JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/jw-12-2000/jw-1229-dates.html

Calculating Java dates

Take the time to learn how to create and use dates

By Robert Nielsen, JavaWorld.com, 12/29/00

To keep track of time, Java counts the number of milliseconds from the start of January 1, 1970. This means, for example, that January 2, 1970, began 86,400,000 milliseconds later. Similarly, December 31, 1969, began 86,400,000 milliseconds before January 1, 1970. The Java `Date` class keeps track of those milliseconds as a `long` value. Because `long` is a signed number, dates can be expressed before and after the start of January 1, 1970. The largest positive and negative values expressible by the `long` primitive can generate dates forward and backward about 290,000,000 years, which suits most people's schedules.

The Date class

The `Date` class, found in the `java.util` package, encapsulates a `long` value representing a specific moment in time. One useful constructor is `Date()`, which creates a `Date` object representing the time the object was created. The `getTime()` method returns the `long` value of a `Date` object. In the program below, I use the `Date()` constructor to create a date based on when the program was run, and the `getTime()` method to find out the number of milliseconds that the date represents:

```
import java.util.*;
public class Now {
   public static void main(String[] args) {
      Date now = new Date();
      long nowLong = now.getTime();
      System.out.println("Value is " + nowLong);
   }
}
```

When I ran that program, it gave me a value of 972,568,255,150. A quick check with my calculator confirms this number is at least in the correct ballpark: it's a bit less than 31 years, which corresponds to the right number of years between January 1, 1970, and the day I wrote this article. While computers may thrive on numbers like the foregoing value, most people are reluctant to say such things as "I'll see you on 996,321,998,346." Fortunately, Java provides a way to convert `Date` objects to `Strings`, which represent dates in more traditional ways. The `DateFormat` class, discussed in the next section, can create `Strings` with alacrity.

The DateFormat class

One purpose of the `DateFormat` class is to create `Strings` in ways that humans can easily deal with them. However, because of language differences, not all people want to see a date in exactly the same way. Someone in France may prefer to see "25 decembre 2000," while someone in the United States may be more accustomed to seeing "December 25, 2000." So when an instance of a `DateFormat` class is created, the object contains information concerning the particular format in which the date is to be displayed. To use the default format of the user's computer, you can apply the `getDateInstance` method in the following way to create the appropriate `DateFormat` object:

```
DateFormat df = DateFormat.getDateInstance();
```

The `DateFormat` class is found in the `java.text` package.

Converting to a String

You can convert a `Date` object to a string with the `format` method. This is shown in the following demonstration program:

```
import java.util.*;
import java.text.*;
public class NowString {
   public static void main(String[] args) {
      Date now = new Date();
      DateFormat df = DateFormat.getDateInstance();
      String s = df.format(now);
      System.out.println("Today is " + s);
   }
}
```

The `getDateInstance` method shown in the code above, with no arguments, creates an object in the default format or style. Java also provides some alternative styles for dates, which you can obtain through the overloaded `getDateInstance(int style)`. For convenience' sake, `DateFormat` provides some ready-made constants that you can use as arguments in the `getDateInstance` method. Some examples are `SHORT`, `MEDIUM`, `LONG`, and `FULL`, which are demonstrated in the program below:

```
import java.util.*;
import java.text.*;
public class StyleDemo {
   public static void main(String[] args) {
      Date now = new Date();
      DateFormat df =  DateFormat.getDateInstance();
      DateFormat df1 = DateFormat.getDateInstance(DateFormat.SHORT);
      DateFormat df2 = DateFormat.getDateInstance(DateFormat.MEDIUM);
      DateFormat df3 = DateFormat.getDateInstance(DateFormat.LONG);
      DateFormat df4 = DateFormat.getDateInstance(DateFormat.FULL);
      String s =  df.format(now);
      String s1 = df1.format(now);
      String s2 = df2.format(now);
      String s3 = df3.format(now);
      String s4 = df4.format(now);
      System.out.println("(Default) Today is " + s);
      System.out.println("(SHORT)   Today is " + s1);
      System.out.println("(MEDIUM)  Today is " + s2);
      System.out.println("(LONG)    Today is " + s3);
      System.out.println("(FULL)    Today is " + s4);
   }
}
```

That program output the following:

```
(Default) Today is Nov 8, 2000
(SHORT)   Today is 11/8/00
(MEDIUM)  Today is Nov 8, 2000
(LONG)    Today is November 8, 2000
(FULL)    Today is Wednesday, November 8, 2000
```

The same program, after being run on my computer with the default regional settings changed to Swedish, displayed this output:

```
(Default) Today is 2000-nov-08
(SHORT)   Today is 2000-11-08
(MEDIUM)  Today is 2000-nov-08
(LONG)    Today is den 8 november 2000
(FULL)    Today is den 8 november 2000
```

From that, you can see that in Swedish the months of the year are not capitalized (although November is still november). Also, note that the `LONG` and `FULL` versions are identical in Swedish, while they differ in American English. Additionally, it is interesting that the Swedish word for Wednesday, *onsdag*, is not included in the `FULL` version, where the English `FULL` version includes the name of the day.

Note that you can use the `getDateInstance` method to change the language for a `DateFormat` instance; however, in the case above, it was done on a Windows 98 machine by changing the regional settings from the control panel. The lesson here is that the default regional setting varies from place to place, which has both advantages and disadvantages of which the Java programmer should be aware. One advantage is that the Java programmer can write a single line of code to display a date, yet have the date appear in tens or even hundreds of different forms when the program is run on computers throughout the world. But that can be a disadvantage if the programmer wants just one format -- which is preferable, for example, in a program that outputs text and dates mixed together. If the text is in English, it would be inconsistent to have dates in other formats, such as German or Spanish. If the programmer relies on the default regional format, the date format will vary according to the executing computer's regional settings.

Parsing a String

You can also use the `DateFormat` class to create `Date` objects from a `String`, via the `parse()` method. This particular method can throw a `ParseException` error, so you must use proper error-handling techniques. A sample program that turns a `String` into a `Date` is shown below:

```
import java.util.*;
import java.text.*;
public class ParseExample {
   public static void main(String[] args) {
      String ds = "November 1, 2000";
      DateFormat df = DateFormat.getDateInstance();
      try {
         Date d = df.parse(ds);
      }
      catch(ParseException e) {
         System.out.println("Unable to parse " + ds);
      }
   }
}
```

The `parse()` method is a useful tool for creating arbitrary dates. I will examine another way of creating arbitrary dates. Also, you will see how to do elementary calculations with dates, such as calculating the date 90 days after another date. You can accomplish both tasks with the `GregorianCalendar` class.

The GregorianCalendar class

One way to create an object representing an arbitrary date is to use the following constructor of the `GregorianCalendar` class, found in the `java.util` package:

```
GregorianCalendar(int year, int month, int date)
```

Note that for the month, January is 0, February is 1, and so on, until December, which is 11. Since those are not the numbers most of us associate with the months of the year, programs will probably be more readable if they use the constants of the parent `Calendar` class: `JANUARY`, `FEBRUARY`, and so on. So, to create an object representing the date that Wilbur and Orville Wright first flew their motored aircraft (December 17, 1903), you can use:

```
GregorianCalendar firstFlight = new GregorianCalendar(1903, Calendar.DECEMBER, 17);
```

For clarity's sake, you should use the preceding form. However, you should also learn how to read the shorter form, below. The following example represents the same December 17, 1903, date (remember, in the shorter form *11* represents December):

```
GregorianCalendar firstFlight = new GregorianCalendar(1903, 11, 17);
```

In the previous section, you learned how to turn `Date` objects into `Strings`. You will do the same again; but first, you need to convert a `GregorianCalendar` object to a `Date`. To do so, you will use the `getTime()` method, which `GregorianCalendar` inherits from its parent `Calendar` class. The `getTime()` method returns a `Date` corresponding to a `GregorianCalendar` object. You can put the whole process of creating a `GregorianCalendar` object, converting it to a `Date`, and getting and outputting the corresponding `String` in the following program:

```
import java.util.*;
import java.text.*;
public class Flight {
   public static void main(String[] args) {
      GregorianCalendar firstFlight = new GregorianCalendar(1903, Calendar.DECEMBER,
      Date d = firstFlight.getTime();
      DateFormat df = DateFormat.getDateInstance();
      String s = df.format(d);
      System.out.println("First flight was " + s);
   }
}
```

Sometimes it is useful to create an instance of the `GregorianCalendar` class representing the day the instance was created. To do so, simply use the `GregorianCalendar` constructor taking no arguments, such as:

```
GregorianCalendar thisday = new GregorianCalendar();
```

A sample program to output today's date, starting with a `GregorianCalendar` object is:

```
import java.util.*;
import java.text.*;
class Today {
   public static void main(String[] args) {
      GregorianCalendar thisday = new GregorianCalendar();      Date d = thisday.ge
      DateFormat df = DateFormat.getDateInstance();
      String s = df.format(d);
      System.out.println("Today is " + s);
   }
}
```

Note the similarities between the `Date()` constructor and the `GregorianCalendar()` constructor: both create an object, which in simple terms, represents today.

Date manipulation

The `GregorianCalendar` class offers methods for manipulating dates. One useful method is `add()`. With the `add()` method, you can add such time units as years, months, and days to a date. To use the `add()` method, you must supply the field being increased, and the integer amount by which it will increase. Some useful constants for the fields are `DATE`, `MONTH`, `YEAR`, and `WEEK_OF_YEAR`. The `add()` method is used in the program below to calculate a date 80 days in the future. Phileas Fogg, the central character in Jules Verne's *Around the World in 80 Days,* could have used such a program to calculate a date 80 days from his departure on October 2, 1872:

```
import java.util.*;
import java.text.*;
public class World {
   public static void main(String[] args) {
      GregorianCalendar worldTour = new GregorianCalendar(1872, Calendar.OCTOBER, 2)
      worldTour.add(GregorianCalendar.DATE, 80);
      Date d = worldTour.getTime();
      DateFormat df = DateFormat.getDateInstance();
      String s = df.format(d);
      System.out.println("80 day trip will end " + s);
   }
}
```

While the example is a bit fanciful, adding days to a date is a common operation: video rentals can be due in 3 days, a library may lend books for 21 days, stores frequently require purchased items to be exchanged within 30 days. The following program shows a calculation using years:

```
import java.util.*;
import java.text.*;
public class Mortgage {
   public static void main(String[] args) {
      GregorianCalendar mortgage = new GregorianCalendar(1997, Calendar.MAY, 18);
      mortgage.add(Calendar.YEAR, 15);
      Date d = mortgage.getTime();
      DateFormat df = DateFormat.getDateInstance();
      String s = df.format(d);
      System.out.println("15 year mortgage amortized on " + s);    }
}
```

One important side effect of the `add()` method is that it changes the original date. Sometimes it is important to have both the original date and the modified date. Unfortunately, you cannot simply create a new `GregorianCalendar` object set equal to the original. The reason is that the two variables have a reference to one date. If the date is changed, both variables now refer to the changed date. Instead, a new object should be created. The following example will demonstrate this:

```
import java.util.*;
import java.text.*;
public class ThreeDates {
   public static void main(String[] args) {
      GregorianCalendar gc1 = new GregorianCalendar(2000, Calendar.JANUARY, 1);
      GregorianCalendar gc2 = gc1;
      GregorianCalendar gc3 = new GregorianCalendar(2000, Calendar.JANUARY, 1);
      //Three dates all equal to January 1, 2000
      gc1.add(Calendar.YEAR, 1);
      //gc1 and gc2 are changed
      DateFormat df = DateFormat.getDateInstance();
      Date d1 = gc1.getTime();
      Date d2 = gc2.getTime();
      Date d3 = gc3.getTime();
      String s1 = df.format(d1);
      String s2 = df.format(d2);
      String s3 = df.format(d3);
      System.out.println("gc1 is " + s1);
      System.out.println("gc2 is " + s2);
      System.out.println("gc3 is " + s3);
   }
}
```

After the program is run, `gc1` and `gc2` are changed to the year 2001 (because both objects are pointing to the same underlying date representation, which has been changed). The object `gc3` is pointing to a separate underlying date representation, which has not been changed.

Calculating review dates

At this point, you have a program based on a real-world example. This particular program will calculate the dates for reviewing material. For example, while reading this article, you might come across some points you want to remember. Unless you have photographic memory, you will likely find that periodically reviewing the new material will aid in remembering it. One reviewing system, discussed in Kurt Hanks and Gerreld L. Pulsipher's *Five Secrets to Personal Productivity,* suggests briefly covering the material immediately after it is first seen, then after one day, one week, one month, three months, and one year. For this article, you would give it a quick review immediately, another tomorrow, then one week, one month, three months, and one year from now. Our program will calculate the dates.

To be most useful, the program would be incorporated in PIM (Personal Information Manager) software, which would run the program and schedule the reviews. The `getDates()` method in the following `ReviewDates` program would be useful for integrating with electronic software as it returns an array of dates (the review dates). Additionally, you can return individual dates using the methods `getFirstDay()`, `getOneDay()`, `getOneWeek()`, `getOneMonth()`, and `getOneYear()`. While it is beyond the scope of this article to integrate the `ReviewDates` class with a PIM, the `ReviewDates` class does show how to calculate dates based on elapsed time. Now you can easily modify it for other operations that require elapsed time, such as library loans, tape rentals, and mortgage calculations. First, the `ReviewDates` class is shown below:

```
import java.util.*;
import java.text.*;
public class ReviewDates {
   private GregorianCalendar firstDay, oneDay, oneWeek, oneMonth, oneQuarter, oneYea
   final int dateArraySize = 6;
   ReviewDates(GregorianCalendar gcDate) {
      int year = gcDate.get(GregorianCalendar.YEAR);
      int month = gcDate.get(GregorianCalendar.MONTH);
      int date = gcDate.get(GregorianCalendar.DATE);
      firstDay = new GregorianCalendar(year, month, date);
      oneDay = new GregorianCalendar(year, month, date);
      oneWeek = new GregorianCalendar(year, month, date);
      oneMonth = new GregorianCalendar(year, month, date);
      oneQuarter = new GregorianCalendar(year, month, date);
      oneYear = new GregorianCalendar(year, month, date);
      oneDay.add(GregorianCalendar.DATE, 1);
      oneWeek.add(GregorianCalendar.DATE, 7);
      oneMonth.add(GregorianCalendar.MONTH, 1);
      oneQuarter.add(GregorianCalendar.MONTH, 3);
      oneYear.add(GregorianCalendar.YEAR, 1);
   }
   ReviewDates() {
      this(new GregorianCalendar());
   }
   public void listDates() {
      DateFormat df = DateFormat.getDateInstance(DateFormat.LONG);
      Date startDate = firstDay.getTime();
      Date date1 = oneDay.getTime();
      Date date2 = oneWeek.getTime();
      Date date3 = oneMonth.getTime();
      Date date4 = oneQuarter.getTime();
      Date date5 = oneYear.getTime();
      String ss =  df.format(startDate);
      String ss1 = df.format(date1);
      String ss2 = df.format(date2);
      String ss3 = df.format(date3);
      String ss4 = df.format(date4);
      String ss5 = df.format(date5);
      System.out.println("Start date is " + ss);
      System.out.println("Following review dates are:");
      System.out.println(ss1);
      System.out.println(ss2);
      System.out.println(ss3);
      System.out.println(ss4);
      System.out.println(ss5);
      System.out.println();
   }
   public GregorianCalendar[] getDates() {
      GregorianCalendar[] memoryDates = new GregorianCalendar[dateArraySize];
      memoryDates[0] = firstDay;
      memoryDates[1] = oneDay;
      memoryDates[2] = oneWeek;
      memoryDates[3] = oneMonth;
      memoryDates[4] = oneQuarter;
      memoryDates[5] = oneYear;
      return memoryDates;
   }
   public GregorianCalendar getFirstDay() {
      return this.firstDay;
   }
   public GregorianCalendar getOneDay() {
      return this.oneDay;
   }
   public GregorianCalendar getOneWeek() {
      return this.oneWeek;
   }
   public GregorianCalendar getOneMonth() {
      return this.oneMonth;
   }
   public GregorianCalendar getOneQuarter() {
      return this.oneQuarter;
   }
   public GregorianCalendar getOneYear() {
      return this.oneYear;
   }
}
```

An example of a program that uses the `ReviewDates` class to make a simple listing of review dates is shown below:

```
import java.util.*;
public class ShowDates {
   public static void main(String[] args) {
      ReviewDates rd = new ReviewDates();
      rd.listDates();
      GregorianCalendar gc = new GregorianCalendar(2001, Calendar.JANUARY, 15);
      ReviewDates jan15 = new ReviewDates(gc);
      jan15.listDates();
   }
}
```

Conclusion

This article has introduced three important classes for working with dates: `Date`, `DateFormat`, and `GregorianCalendar`. These classes let you create dates, change them into `String`s, and make elementary calculations with dates. In terms of working with dates in Java, this article has only scratched the surface. However, the classes and methods that I have introduced here not only serve as springboard for more advanced learning, but in themselves can also handle many common date-related tasks.

About the author

Robert Nielsen is a Sun-Certified Java 2 Programmer. He holds a master's degree in education with a specialty in computer-assisted instruction, and has taught for several years. He has also published computer-related articles in a variety of magazines.