```

The gettimeofday() function returns an associative array consisting of elements regarding the current time. For those running PHP 5.1.0 and newer, the optional parameter return_float causes gettimeofday() to return the current time as a float value. In total, four elements are returned, including:

- dsttime: Indicates the daylight savings time algorithm used, which varies according to geographic location. There are 11 possible values, including 0 (no daylight savings time enforced), 1 (United States), 2 (Australia), 3 (Western Europe), 4 (Middle Europe), 5 (Eastern Europe), 6 (Canada), 7 (Great Britain and Ireland), 8 (Romania), 9 (Turkey), and 10 (the Australian 1986 variation).
- minuteswest: The number of minutes west of Greenwich Mean Time (GMT).

- sec: The number of seconds since the Unix epoch.
- usec: The number of microseconds should the time fractionally supercede a whole second value.

Executing gettimeofday() from a test server on April 23, 2005 16:24:55 EDT produces the following output:

```
Array (
  [sec] => 1114287896
  [usec] => 110683
  [minuteswest] => 300
  [dsttime] => 1
)
```

Of course, it's possible to assign the output to an array and then reference each element as necessary:

```
$time = gettimeofday();
$GMToffset = $time['minuteswest'] / 60;
echo "Server location is $GMToffset hours west of GMT.";
```

This returns the following:

```
Server location is 5 hours west of GMT.
```

mktime()

```
int mktime ([int hour [, int minute [, int second [, int month
            [, int day [, int year [, int is_dst]]]]]]])
```

The mktime() function is useful for producing a timestamp, in seconds, between the Unix epoch and a given date and time. The purpose of each optional parameter should be obvious, save for perhaps is_dst, which should be set to 1 if daylight savings time is in effect, 0 if not, or –1 (default) if you're not sure. The default value prompts PHP to try to determine whether daylight savings is in effect. For example, if you want to know the timestamp for April 27, 2005 8:50 p.m., all you have to do is plug in the appropriate values:

```
echo mktime(20,50,00,4,27,2005);
```

This returns the following:

```
1114649400
```

This is particularly useful for calculating the difference between two points in time. For instance, how many hours are there between now and midnight April 15, 2006 (the next major U.S. tax day)?

```
$now = mktime();
$taxday = mktime(0,0,0,4,15,2006);

// Difference in seconds
$difference = $taxday - $now;

// Calculate total hours
$hours = round($difference / 60 / 60);

echo "Only $hours hours until tax day!";
```

This returns the following:

```
Only 8451 hours until tax day!
```

time()

```
int time()
```

The `time()` function is useful for retrieving the present Unix timestamp. The following example was executed at 15:25:00 EDT on April 23, 2005:

```
echo time();
```

This produces the following:

```
1114284300
```

Using the previously introduced `date()` function, this timestamp can later be converted back to a human-readable date:

```
echo date("F d, Y h:i:s", 1114284300);
```

This returns the following:

```
April 23, 2005 03:25:00
```

If you'd like to convert a specific date/time value to its corresponding timestamp, see the previous section for `mktime()`.

Date Fu

Some prize fighters never reach the upper echelons of their sport because they're one-dimensional. That is, they rely too heavily on one particular aspect of their fighting repertoire, a left hook, for instance. The truly world-class boxers take advantage of everything at their disposal, using combinations to attack, wear down, and ultimately defeat their competitors. This is analogous to effective use of the date functions: While sometimes only one function is all you need, often their true power becomes apparent when you use two or three together to produce the desired outcome. This section demonstrates several of the most commonly requested date-related "moves" (tasks), some of which involve just one function, and others that involve some combination of several functions.

Displaying the Localized Date and Time

Throughout this chapter, and indeed this book, the Americanized temporal and monetary formats have been commonly used, such as 04-12-05 and $2,600.93. However, other parts of the world use different date and time formats, currencies, and even character sets. Given the Internet's global reach, you may have to create an application that's capable of adhering to foreign, or *localized,* formats. In fact, neglecting to do so can cause considerable confusion. For instance, suppose you are going to create a Web site that books reservations for a popular hotel in Orlando, Florida. This particular hotel is popular among citizens of various other countries, so you decide to create several localized versions of the site. How should you deal with the fact that most countries use their own currency and date formats, not to mention different languages? While you could go to the trouble of creating a tedious method of managing such matters, it likely would be error-prone and take some time to deploy. Thankfully, PHP offers a built-in set of features for localizing this type of data.

PHP not only can facilitate proper formatting of dates, times, currencies, and such, but also can translate the month name accordingly. In this section, you'll learn how to take advantage of this feature to format dates according to any locality you please. Doing so essentially requires two functions, `setlocale()` and `strftime()`. Both are introduced, followed by a few examples.

setlocale()

```
string setlocale (mixed category, string locale [, string locale...])
string setlocale (mixed category, array locale)
```

The `setlocale()` function changes PHP's localization default by assigning the appropriate value to `locale`. Localization strings officially follow this structure:

```
language_COUNTRY.characterset
```

For example, if you wanted to use Italian localization, the locale string should be set to `it_IT`. Israeli localization would be set to `he_IL`, British localization to `en_GB`, and United States localization to `en_US`. The `characterset` component would come into play when potentially several character sets are available for a given locale. For example the locale string `zh_CN.gb18030` is used for handling Tibetan, Uigur, and Yi characters, whereas `zh_CN.gb3212` is for Simplified Chinese.

You'll see that the `locale` parameter can be passed as either several different strings or an array of locale values. But why pass more than one locale? This feature is in place (as of PHP

version 4.2.0) to counter the discrepancies between locale codes across different operating systems. Given that the vast majority of PHP-driven applications target a specific platform, this should rarely be an issue; however, the feature is there should you need it.

Finally, if you're running PHP on Windows, keep in mind that, apparently in the interests of keeping us on our toes, Microsoft has devised its own set of localization strings. You can retrieve a list of the language and country codes from `http://msdn.microsoft.com`.

■ **Tip** On some Unix-based systems, you can determine which locales are supported by running the command: `locale -a`.

It's possible to specify a locale for a particular classification of data. Six different categories are supported:

- `LC_ALL`: Set localization rules for all of the following five categories.
- `LC_COLLATE`: String comparison. This is useful for languages using characters such as â and é.
- `LC_CTYPE`: Character classification and conversion. For example, setting this category allows PHP to properly convert â to its corresponding lowercase representation of Â using the `strtolower()` function.
- `LC_MONETARY`: Monetary representation. For example, Americans represent 50 dollars as $50.00, whereas Italians represent 50 Euro as 50,00.
- `LC_NUMERIC`: Numeric representation. For example, Americans represent one thousand four hundred and twelve as 1,412.00, whereas Italians represent it as 1.412,00.
- `LC_TIME`: Date and time representation. For example, Americans represent dates with the month followed by the day, and finally the year. For example, February 12, 2005 might be represented as 02-12-2005. However, Europeans (and much of the rest of the world) represent this date as 12-02-2005. Once set, you can use the `strftime()` function to produce the localized format.

For example, suppose we were working with monetary values and wanted to ensure that the sums were formatted according to the Italian locale:

```
setlocale(LC_MONETARY, "it_IT");
echo money_format("%i", 478.54);
```

This returns:

```
EUR 478,54
```

To localize dates and times, you need to use `setlocale()` in conjunction with `strftime()`, introduced next.

strftime()

```
string strftime (string format [, int timestamp])
```

The `strftime()` function formats a date and time according to the localization setting as specified by `setlocale()`. While it works in the same format as `date()`, accepting conversion parameters that determine the layout of the requested date and time, unfortunately, the parameters are different from those used by `date()`, necessitating reproduction of all available parameters in Table 12-2 for your reference. Keep in mind that all parameters will produce the output according to the set locale. Also, note that some of these parameters aren't supported on Windows.

Table 12-2. *The strftime() Function's Format Parameters*

Parameter	Description	Examples or Range
%a	Abbreviated weekly name	Mon, Tue
%A	Complete weekday name	Monday, Tuesday
%b	Abbreviated month name	Jan, Feb
%B	Complete month name	January, February
%c	Standard date and time	04/26/05 21:40:46
%C	Century number	21
%d	Numerical day of month, with leading zero	01, 15, 26
%D	Equivalent to %m/%d/%y	04/26/05
%e	Numerical day of month, no leading zero	26
%g	Same output as %G, but without the century	05
%G	Numerical year, behaving according to rules set by %V	2005
%h	Same output as %b	Jan, Feb
%H	Numerical hour (24-hour clock), with leading zero	00 through 23
%I	Numerical hour (12-hour clock), with leading zero	00 through 12
%j	Numerical day of year	001 through 366
%m	Numerical month, with leading zero	01 through 12
%M	Numerical month, with leading zero	00 through 59
%n	Newline character	\n
%p	Ante meridiem and post meridiem	AM, PM
%r	Ante meridiem and post meridiem, with periods	A.M., P.M.
%R	24-hour time notation	00:01:00 through 23:59:59
%S	Numerical seconds, with leading zero	00 through 59

Table 12-2. *The strftime() Function's Format Parameters (Continued)*

Parameter	Description	Examples or Range
%t	Tab character	\t
%T	Equivalent to %H:%M:%S	22:14:54
%u	Numerical weekday, where 1 = Monday	1 through 7
%U	Numerical week number, where first Sunday is first day of first week	17
%V	Numerical week number, where week 1 = first week with >= 4 days	01 through 53
%W	Numerical week number, where first Monday is first day of first week	08
%w	Numerical weekday, where 0 = Sunday	0 through 6
%x	Standard date	04/26/05
%X	Standard time	22:07:54
%y	Numerical year, without century	05
%Y	Numerical year, with century	2005
%Z or %z	Time zone	Eastern Daylight Time
%%	The percentage character	%

By using `strftime()` in conjunction with `setlocale()`, it's possible to format dates according to your user's local language, standards, and customs. Recalling the travel site, it would be trivial to provide the user with a localized itinerary with travel dates and the ticket cost:

```
Benvenuto abordo, Sr. Sanzi<br />
<?php
   setlocale(LC_ALL, "it_IT");
   $tickets = 2;
   $departure_time = 1118837700;
   $return_time = 1119457800;
   $cost = 1350.99;
?>
Numero di biglietti: <?php echo $tickets; ?><br />
Orario di partenza: <?php echo strftime("%d %B, %Y", $departure_time); ?><br />
Orario di ritorno: <?php echo strftime("%d %B, %Y", $return_time); ?><br />
Prezzo IVA incluso: <?php echo money_format('%i', $cost); ?><br />
```

This example returns the following:

```
Benvenuto abordo, Sr. Sanzi
Numero di biglietti: 2
Orario di partenza: 15 giugno, 2005
Orario di ritorno: 22 giugno, 2005
Prezzo IVA incluso: EUR 1.350,99
```

Displaying the Web Page's Most Recent Modification Date

Barely a decade old, the Web is already starting to look like a packrat's office. Documents are strewn everywhere, many of which are old, outdated, and often downright irrelevant. One of the commonplace strategies for helping the visitor determine the document's validity involves adding a timestamp to the page. Of course, doing so manually will only invite errors, as the page administrator will eventually forget to update the timestamp. However, it's possible to automate this process using `date()` and `getlastmod()`. You already know `date()`, so this opportunity is taken to introduce `getlastmod()`.

getlastmod()

```
int getlastmod()
```

The `getlastmod()` function returns the value of the page's Last-Modified header, or `FALSE` in the case of an error. If you use it in conjunction with `date()`, providing information regarding the page's last modification time and date is trivial:

```
$lastmod = date("F d, Y h:i:sa", getlastmod());
echo "Page last modified on $lastmod";
```

This returns output similar to the following:

```
Page last modified on April 26, 2005 07:59:34pm
```

Determining the Number Days in the Current Month

To determine the number of days found in the present month, use the `date()` function's `t` parameter. Consider the following code:

```
printf("There are %d days in %s.", date("t"), date("F"));
```

If this was executed in April, the following result would be output:

```
There are 30 days in April.
```

Determining the Number of Days in Any Given Month

Sometimes you might want to determine the number of days in some month other than the present month. The `date()` function alone won't work because it requires a timestamp, and you might only have a month and year available. However, the `mktime()` function can be used in conjunction with `date()` to produce the desired result. Suppose you want to determine the number of days found in February of 2006:

```
$lastday = mktime(0, 0, 0, 3, 0, 2006);
printf("There are %d days in February, 2006.", date("t",$lastday));
```

Executing this snippet produces the following output:

```
There are 28 days in February, 2006.
```

Calculating the Date *X* Days from the Present Date

It's often useful to determine the precise date some specific number of days into the future or past. Using the `strtotime()` function and GNU date syntax, such requests are trivial. Suppose you want to know what the date will be 45 days into the future, based on today's date of April 23, 2005:

```
$futuredate = strtotime("45 days");
echo date("F d, Y", $futuredate);
```

This returns:

```
June 07, 2005
```

By prepending a negative sign, you can determine the date 45 days into the past:

```
$pastdate = strtotime("-45 days");
echo date("F d, Y", $pastdate);
```

This returns the following:

```
March 09, 2005
```

What about 10 weeks and 2 days from today?

```
$futuredate = strtotime("10 weeks 2 days");
echo date("F d, Y", $futuredate);
```

This returns:

```
July 04, 2005
```

Using strtotime() and the supported GNU date input formats, making such determinations is largely limited to your imagination.

Creating a Calendar

The Calendar package consists of 12 classes capable of automating numerous chronological tasks. The following list highlights just a few of the useful ways in which you can apply this powerful package:

- Render a calendar of any scope (hourly, daily, weekly, monthly, and yearly being the most common) in a format of your choice.
- Navigate calendars in a manner reminiscent of that used by the Gnome Calendar and Windows Date & Time Properties interface.
- Validate any date. For example, you can use Calendar to determine whether April 1, 2019 falls on a Monday (it does).
- Extend Calendar's capabilities to tackle a variety of other tasks, date analysis for instance.

In this section, you'll learn about Calendar's most important capabilities, followed by several examples showing you how to actually implement some of these interesting features. But before you can begin taking advantage of this powerful package, you need to install it. Although you learned all about the PEAR package installation process in Chapter 11, for those of you not yet entirely familiar with the installation process, the necessary steps are reproduced next.

Installing Calendar

To capitalize upon all of Calendar's features, you also need to install the Date package. Let's take care of both during the Calendar installation process, which follows:

```
%>pear install Date
downloading Date-1.4.3.tgz ...
Starting to download Date-1.4.3.tgz (42,048 bytes)
............done: 42,048 bytes
install ok: Date 1.4.3
%>pear install -f Calendar
Warning: Calendar is state 'beta' which is less stable than state 'stable'
downloading Calendar-0.5.2.tgz ...
Starting to download Calendar-0.5.2.tgz (60,164 bytes)
............done: 60,164 bytes
Optional dependencies:
package `Date' is recommended to utilize some features.
install ok: Calendar 0.5.2
%>
```

The –f flag is included when installing Calendar here because, at the time of this writing, Calendar is still a beta release. By the time of publication, Calendar could be officially stable, meaning you won't need to include this flag. See Chapter 11 for a complete introduction to PEAR and the install command.

Calendar Fundamentals

Calendar is a rather large package, consisting of 12 public classes broken down into four distinct groups:

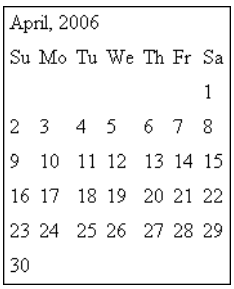
- **Date classes**: Used to manage the six date components: years, months, days, hours, minutes, and seconds. A separate class exists for each component: Calendar_Year, Calendar_Month, Calendar_Day, Calendar_Hour, Calendar_Minute, and Calendar_Second, respectively.
- **Tabular date classes**: Used to build monthly and weekly grid-based calendars. Three classes are available: Calendar_Month_Weekdays, Calendar_Month_Weeks, and Calendar_Week. These classes are useful for building monthly tabular calendars in daily and weekly formats, and weekly tabular calendars in seven-day format, respectively.
- **Validation classes**: Used to validate dates. The two classes are Calendar_Validator, which is used to validate any component of a date and can be called by any subclass, and Calendar_Validation_Error, which offers an additional level of reporting if something is wrong with a date, and provides several methods for dissecting the date value.
- **Decorator classes**: Used to extend the capabilities of the other subclasses without having to actually extend them. For instance, suppose you want to extend Calendar's functionality with a few features for analyzing the number of Saturdays falling on the 17th of any given month. A decorator would be an ideal way to make that feature available. Several decorators are offered for reference and use, including Calendar_Decorator, Calendar_Decorator_Uri, Calendar_Decorator_Textual, and Calendar_Decorator_Wrapper. In the interests of sticking to a discussion of the most commonly used features, Calendar's decorator internals aren't discussed here; consider examining the decorators installed with Calendar for ideas regarding how you can go about creating your own.

All four classes are subclasses of Calendar, meaning all of the Calendar class's methods are available to each subclass. For a complete summary of the methods for this superclass and the four subclasses, see http://pear.php.net/package/Calendar.

Creating a Monthly Calendar

These days, grid-based monthly calendars seem to be one of the most commonly desired Web site features, particularly given the popularity of time-based content such as blogs. Yet creating one from scratch can be deceivingly difficult. Thankfully, Calendar handles all of the tedium for you, offering the ability to create a grid calendar with just a few lines of code. For example, suppose we want to create a calendar for the present month and year, as shown in Figure 12-1.

The code for creating this calendar is surprisingly simple, and is presented in Listing 12-1. An explanation of key lines follows the code, referring to their line numbers for convenience.



April, 2006						
Su	Mo	Tu	We	Th	Fr	Sa
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

Figure 12-1. *A grid calendar for April, 2006*

Listing 12-1. *Creating a Monthly Calendar*

```
<?php
  require_once 'Calendar/Month/Weekdays.php';

  $month = new Calendar_Month_Weekdays(2006, 4, 0);

  $month->build();

  echo "<table cellspacing='5'>\n";
  echo "<tr><td class='monthname' colspan='7'>April, 2006</td></tr>";
  echo "<tr><td>Su</td><td>Mo</td><td>Tu</td><td>We</td>
           <td>Th</td><td>Fr</td><td>Sa</td></tr>";
  while ($day = $month->fetch()) {
     if ($day->isFirst()) {
        echo "<tr>";
     }

     if ($day->isEmpty()) {
        echo "<td> </td>";
     } else {
        echo '<td>'.$day->thisDay()."</td>";
     }

     if ($day->isLast()) {
        echo "</tr>";
     }
  }

  echo "</table>";
?>
```

- **Line 02:** Because we want to build a grid calendar representing a month, the `Calendar_Month_Weekdays` class is required. Line 02 makes this class available to the script.
- **Line 04:** The `Calendar_Month_Weekdays` class is instantiated, and the date is set to April, 2006. The calendar should be laid out from Sunday to Saturday, so the third parameter is set to `0`, which is representative of the Sunday numerical offset (`1` for Monday, `2` for Tuesday, and so forth).
- **Line 06:** The `build()` method generates an array consisting of all dates found in the month.
- **Line 12:** A `while` loop begins, responsible for cycling through each day of the month.
- **Lines 13–15:** If `$Day` is the first day of the week, output a `<tr>` tag.
- **Lines 17–21:** If `$Day` is empty, output an empty cell. Otherwise, output the day number.
- **Lines 23–25:** If `$Day` is the last day of the week, output a `</tr>` tag.

Pretty simple isn't it? Creating weekly and daily calendars operates on a very similar premise. Just choose the appropriate class and adjust the format as you see fit.

Validating Dates and Times

While PHP's `checkdate()` function is useful for validating a date, it requires that all three date components (month, day, and year) are provided. But what if you want to validate just one date component, the month, for instance? Or perhaps you'd like to make sure a time value (`hours:minutes:seconds`), or some particular part of it, is legitimate before inserting it into a database. The `Calendar` package offers several methods for confirming both dates and times, or any part thereof. This list introduces these methods:

- `isValid()`: Executes all the other time and date validator methods, validating a date and time
- `isValidDay()`: Ensures that a day falls between 1 and 31
- `isValidHour()`: Ensures that the value falls between 0 and 23
- `isValidMinute()`: Ensures that the value falls between 0 and 59
- `isValidMonth()`: Ensures that the value falls between 1 and 12
- `isValidSecond()`: Ensures that the value falls between 0 and 59
- `isValidYear()`: Ensures that the value falls between 1902 and 2037 on Unix, or 1970 and 2037 on Windows

PHP 5.1

While the built-in date functions discussed earlier in this chapter are very useful, users interested in manipulating and navigating dates are left out in the cold. For example, there is no readily available function for determining what day comes after Monday, what month comes

after November, or whether a given year is a leap year. While the `Calendar` package introduced in the last section offers these capabilities, it would be nice to make these enhancements available via the default distribution. Those of you who have long yearned for such features are in luck, because the PECL[3] Date and Time extension has been incorporated into the standard PHP distribution as of version 5.1. Authored by Pierre-Alain Joye, the Date and Time Library (hereafter referred to as `Date`) is guaranteed to make the lives of many PHP programmers significantly easier. In this section, you'll learn about `Date` and see its powerful capabilities demonstrated through several examples.

■ **Caution** This chapter was written several months ahead of the official PHP 5.1 release, at a time when no documentation was available for the `Date` extension. Therefore, be forewarned that any information found in this section could indeed be incorrect by the time you read this. Nor does this section offer a comprehensive summary of all available features, as at the time of writing several of the methods weren't working properly, and therefore it was decided better to omit them from the material. Such are the risks one takes to stay on the leading edge of technology!

Date Fundamentals

Earlier in the chapter, it was half-jokingly mentioned that offering `date()` examples was just for the sake of demonstration, because you'll nonetheless need to refer to the documentation (or this book) for years in order to recall what the somewhat nonsensical parameters do. `Date` takes away much of the guesswork because it's fully object-oriented, meaning the process involved in juggling dates is somewhat natural because the method names are rather self-explanatory. For example, to set the month, you call the `setMonth()` mutator; to retrieve the year, you call the `getYear()` accessor; and so on. The remainder of this chapter is devoted to an introduction of this class and its many methods.

■ **Note** Because `Date` relies on object-oriented features available as of version 5.0, you cannot use `Date` in conjunction with any earlier version. If you haven't yet upgraded to version 5.1 (but are using version 5.0.X) and want to use `Date`, download it from `http://pecl.php.net/package/date_time`.

The Date Constructor

Before you can use the `Date` features, you need to instantiate a date object via its class constructor. This constructor is introduced in this section.

---

3. PECL is the PHP Extension Community Library, containing PHP extensions written in the C language. Learn more about it at `http://pecl.php.net`.

date()

```
object date ([integer day [, integer month [, integer year [, integer weekstart]]]])
```

The `date()` method is the class constructor. You can set the date either at the time of instantiation by using the `day`, `month`, and `year` parameters, or later by using a variety of mutators (setters), which are introduced next. To create an empty date object, just call `date()` like so:

```
$date = new Date();
```

To create an object and set the date to April 29, 2005, execute:

```
$date = new Date(29,4,2005);
```

You can use the optional `weekstart` parameter to tell the object which day of the week should be considered the first. By default, date objects assume the week begins with Monday, meaning Monday has the offset 1.

Curiously, there is no convenient means for setting the date object to the current date. To do so, you need to use the `date()` function:

```
$date = new Date(date("j"),date("n"),date("Y"));
```

Accessors and Mutators

`Date` offers several accessors (getters) and mutators (setters) that are useful for manipulating and retrieving date component values. Those methods are introduced in this section.

setDMY()

```
boolean setDMY (integer day, integer month, integer year)
```

The `setDMY()` method sets the date object's day, month, and year, returning `TRUE` on success and `FALSE` otherwise. Let's set the date to April 29, 2005:

```
$date = new Date();
$date->setDMY(29,4,2005);
$dcs = $date->getArray();
print_r($dcs);
```

This returns the following:

```
Array (
   [day] => 29 [month] => 4 [year] => 2005
   [hour] => 0 [min] => 0 [sec] => 0
)
```

The `getArray()` method is convenient for easily storing all three date components in an array. This method is introduced next.

getArray()

```
array getArray()
```

The `getArray()` method returns an associative array consisting of three keys: `day`, `month`, and `year`:

```
$date = new Date();
$date->setDMY(29,4,2005);
$dcs = $date->getArray();
echo "The month: ".$dcs['month']."<br />";
echo "The day: ".$dcs['day']."<br />";
echo "The year: ".$dcs['year']."<br />";
```

The result follows:

```
The month: 4
The day: 29
The year: 2005
```

setDay()

```
boolean setDay (integer day)
```

The `setDay()` method sets the date object's `day` attribute to `day`, returning `TRUE` on success and `FALSE` otherwise. The following example sets the date to April 29, 2006 and then changes the day to 15:

```
$date = new Date(29,4,2006);
$date->setDay15);
// The date is now set to April 15, 2006
```

getDay()

```
integer getDay()
```

The `getDay()` method returns the `day` attribute from the date object. An example follows:

```
$date = new Date(29,4,2006);
echo $date->getDay();
```

The following is returned:

```
29
```

setJulian()

The Julian date was created by historian Joseph Scaliger (1540–1609) in an attempt to convert between the many disparate calendaring systems he encountered when studying historical documents. It's based on a 7,980-year cycle, because this number is a multiple of several common time cycles (namely the lunar and solar cycles and a Roman taxation cycle) that served as the foundation for these systems. Julian dates are represented by the number of days elapsed from a specific date, and the first Julian cycle began at noon on January 1, 4,713 B.C. on the Julian calendar; therefore, the Julian date equivalent for April 29, 2006 is 2453851.5.

■ **Caution** Julian dates bear no relation to the 365-day Julian calendaring system we use today, which was instituted by Julius Caesar in 46 B.C.

---

getJuliaan()

```
int getJuliaan()
```

The `getJuliaan()` method returns the Julian date calculated from the date specified by the date object. Interestingly, as of the time of writing, Julian is misspelled as Juliaan. If you use this method, be sure to monitor future releases, because this is likely to change to the correct spelling in the future.

setMonth()

```
boolean setMonth (integer month)
```

The `setMonth()` method sets the date object's `month` attribute to `month`, returning `TRUE` on success and `FALSE` otherwise. The following example sets the date to April 29, 2005 and then changes the month to July:

```
$date = new Date(29,4,2005);
$date->setMonth(7);
// The month is now set to July (7)
```

getMonth()

```
integer getMonth()
```

The `getMonth()` method returns the month attribute from the date object. An example follows:

```
$date = new Date(29,4,2005);
echo $date->getMonth();
```

This returns:

4

---

setYear()

```
boolean setYear (integer year)
```

The setYear() method sets the date object's year attribute to year, returning TRUE on success and FALSE otherwise. The following example sets the date to April 29, 2005 and then changes the year to 2006:

```
$date = new Date(29,4,2005);
$date->setYear(2006);
// The year is now set to 2006
```

getYear()

```
integer getYear()
```

The getYear() method returns the year attribute from the date object. An example follows:

```
$date = new Date(29,4,2005);
echo $date->getYear();
```

The result returned follows:

```
2005
```

Validators

Date offers a method for determining whether the date falls on a leap year and a method for validating the date's correctness. Both of those methods are introduced in this section.

isLeap()

```
boolean isLeap()
```

The isLeap() method returns TRUE if the year represented by the date object is a leap year, and FALSE otherwise. The following script uses isLeap() in conjunction with a ternary operator to inform the user whether a given year is a leap year:

```
$year = 2005;
$date = new Date(date("j"),date("n"),$year);
echo "$year is ". ($date->isLeap() == 1 ? "" : "not"). " a leap year.";
```

This produces the following output:

```
2005 is not a leap year.
```

isValid()

```
boolean isValid()
```

The `isValid()` method returns `TRUE` if the date represented by the date object is valid, and `FALSE` otherwise. Because this method can't be called statically, and it's impossible to set an invalid date using the constructor of any of the mutators, it isn't presently apparent why `isValid()` exists.

Manipulation Methods

Of course, the true applicability of this class comes from its date-manipulation capabilities. In this section, you'll learn about the functions that allow you to manipulate dates with ease

addDays()

```
boolean addDays (int days)
```

The `addDays()` method adds `days` days to the date object, adjusting the month and year accordingly should the new `day` value surpass the present month's total number of days, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2005 and we use `addDays()` to add five days:

```
$date = new Date();
$date->setDMY(28,4,2005);
$date->addDays(5);
$dcs = $date->getArray();
print_r($dcs);
```

The following is returned:

```
Array (
   [day] => 3 [month] => 5 [year] => 2005
   [hour] => 0 [min] => 0 [sec] => 0
)
```

subDays()

```
boolean subDays (int days)
```

The `subDays()` method subtracts `days` days from the date object, adjusting the month and year accordingly should `days` be greater than the date's `day` component, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `addDays()` to subtract 14 days:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->subDays(14);
$dcs = $date->getArray();
print_r($dcs);
```

This returns:

```
Array (
   [day] => 14 [month] => 4 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

addMonths()

```
boolean addMonths (int months)
```

The `addMonths()` method adds `months` months to the date object's `month` attribute, adjusting the year accordingly should the new month value be greater than 12, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `addMonths()` to add nine months:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->addMonths(9);
$dcs = $date->getArray();
print_r($dcs);
```

The following is the output:

```
Array (
   [day] => 28 [month] => 1 [year] => 2007
   [hour] => 0 [min] => 0 [sec] => 0
)
```

In the case that the new month does not possess the number of days found in the `day` attribute, then `day` will be adjusted downward to the last day of the new month.

subMonths()

```
boolean subMonths (int months)
```

The `subMonths()` method subtracts `months` months from the date object's `month` attribute, adjusting the year accordingly should the new `month` value be less than zero, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `subMonths()` to add nine months:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->subMonths(9);
$dcs = $date->getArray();
print_r($dcs);
```

This returns:

```
Array (
   [day] => 28 [month] => 7 [year] => 2005
   [hour] => 0 [min] => 0 [sec] => 0
)
```

In the case that the new month does not possess the number of days found in the day attribute, then day will be adjusted downward to the last day of the new month.

addWeeks()

```
boolean addWeeks (int weeks)
```

The addWeeks() method adds weeks weeks to the date object's date, returning TRUE on success and FALSE otherwise. For example, suppose the object's date is set to April 28, 2006 and we use addWeeks() to add seven weeks:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->addWeeks(7);
$dcs = $date->getArray();
print_r($dcs);
```

The following is returned:

```
Array (
   [day] => 16 [month] => 6 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

subWeeks()

```
boolean subWeeks (int weeks)
```

The subWeeks() method subtracts weeks weeks from the date object's date, returning TRUE on success and FALSE otherwise. For example, suppose the object's date is set to April 28, 2006 and we use subWeeks() to subtract seven weeks:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->subWeeks(7);
$dcs = $date->getArray();
print_r($dcs);
```

This returns the following:

```
Array (
   [day] => 10 [month] => 3 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

addYears()

```
boolean addYears (int years)
```

The `addYears()` method adds `years` years from the date object's `year` attribute, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `addYears()` to add four years:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->addYears(4);
$dcs = $date->getArray();
print_r($dcs);
```

This returns the following:

```
Array (
   [day] => 28 [month] => 4 [year] => 2010
   [hour] => 0 [min] => 0 [sec] => 0
)
```

subYears()

```
boolean subYears (int years)
```

The `subYears()` method subtracts `years` years from the date object's `year` attribute, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `subYears()` to subtract two years:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->subYears(2);
$dcs = $date->getArray();
print_r($dcs);
```

The following output is returned:

```
Array (
   [day] => 28 [month] => 4 [year] => 2004
   [hour] => 0 [min] => 0 [sec] => 0
)
```

getWeekday()

```
integer getWeekday()
```

The `getWeekday()` method returns the numerical offset of the day specified by the date object. An example follows:

```
$date = new Date();
$date->setDMY(30,4,2006);
echo $date->getWeekday();
```

This returns the following, which is a Sunday, because Sunday's numerical offset is 7:

```
7
```

setToWeekday()

```
boolean setToWeekday (int weekday, int n [, int month [, int year]])
```

The `setToWeekday()` method sets the date to the `n`th `weekday` of the `month` and `year`, returning `TRUE` on success and `FALSE` otherwise. If no `month` and `year` are provided, the present month and year are used. As of the time of writing, this method was broken; quite likely it will have been fixed by the time this book is published.

getDayOfYear()

```
integer getDayOfYear()
```

The `getDayOfYear()` method returns the numerical offset of the day specified by the date object. An example follows:

```
$date = new Date();
$date->setDMY(4,7,1776);
echo $date->getDayOfYear();
```

The following is the result:

```
186
```

getWeekOfYear()

```
integer getWeekOfYear()
```

The `getDayOfYear()` method returns the numerical offset of the week specified by the date object:

```
$date = new Date();
$date->setDMY(4,7,1776);
echo $date->getWeekOfYear();
```

This returns:

```
27
```

getISOWeekOfYear()

```
integer getISOWeekOfYear()
```

The `getISOWeekOfYear()` method returns the week number of the date represented by the date object according to the ISO 8601 specification. ISO 8601 states that the first week of the year is the week containing the first Thursday. For instance, the first day of 2005 fell on a Sunday, but January 2 through 8 contained the first Thursday; therefore, January 1 does not even count as falling in the first week of the year. You might think this a tad odd; however, the decision is almost arbitrary in that it just standardizes the method for determining what constitutes the year's first week. Let's see this explanation in action by querying for the week number in which January 4 falls:

```
$date = new Date();
$date->setDMY(4,1,2005);
echo $date->getISOWeekOfYear();
```

The following is returned:

```
1
```

So, given that January 1 doesn't qualify as falling within the first week of the year, within what week does it fall? You might be surprised to learn the ISO standard actually considers it to be the 53rd week of 2004:

```
$date = new Date();
$date->setDMY(1,1,2005);
echo $date->getISOWeekOfYear();
```

This returns:

```
53
```

setToLastMonthDay()

```
boolean setToLastMonthDay()
```

The `setToLastMonthDay()` method adjusts the date object's `day` attribute to the last day of the month specified by the `month` attribute, returning `TRUE` on success and `FALSE` otherwise. An example follows:

```
$date = new Date();
$date->setDMY(1,4,2006);
$date->setToLastMonthDay();
echo $date->getDay();
```

The following output is returned:

```
30
```

setFirstDow()

```
boolean setFirstDow()
```

The `setFirstDow()` method sets the date object's `day` attribute to the first day of the week as specified by the `weekstart` attribute, returning `TRUE` on success and `FALSE` otherwise. By default, `weekstart` is set to Monday. The following example sets the date April 28, 2006 (which is a Friday), and then moves the date to the first day of the week (a Monday):

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->setFirstDow();
$dcs = $date->getArray();
print_r($dcs);
```

This returns:

```
Array (
   [day] => 24 [month] => 4 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

setLastDow()

```
boolean setLastDow()
```

The `setLastDow()` method sets the date object's `day` attribute to the last day of the week, returning `TRUE` on success and `FALSE` otherwise. This day is dependent upon the value of the `weekstart` attribute, which is set to Monday by default. The following example sets the date April 28, 2006 (which is a Friday), and then moves the date to the last day of the week (a Sunday):

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->setLastDow();
$dcs = $date->getArray();
print_r($dcs);
```

This returns:

```
Array (
   [day] => 30 [month] => 4 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

Summary

This chapter covered quite a bit of material, beginning with an overview of several date and time functions that appear almost daily in typical PHP programming tasks. Next up was a journey into the ancient art of Date Fu, where you learned how to combine the capabilities of these functions to carry out useful chronological tasks. We also covered the useful `Calendar` PEAR package, where you learned how to create grid-based calendars, and both validation and navigation mechanisms. Finally, for those readers living on the frayed edges of emerging technology, an introduction to PHP 5.1's new date-manipulation features was provided.

The next chapter is focused on the topic that is likely responsible for piquing your interest in learning more about PHP: user interactivity. We'll jump into data processing via forms, demonstrating both basic features and advanced topics such as how to work with multivalued form components and automated form generation. You'll also learn how to facilitate user navigation by creating breadcrumb navigation trails and custom 404 messages.