

Date and Time Classes

Did you know that the last minute of 1995 had 61 seconds? This fact may have eluded the creators of *Trivial Pursuit*, but it did not pass unnoticed to the creators of Java. If you don't really care about things like this, you may find Java's date and time support frustratingly complicated. However, if you desire software that supports international interpretations of date and time, you've come to the right programming language. Whether your interest lies in time zones, daylight savings, a lunar calendar for your favourite corner of planet Earth, or micro-adjustments for discrepancies in the earth's rotation, there is support in Java for creating accurate calendars and clocks to meet your needs.

Dates and times in Java are based on coordinated universal time (UTC). This is similar to Greenwich mean time (GMT), with an additional correction to compensate for the earth's non-uniform rotation. In the UTC time system, leap seconds are occasionally added at the end of a year, hence the 61 seconds in the last minute of 1995.

We will not attempt a thorough discussion of Java's date and time classes. For more details consult the Java API documentation. Table 1 summarizes the most important classes for creating and working with date and time objects.

Table 1. Date and Time Classes

Class (package)	Summary
Calendar (java.util)	an abstract base class for converting between a Date object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on. Subclasses of Calendar interpret a Date according to the rules of a specific calendar system. At present, there is one concrete subclass of Calendar: GregorianCalendar
Date (java.util)	a class that represents a specific instant in time, with millisecond precision
DateFormat (java.text)	an abstract class for date/time formatting subclasses which formats and parses dates or time in a language-independent manner. At present, there is one concrete subclass of DateFormat: SimpleDateFormat
GregorianCalendar (java.util)	a concrete subclass of Calendar that provides a Gregorian calendar
SimpleDateFormat (java.text)	a concrete subclass of DateFormat for formatting and parsing dates in a locale-sensitive manner. It allows for formatting (date-to-text), parsing (text-to-date), and normalizing
SimpleTimeZone (java.util)	a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar. This class does not handle historical changes.
TimeZone (java.util)	a class that represents a time zone offset, and also figures out daylight savings time
Locale (java.util)	a class that represents a specific geographical, political, or cultural region.

Let's examine a few simple demo programs that exercise Java's date and time classes.

Date and Time Formats

The `DemoDateTime` program uses the `Date` class and the `SimpleDateFormat` class to obtain the current date and time and print it in some common formats (see Figure 1).

```
1 import java.util.*;
2 import java.text.*;
3
4 public class DemoDateTime
5 {
6     public static void main(String[] args)
7     {
8         // get current date and time
9         Date currentTime = new Date();
10
11         // create SimpleDateFormat object with default format
12         SimpleDateFormat sdf = new SimpleDateFormat();
13
14         // print current date and time using default format
15         System.out.println(sdf.format(currentTime));
16
17         // set a new format
18         sdf.applyPattern("'Date:' MMMM d, yyyy 'Time:' hh:mm:ss a zzz");
19
20         // print again using new format
21         System.out.println(sdf.format(currentTime));
22
23         // set a format for day of week
24         sdf.applyPattern("'Today is' EEEE");
25
26         // print day of the week using new format pattern
27         System.out.println(sdf.format(currentTime));
28     }
29 }
```

Figure 1. `DemoDateTime.java`

This program generated the following output:

```
1/2/99 5:15 AM
Date: January 2, 1999 Time: 05:15:01 AM EST
Today is Saturday
```

This was a correct representation of the date and time on the local computer when the program was run. The program begins by instantiating a `Date` object (line 9) named `currentTime` and a `SimpleDateFormat` object (line 12) named `sdf`. The default constructor (no arguments) is used in both cases. The current date and time is printed in line 15 using the default format, and this appears in the first line of output above. Within the print statement, the following expression appears:

```
sdf.format(currentTime)
```

This method returns a `String` representation of the `currentTime` object, formatted according to the `sdf` object.

Let's digress briefly to show how *inheritance* is at work in the expression above. If you look in the Java API documentation for the `format()` method in the `SimpleDateFormat` class, you'll discover that it does not appear in the form above — with one `Date` object as an argument. The `format()` method, as used above, appears in the `DateFormat` class, which is the

superclass to `SimpleDateFormat` class. Since `SimpleDateFormat` is a subclass to `DateFormat`, it inherits all the methods of `DateFormat`, such as `format(String s)`. We will talk about inheritance in more detail later.

If we want a different format for the date and/or time, then the `applyPattern()` method of the `SimpleDateFormat` class is used. This is illustrated in line 18 of Figure 1. The effect of the new format is shown in line 21 by printing the date and time again. This is shown in the second line of output. The characters in the format pattern in line 18 are part of a flexible set of formatting symbols. The complete set is shown in Table 2.

Table 2. Date and Time Formatting Symbols

Symbol	Meaning	Example
G	era designator	AD
Y	year	1996
M	month in year	July & 07
d	day in month	10
h	hour in am/pm (1~12)	12
H	hour in day (0~23)	0
m	minute in hour	30
s	second in minute	55
S	millisecond	978
E	day in week	Tuesday
D	day in year	189
F	day of week in month	2 (2nd Wed in July)
w	week in year	27
W	week in month	2
a	am/pm marker	PM
k	hour in day (1~24)	24
K	hour in am/pm (0~11)	0
z	time zone	Pacific Standard Time
'	escape for text	
''	single quote	'

Clearly, the symbols in Table 2 offer many possibilities for working with dates and time. The count of the formatting symbols determines the format. For text fields, if four or more symbols appear, the full form is used (e.g., "EEEE" might yield `Tuesday`), otherwise an abbreviated form is used (e.g., "EEE" might yield `Tue`). For numeric fields, the number of characters determines the minimum number of digits, with shorter numbers padded left with zeros. The year symbol is handled differently: If the count is 2, year is truncated to 2 digits. The month-in-year field also has a special interpretation. If three or more characters appear, the text form is used (e.g., "MMM" might yield `January`), otherwise the numeric form is used (e.g., "MM" might yield `01`). Punctuation characters in the formatting string are presented as such. Additional alpha characters are included by enclosing them in single quotes.

The formatting pattern is changed in line 24 to specify only the day of the week, as printed in line 27 and shown in the third line of output.

Real-Time Clock

Time and tide wait for no man (Chaucer, c. 1390) — and neither does the host system's clock. The `DemoDateTime` program illustrated flexible ways to control the printed format of dates and times. However, the three lines of output all represented the single instance in time when the `Date()` constructor was called (see line 9 in Figure 1). In this section, we'll examine a real-time clock that outputs a new date and time each second.

To update the time "each second", we need access to the "seconds" field in the current date and time. Although we could examine the strings created by the `format()` method of the `DateFormat` class, this is awkward. A better approach is to use a calendar object. Objects of Java's `Calendar` class (and its subclass, `GregorianCalendar`) can be updated using the `setTime()` method so that they reflect the current time. Furthermore, individual fields (seconds, day of the week, etc.) can be accessed using the `set()` and `get()` methods. Let's illustrate this in a demo program. The program `DemoClock` creates a real-time clock on the host system's display (see Figure 2).

```
1 import java.util.*;
2
3 public class DemoClock
4 {
5     public static void main(String[] args)
6     {
7         Calendar today = new GregorianCalendar();
8         int seconds = today.get(Calendar.SECOND);
9         while (true)
10        {
11            today.setTime(new Date());
12            int newSeconds = today.get(Calendar.SECOND);
13            if (newSeconds != seconds)
14            {
15                System.out.print(today.getTime() + "\r");
16                seconds = newSeconds;
17            }
18        }
19    }
20 }
```

Figure 2. `DemoClock.java`

This program generated the following output when it was executed on the local system:

```
Sat Jan 02 10:34:28 EST 1999
```

The output was updated every second. By using the `print()` method, instead of `println()`, and by appending `"\r"` to the string representing the current time, each update overwrites the previous update (see line 15).¹ This creates a nice visual effect for the output.

¹ Recall that `"\r"` represents "return", whereas `"\n"` represents "newline". The effect of printing `"\r"` is to return the pointer to the left side of the current line without advancing to the next line. Characters subsequently written overwrite previous characters.

A new `Calendar` object is instantiated in line 7 and a reference to it is assigned to the object variable `today`. Note that the constructor for a `GregorianCalendar` object is used. Since `GregorianCalendar` is a subclass of `Calendar`, this is a legitimate operation. The instantiated object is a `Calendar` object bearing the characteristics of the current date and time as per the Gregorian calendar — the calendar used in most of the world.

Initially, `today` holds a representation of the current date and time. In line 8, the seconds value is extracted from `today`, and assigned to the `int` variable `seconds`. This occurs using the `get()` method of the `Calendar` class with the field `Calendar.SECONDS` as an argument.

Lines 9-18 contain an infinite loop that outputs the current date and time once per second. (The program is terminated by entering Control-c on the keyboard.) Within the loop, the first task is to update `today` to the current date and time. This happens in line 11 using the `setTime()` method. The argument to `setTime()` is a `Date` object. The expression "`new Date()`" is the argument, and this achieves the desired effect of getting the "current" date and time. Then, the seconds value is extracted from `now` and assigned to the `int` variable `newSeconds` in line 12. The first time line 12 executes, `newSeconds` is likely assigned the same value as `seconds` in line 8. (After all, today's computers are pretty fast!) `newSeconds` and `seconds` are compared in the `if` statement in line 13. If they differ, lines 15-16 execute: The current date and time are printed (line 15) and `newSeconds` is updated to `seconds` (line 16). If they are the same, another iteration of the `while` loop begins and `newSeconds` is updated again.

If the description above sounds familiar, it should. The exact same technique was employed in the `CountSeconds` program presented earlier.

Calendar Entries

`Calendar` objects are also used to represent important dates in the future, such as anniversaries, holidays, or appointments. A variety of manipulations may be required for such objects. We may want to know how many shopping days are left until Christmas or some other gift-sharing holiday, for example. Or, perhaps we want a reminder when someone's birthday arrives. The `Calendar` and `GregorianCalendar` classes include the methods necessary to design programs for such purposes. Let's look at an example.

In the northern hemisphere, summer usually begins on June 21. The program `DaysToSummer` provides some information that might be of interest to sun seekers (see Figure 3).

```

1  import java.util.*;
2
3  public class DaysToSummer
4  {
5      public static void main(String[] args)
6      {
7          Calendar today = new GregorianCalendar();
8
9          Calendar summerStart = new
10             GregorianCalendar(today.get(Calendar.YEAR), Calendar.JUNE, 21);
11
12             int days = summerStart.get(Calendar.DAY_OF_YEAR) -
13                 today.get(Calendar.DAY_OF_YEAR);
14
15             if (days > 0)
16                 System.out.println(days + " day(s) to summer");
17             else if (days == 0)
18                 System.out.println("Summer begins today!");
19             else
20                 System.out.println("Summer began " + -days + " day(s) ago");
21         }
22     }

```

Figure 3. `DaysToSummer.java`

There are three possible outputs from this program. If it is executed between January 1 and June 20 in any year, the output is

`n days to summer`

where n is the number of days to the beginning of summer. If it is executed on June 21, the output is

`Summer begins today!`

If it is executed after June 21, the output is

`Summer began n day(s) ago`

where n is the number of days since summer started.

The program begins by creating two `Calendar` objects. The first is `today`, representing the current day (line 7), and the second is `summerStart` representing June 21 in the current year (lines 10-11). Both objects bear the characteristics of an instance in time according to the rules of the Gregorian calendar. Two different constructors are used. If the default constructor (no arguments) is used, the instantiated object represents the current date and time. The constructor in line 10 uses three arguments. The first is the expression `today.get(Calendar.YEAR)`, representing the current year. The second is `Calendar.JUNE`, which is a public data field of the `Calendar` class. It happens to be an `int` data constant. The specific value is irrelevant for our purposes. The third argument is 21, an integer representing the day of the month when summer begins.

The number of days until the beginning of summer is calculated in lines 12-13 using two calls to the `get()` method, with `Calendar.DAY_OF_YEAR` as an argument. These return integers equal to the "day number in the year" of the `Calendar` objects specified: `summerStart` and `today`. They are subtracted and assigned to the `int` variable `days`. There are three possibilities for `days`. It is positive if summer has yet to arrive. It is negative if summer has

already started. It is zero if today is the first day of summer. These three possibilities are tested for in lines 15-17, and an appropriate message is printed.

Class Hierarchy - Date and Time Classes

Figure 4 illustrates the Java's date and time class in a class hierarchy diagram. The `TimeZone`, `SimpleTimeZone`, and `Locale` classes were not discussed in this section. (You may want to investigate these on your own.)

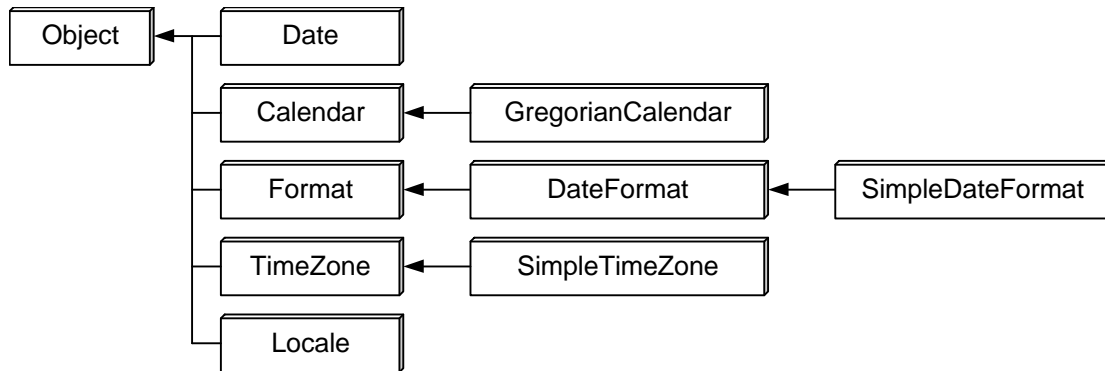


Figure 4. Class hierarchy - date and time classes