Package java.time

The main API for dates, times, instants, and durations.

See: Description

Class Summary

Class	Description
Clock	A clock providing access to the current instant, date and time using a time-zone.
Duration	A time-based amount of time, such as '34.5 seconds'.
Instant	An instantaneous point on the time-line.
LocalDate	A date without a time-zone in the ISO-8601 calendar system, such as `2007-12-03`.
LocalDateTime	A date-time without a time-zone in the ISO-8601 calendar system, such as `2007-12-03T10:15:30`.
LocalTime	A time without a time-zone in the ISO-8601 calendar system, such as `10:15:30`.
MonthDay	A month-day in the ISO-8601 calendar system, such as `--12-03`.
OffsetDateTime	A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as `2007-12-03T10:15:30+01:00`.
OffsetTime	A time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as `10:15:30+01:00`.
Period	A date-based amount of time in the ISO-8601 calendar system, such as '2 years, 3 months and 4 days'.
Year	A year in the ISO-8601 calendar system, such as `2007`.
YearMonth	A year-month in the ISO-8601 calendar system, such as `2007-12`.
ZonedDateTime	A date-time with a time-zone in the ISO-8601 calendar system, such as `2007-12-03T10:15:30+01:00 Europe/Paris`.

Enum Summary

Enum	Description
DayOfWeek	A day-of-week, such as 'Tuesday'.
Month	A month-of-year, such as 'July'.

Exception Summary

Exception	Description
DateTimeException	Exception used to indicate a problem while calculating a date-time.

Package java.time Description

The main API for dates, times, instants, and durations.

The classes defined here represent the principle date-time concepts, including instants, durations, dates, times, time-zones and periods. They are based on the ISO calendar system, which is the *de facto* world calendar following the proleptic Gregorian rules. All the classes are immutable and thread-safe.

Each date time instance is composed of fields that are conveniently made available by the APIs. For lower level access to the fields refer to the `java.time.temporal` package. Each class includes support for printing and parsing all manner of dates and times. Refer to the `java.time.format` package for customization options.

The `java.time.chrono` package contains the calendar neutral API `ChronoLocalDate`, `ChronoLocalDateTime`, `ChronoZonedDateTime` and `Era`. This is intended for use by applications that need to use localized calendars. It is recommended that applications use the ISO-8601 date and time classes from this package across system boundaries, such as to the database or across the network. The calendar neutral API should be reserved for interactions with users.

Dates and Times

`Instant` is essentially a numeric timestamp. The current Instant can be retrieved from a `Clock`. This is useful for logging and persistence of a point in time and has in the past been associated with storing the result from `System.currentTimeMillis()`.

`LocalDate` stores a date without a time. This stores a date like '2010-12-03' and could be used to store a birthday.

`LocalTime` stores a time without a date. This stores a time like '11:30' and could be used to store an opening or closing time.

`LocalDateTime` stores a date and time. This stores a date-time like '2010-12-03T11:30'.

`ZonedDateTime` stores a date and time with a time-zone. This is useful if you want to perform accurate calculations of dates and times taking into account the `ZoneId`, such as 'Europe/Paris'. Where possible, it is recommended to use a simpler class without a time-zone. The widespread use of time-zones tends to add considerable complexity to an application.

Duration and Period

Beyond dates and times, the API also allows the storage of periods and durations of time. A `Duration` is a simple measure of time along the time-line in nanoseconds. A `Period` expresses an amount of time in units meaningful to humans, such as years or days.

Additional value types

`Month` stores a month on its own. This stores a single month-of-year in isolation, such as 'DECEMBER'.

`DayOfWeek` stores a day-of-week on its own. This stores a single day-of-week in isolation, such as 'TUESDAY'.

`Year` stores a year on its own. This stores a single year in isolation, such as '2010'.

`YearMonth` stores a year and month without a day or time. This stores a year and month, such as '2010-12' and could be used for a credit card expiry.

`MonthDay` stores a month and day without a year or time. This stores a month and day-of-month, such as '--12-03' and could be used to store an annual event like a birthday without storing the year.

`OffsetTime` stores a time and offset from UTC without a date. This stores a date like '11:30+01:00'. The `ZoneOffset` is of the form '+01:00'.

`OffsetDateTime` stores a date and time and offset from UTC. This stores a date-time like '2010-12-03T11:30+01:00'. This is sometimes found in XML messages and other forms of persistence, but contains less information than a full time-zone.

Package specification

Unless otherwise noted, passing a null argument to a constructor or method in any class or interface in this package will cause a `NullPointerException` to be thrown. The Javadoc "@param" definition is used to summarise the null-behavior. The "@throws `NullPointerException`" is not explicitly documented in each method.

All calculations should check for numeric overflow and throw either an `ArithmeticException` or a `DateTimeException`.

Design notes (non normative)

The API has been designed to reject null early and to be clear about this behavior. A key exception is any method that takes an object and returns a boolean, for the purpose of checking or validating, will generally return false for null.

The API is designed to be type-safe where reasonable in the main high-level API. Thus, there are separate classes for the distinct concepts of date, time and date-time, plus variants for offset and time-zone. This can seem like a lot of classes, but most applications can begin with just five date/time types.

- `Instant` - a timestamp
- `LocalDate` - a date without a time, or any reference to an offset or time-zone
- `LocalTime` - a time without a date, or any reference to an offset or time-zone
- `LocalDateTime` - combines date and time, but still without any offset or time-zone
- `ZonedDateTime` - a "full" date-time with time-zone and resolved offset from UTC/Greenwich

The API has a relatively large surface area in terms of number of methods. This is made manageable through the use of consistent method prefixes.

- `of` - static factory method
- `parse` - static factory method focussed on parsing
- `get` - gets the value of something
- `is` - checks if something is true
- `with` - the immutable equivalent of a setter
- `plus` - adds an amount to an object
- `minus` - subtracts an amount from an object
- `to` - converts this object to another type
- `at` - combines this object with another, such as `date.atTime(time)`

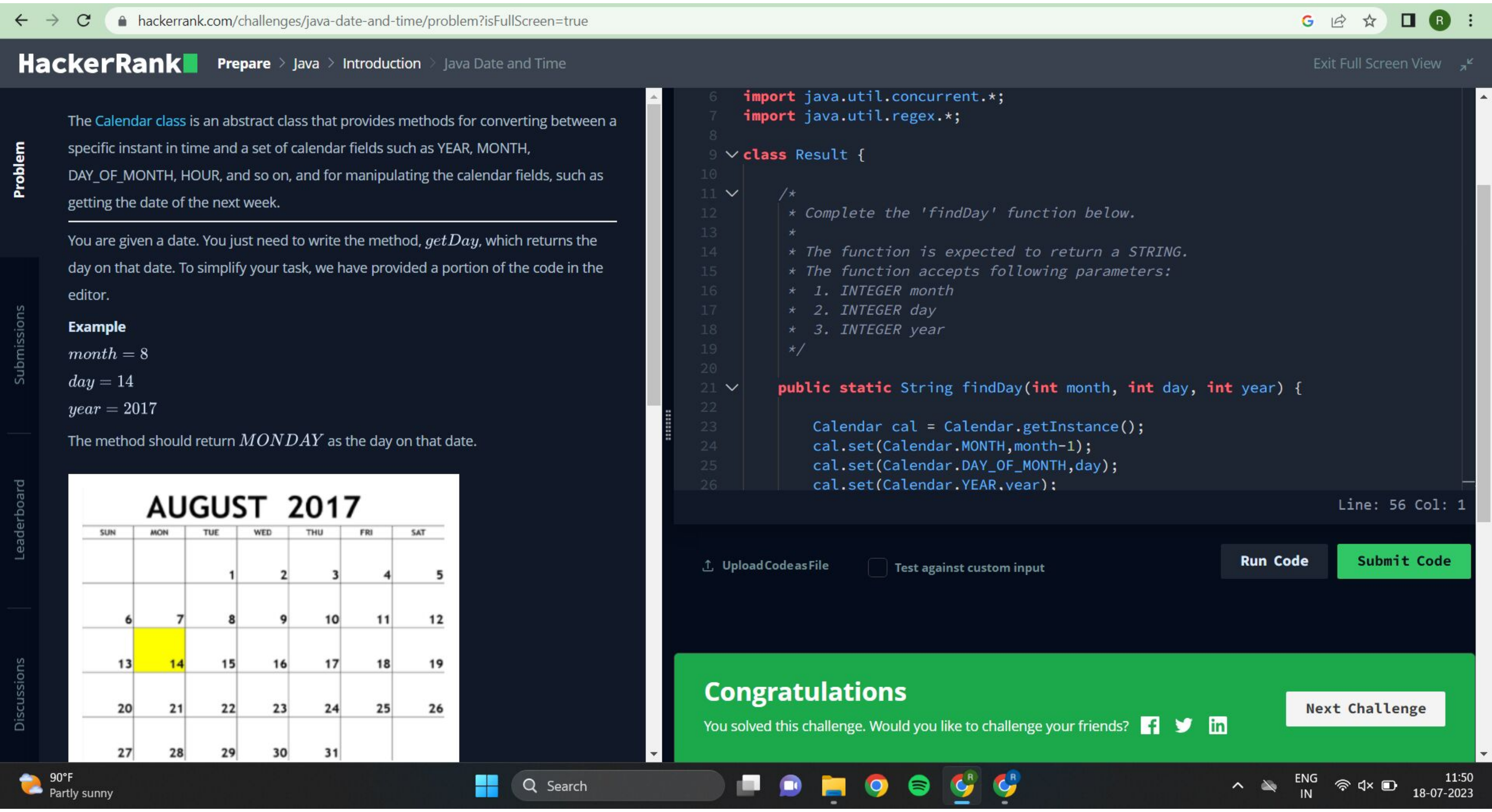
Multiple calendar systems is an awkward addition to the design challenges. The first principle is that most users want the standard ISO calendar system. As such, the main classes are ISO-only. The second principle is that most of those that want a non-ISO calendar system want it for user interaction, thus it is a UI localization issue. As such, date and time objects should be held as ISO objects in the data model and persistent storage, only being converted to and from a local calendar for display. The calendar system would be stored separately in the user preferences.

There are, however, some limited use cases where users believe they need to store and use dates in arbitrary calendar systems throughout the application. This is supported by `ChronoLocalDate`, however it is vital to read all the associated warnings in the Javadoc of that interface before using it. In summary, applications that require general interoperation between multiple calendar systems typically need to be written in a very different way to those only using the ISO calendar, thus most applications should just use ISO and avoid `ChronoLocalDate`.

The API is also designed for user extensibility, as there are many ways of calculating time. The field and unit API, accessed via `TemporalAccessor` and `Temporal` provide considerable flexibility to applications. In addition, the `TemporalQuery` and `TemporalAdjuster` interfaces provide day-to-day power, allowing code to read close to business requirements:

```
LocalDate customerBirthday = customer.loadBirthdayFromDatabase();
LocalDate today = LocalDate.now();
if (customerBirthday.equals(today)) {
  LocalDate specialOfferExpiryDate = today.plusWeeks(2).with(next(FRIDAY));
  customer.sendBirthdaySpecialOffer(specialOfferExpiryDate);
}
```

Since:
JDK1.8



hackerrank.com/challenges/java-date-and-time/problem?isFullScreen=true

HackerRank Prepare > Java > Introduction > Java Date and Time   Exit Full Screen View

The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.

You are given a date. You just need to write the method, $getDay$, which returns the day on that date. To simplify your task, we have provided a portion of the code in the editor.

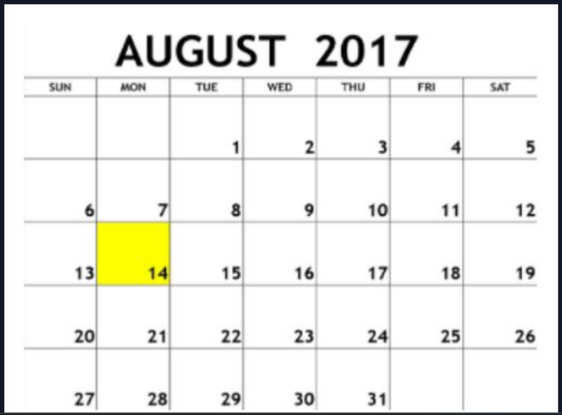
Example

$month = 8$

$day = 14$

$year = 2017$

The method should return $MONDAY$ as the day on that date.


AUGUST 2017

SUN	MON	TUE	WED	THU	FRI	SAT
		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	31		

```java
import java.util.concurrent.*;
import java.util.regex.*;

class Result {

    /*
     * Complete the 'findDay' function below.
     *
     * The function is expected to return a STRING.
     * The function accepts following parameters:
     *  1. INTEGER month
     *  2. INTEGER day
     *  3. INTEGER year
     */

    public static String findDay(int month, int day, int year) {

        Calendar cal = Calendar.getInstance();
        cal.set(Calendar.MONTH,month-1);
        cal.set(Calendar.DAY_OF_MONTH,day);
        cal.set(Calendar.YEAR,year);
```

Line: 56 Col: 1

Upload Code as File   Test against custom input   Run Code   Submit Code

Congratulations

You solved this challenge. Would you like to challenge your friends?

Next Challenge