WORKED EXAMPLE 2.1 How Many Days Have You Been Alive?

Many programs need to process dates such as "February 15, 2010". The `worked_example_1` directory of this chapter's companion code contains a `Day` class that was designed to work with calendar days.

The `Day` class knows about the intricacies of our calendar, such as the fact that January has 31 days and February has 28 or sometimes 29. The Julian calendar, instituted by Julius Caesar in the first century BCE, introduced the rule that every fourth year is a leap year. In 1582, Pope Gregory XIII ordered the implementation of the calendar that is in common use throughout the world today, called the Gregorian calendar. It refines the leap year rule by specifying that years divisible by 100 are not leap years, unless they are divisible by 400. Thus, the year 1900 was not a leap year but the year 2000 was. All of these details are handled by the internals of the `Day` class.

The `Day` class lets you answer questions such as

- How many days are there between now and the end of the year?
- What day is 100 days from now?

Problem Statement Your task is to write a program that determines how many days you have been alive. You should *not* look inside the internal implementation of the `Day` class. Use the API documentation by pointing your browser to the file `index.html` in the `ch02/worked_example_1/api` subdirectory.

As you can see from the API documentation (see figure on next page), you construct a `Day` object from a given year, month, and day, like this:

```
Day jamesGoslingsBirthday = new Day(1955, 5, 19);
```

There is a method for adding days to a given day, for example:

```
Day later = jamesGoslingsBirthday.addDays(100);
```

You can then find out what the result is, by applying the `getYear/getMonth/getDate` methods:

```
System.out.println(later.getYear());
System.out.println(later.getMonth());
System.out.println(later.getDate());
```

However, that approach does not solve our problem (unless you are willing to replace 100 with other values until, by trial and error, you obtain today's date). Instead, use the `daysFrom` method. According to the API documentation, we need to supply another day. That is, the method is called like this:

```
int daysAlive = day1.daysFrom(day2);
```

In our situation, one of the `Day` objects is `jamesGoslingsBirthday`, and the other is today's date. This can be obtained with the constructor that has no arguments:

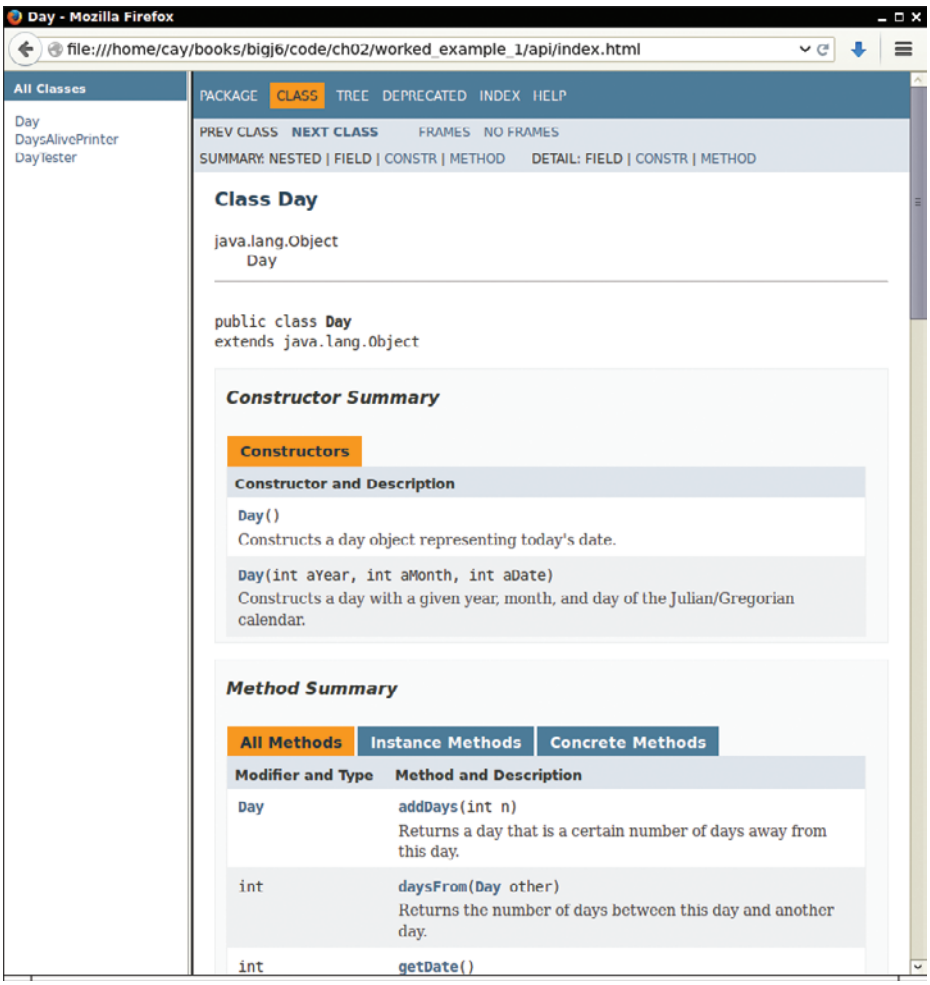
```
Day today = new Day();
```

We have two candidates on which the `daysFrom` method could be invoked, yielding the call

```
int daysAlive = jamesGoslingsBirthday.daysFrom(today);
```

or

```
int daysAlive = today.daysFrom(jamesGoslingsBirthday);
```



Which is the right choice? Fortunately, the author of the `Day` class has anticipated this question. The detail comment of the `daysFrom` method contains this statement:

> `Returns:` the number of days that this day is away from the other
> (larger than `0` if this day comes later than `other`)

We want a positive result. Therefore, the second form is the correct one.

Here is the program that solves our problem (see `ch02/worked_example_1` in your source code):

worked_example_1/DaysAlivePrinter.java

```
1  public class DaysAlivePrinter
2  {
3     public static void main(String[] args)
4     {
5        Day jamesGoslingsBirthday = new Day(1955, 5, 19);
6        Day today = new Day();
7        System.out.print("Today: ");
8        System.out.println(today.toString());
9        int daysAlive = today.daysFrom(jamesGoslingsBirthday);
```

```
10         System.out.print("Days alive: ");
11         System.out.println(daysAlive);
12      }
13   }
```

Program Run

```
Today: 2015-02-09
Days alive: 21826
```