<https://habrahabr.ru/post/274905/>

# Java и время: часть вторая

* [Разработка веб-сайтов](https://habrahabr.ru/hub/webdev/),
* [Программирование](https://habrahabr.ru/hub/programming/),
* [Java](https://habrahabr.ru/hub/java/)
* Tutorial

Эта статья написана в продолжение к [первой части](http://habrahabr.ru/post/274811/) и посвящена [новому Date Time API](https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html), который был введен в Java 8. Я изначально хотел оформить эту тему отдельно, поскольку она достаточно большая и серьезная. Я еще сам не в полной мере начал использовать этот API в проектах, поэтому разбираться будем вместе по ходу. В принципе в переходе на новый API нет никакой срочной необходимости, более того многие еще и не начинали проекты на Java 8, а это означает, что время на освоение еще есть.  
  
В статье я постараюсь не скатываться в банальный перевод штатной документации, больше я хотел бы сконцентрироваться на том, что мне показалось особенно важным.

# История

Что касается работы с временем — нарекания к стандартной библиотеке Java были давно. Критикуемая версия API разрабатывалась очень давно и при ее проектировании были допущены серьезные ошибки. В качестве альтернативы многие использовали стороннюю библиотеку Joda-time. Сам я не очень большой поклонник Joda-time по нескольким причинам:

* классов стандартной библиотеки все равно не избежать, в 99% случаях их функциональность с задачей справляется, а умножать сущности сверх необходимости не хочется;
* библиотека joda-time не использует стандартную базу временных зон из JVM, поэтому при очередном маневре законодателей приходится помнить, что обновлять tzdata нужно не только в JDK, но [и в библиотеке joda-time](http://joda-time.sourceforge.net/tz_update.html).

# Сравнение

Начать наверное стоит с того, что именно не устраивало многих в старом API. И тут же, чтобы не терять время сразу укажу, что в новом API изменилось к лучшему.  
  
Разбиение классов по пакетам:

* В старом API классы для работы со временем усажены в пакеты java.util и java.sql — среди большого множества других классов. Кроме того существуют еще классы java.util.concurrent.TimeUnit и java.text.DateFormat c наследниками.
* В новом API для работы с временем выделен отдельный пакет java.time

Названия классов:

* Названия классов в старом API не отражают суть происходящего. В старом API есть два класса которые способны обозначить точку на временной оси: java.util.Date и java.util.Calendar. Класс java.util.Date обозначает время в миллисекундах по Unix-time, а вовсе не дату (он назван так скорее всего из тех же соображений, по которым время в командной строке выдает утилита /bin/date). Класс java.util.Calendar также вовсе не календарь, у него есть состояние в виде временной зоны, календарных и временных полей.
* В новом API названия классов даны более осмысленно. Есть классы аналогичные уже упомянутым: java.time.Instant и java.time.ZonedDateTime. Существует также множество других классов для более специализированного использования.

Неизменяемость и потокобезопасность:

* Класс java.util.Date не является immutable и отягощен большим количеством лишних методов, которые хоть уже и помечены как устаревшие, но вряд ли будут удалены в обозримом будущем. Изменяемость java.util.Date заставляет некоторых клонировать инстанты java.util.Date — для того чтобы враг не пробрался:
* public class UserBean {
* private final Date created;
* public UserBean(Date created) {
* this.created = (Date) created.clone();
* }
* }

Класс java.util.Calendar также изменяем. Хотя это особых проблем это не доставляет поскольку большинство понимает что у него есть внутреннее состояние которое меняется, да и передавать его аргументами как-то не очень принято.  
  
Поскольку классы в старом API изменяемые, использовать их в многопоточной среде нужно с осторожностью. В частности java.util.Date можно признать «эффективно» потоко-безопасным, если вы не вызываете у него устаревшие методы.

* Все классы в новом API неизменяемые и как следствие потоко-безопасные

Точность:

* Точность представления в времени составляет одну миллисекунду. Для большинства практических задач этого более чем достаточно, но иногда хочется иметь точность повыше.
* В новом API представления времени составляет одну наносекунду, что в миллион раз точнее.

Хранение меток времени и даты:

* Классы для меток времени и даты (java.sql.Date и java.sql.Time) не являются чистым представлением меток времени и даты, поскольку унаследованы от java.util.Date и так или иначе хранят полное значение Unix-time с игнорированием части этого значения.
* В новом API соответствующие классы java.time.LocalDate и java.time.LocalTime хранят чистые кортежи (yyyy,MM,dd) и (HH,mm,dd) соответственно, и никакой лишней информации или логики в этих классах нет. Также введен класс java.time.LocalDateTime который хранит оба кортежа.

Указание временной зоны:

* В старом API многие действия, где необходимо указание временной зоны, могут быть выполнены без ее указания. В этом случае берется временная зона по-умолчанию, а программист может даже и не догадаться о том, что он что-то упустил.
* В новом API все действия, где необходимо указание временной зоны, требуют ее явно: либо в виде аргумента метода, либо временная зона отображена прямо в названии метода. Другими словами временная зона «по-умолчанию» нигде по умолчанию не используется.

Тестирование:

* Старый API очень сложно использовать в тестах, в которых нужно протестировать поведение логики с течением времени (об этом подробно расписано в предыдущей статье).
* В новом API введен специальный абстрактный класс java.time.Clock, единый экземпляр которого можно инжектить в контекст или просто передавать в свою логику. Переопределив этот класс для тестов, можно контролировать течение времени для своего кода в ходе его выполнения.

Нумерация месяцев:

* В старом API номера месяцев идут с 0, что очень неинтуитивно.
* В новом API номера месяцев идут с 1. Появилось новое перечисление [java.time.Month](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html).

Установка меток:

* В java.util.Calendar устанавливала год-месяц-день-час-минуту-секунду, но для сброса миллисекунд нужно было сделать отдельный вызов.
* В java.time.ZonedDateTime устанавливаются все поля сразу, включая наносекунды.

Обозначение длительности:

* В старом API нет классов для определения длительности и промежутков времени. Обычно используется простой long и хранение длительности в виде миллисекунд.
* В новом API определены специальные классы для длительности и периодов.

# Опасения

Также наверное стоит рассказать о том, что я точно не стану называть «ухудшениями», а осторожно назову «опасениями»:

* если раньше было два активно используемых класса: java.util.Date и java.util.Calendar, то сейчас классов стало сильно больше, плюс к ним добавилась иерархия интерфейсов и абстрактных классов.
* отчасти из-за большого количества новых классов появились некоторые нюансы работы, которые я успел найти уже за несколько часов исследований и о которых мы поговорим позже.
* новый API не контролирует правильность операций в compile-time. Многие проблемы с отсутствием временной зоны будут проявляться только в runtime — как это далее будет видно в примерах. Я бы предпочел возможно менее гибкий, но более строгий контракт.
* в ходе работы генерируется большее количество объектов. Например получение текущей временной точки через Instance.now() кроме самого экземпляра java.time.Instance создает еще и java.time.Clock на каждый запрос, хотя это вообще ему ни к чему — в текущей реализации достаточно было бы вызова System.currentTimeMillis(). Также промежуточные объекты создаются и при многих других действиях. Но я не думаю, что для типичного бэкенда это будет представлять какую-нибудь проблему — ни по потреблению памяти, ни по времени исполнения. Хардкорщики все равно хранят время в long или даже пакуют в int.
* проблему (проблема ли это?) с учетом leap second никак не решили. Фактически новая библиотека все также не отходит ни на шаг от Unix-time и полагается на внешний перевод секундной стрелки назад. Прошлое при этом все также каждый раз теряет секунду и сдвигается вперед. Библиотеку для точных научных расчетов мы так и не получили.

Об этих и других, настороживших меня кейсах расскажу подробнее уже в примерах.

# Временные зоны

Начнем как обычно с временных зон. Новый класс [java.time.ZoneId](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html) обозначает временную зону. Два его сабкласса java.time.ZoneRegion и [java.time.ZoneOffset](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneOffset.html) реализуют два типа временных зон: временную зону по географическому принципу и временную зону по простому смещению относительно UTC, UT или GMT. Правила перевода стрелок вынесены в отдельных класс [java.time.zone.ZoneRules](https://docs.oracle.com/javase/8/docs/api/java/time/zone/ZoneRules.html), экземпляр которого доступен через метод java.time.ZoneId#getRules.  
  
В целом, кроме указанного рефакторинга, я не нашел тут особых принципиальных изменений, разве что правила перевода стрелок теперь предоставляют больше методов для запроса информации. Посему все написанное в [старой статье](http://habrahabr.ru/post/274811/) справедливо и для новых классов временных зон, разве что методы несколько отличаются по имени.

@Test

public void testZoneId() throws Exception {

*// case-1*

ZoneId zid1 = ZoneId.of("Europe/Moscow");

Assert.assertEquals("ZoneRegion", zid1.getClass().getSimpleName());

*// case-2*

ZoneId zid2 = ZoneId.of("UTC+4");

Assert.assertEquals("ZoneRegion", zid2.getClass().getSimpleName());

*// case-3*

ZoneId zid3 = ZoneId.of("+03:00:00");

Assert.assertEquals("ZoneOffset", zid3.getClass().getSimpleName());

*// case-4*

ZoneId zid4 = ZoneId.ofOffset("UTC", ZoneOffset.of("+03:00:00"));

Assert.assertEquals("ZoneRegion", zid4.getClass().getSimpleName());

}

Не очень понятно почему case-4, который фактически запрашивает тоже что и case-3, в результате создает java.time.ZoneRegion, а не java.time.ZoneOffset.

@Test

public void testZoneUTC() throws Exception {

ZoneId zid1 = ZoneOffset.UTC;

Assert.assertEquals("ZoneOffset", zid1.getClass().getSimpleName());

ZoneId zid2 = ZoneId.of("Z");

Assert.assertEquals("ZoneOffset", zid2.getClass().getSimpleName());

Assert.assertSame(ZoneOffset.UTC, zid2);

ZoneId zid3 = ZoneId.of("UTC");

Assert.assertEquals("ZoneRegion", zid3.getClass().getSimpleName());

}

Для временной зоны UTC заведена специальная константа java.time.ZoneOffset#UTC, но тем не менее запрос на ZoneId.of(«UTC») в новом API выдает уже объект класса java.util.ZoneRegion, а не эту константу.

# Часы

"[Время — это часы](http://phoenix.dubna.ru/el-bib/sazonov/saz-html/saz-2.htm)" — как утверждают некоторые физики. И это фраза является ключевой для нового API, где класс [java.time.Clock](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html) является краеугольным. И также как некоторые из наших часов, время для нас может быть: константным (неидущим), опаздывающим, идущим с различной степенью точности, двигающем стрелки по разному в разных часовых поясах. В общем в новом API можно использовать (либо определить самому) практически любой ход времени, в том числе и для проверки тестов.  
  
Стандартный экземпляр java.time.Clock можно создать только фабричными статическими методами (сам класс абстрактный).  
  
Стандартный экземпляр java.time.Clock всегда знает о временной зоне в которой его создали (хотя это бывает и ненужным).  
  
Пройдемся по фабричным методам:

* java.time.Clock#systemDefaultZone — метод создает системные часы во временной зоне по-умолчанию.
* java.time.Clock#systemUTC — метод создает системные часы во временной зоне UTC.
* java.time.Clock#system — метод создает системные часы в указанной временной зоне.
* java.time.Clock#fixed — метод создает часы константного времени, то есть часы не идут, а стоят на месте.
* java.time.Clock#offset — метод создает прокси над указанными часами, который смещает время на указанную величину.
* java.time.Clock#tickSeconds — метод создает системные часы в указанной временной зоне, значение которых округлено до целых секунд.
* java.time.Clock#tickMinutes — метод создает системные часы в указанной временной зоне, значение которых округлено до целых минут.
* java.time.Clock#tick — метод создает прокси над указанными часами, который округляет значения времени до указанного периода.
* java.time.Clock#withZone — метод создает копию текущих часов в другой временной зоне.

Можно переопределить [java.time.Clock](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html) и написать любую свою логику выдачи времени, например часы которые выдают случайное время на каждый запрос, почему бы и нет?   
  
У объекта java.time.Clock всего три рабочих метода:

* java.time.Clock#getZone — запросить временную зону в которой работают часы.
* java.time.Clock#millis — запросить текущее время в миллисекундах по Unix-time
* java.time.Clock#instant — запросить текущее время в самом общем смысле (по факту — в наносекундах по Unix-time)

Теперь немного критики:

* Я бы завел чистый интерфейс java.time.Clock и отдельно фабрику java.time.Clocks — но я не настаиваю.
* Часам зачем-то в обязательном порядке навязывают временную зону. Самим часам она вообще не нужна: ни java.time.Clock#millis, ни java.time.Clock#instant временную зону не используют. Временная зона часов запрашивается в фабричных методах {Zoned,Local,Offset}DateTime, но именно туда ее и можно было передавать отдельным параметром в методе, а не хранить балластом в java.time.Clock.
* К сожалению класса MockClock для ручного управления временем для тестов нет, придется писать его самим — это не проблема, но было лучше бы если бы он был сразу.
* У часов нет метода java.time.Clock#ticks для измерения вневременных наносекундных тиков (аналог java.lang.System#nanoTime). С одной стороны отсутствие такого метода объяснимо, потому как к исчислению времени не относится. Но с другой стороны, это относится к измерению длительности операций. Поэтому для управлениями вневременными тиками (и измерением длительности соответственно) в тестах было бы неплохо, если бы метод для тиков находился бы также в этом интерфейсе, хотя бы потому что ручное продвижение времени вперед в MockClock по умолчанию продвигало бы одновременно как время, так и тики.

# Instant

[java.time.Instant](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html) — это новый java.util.Date, только неизменяемый, с наносекундной точностью и корректным названием. Внутри хранит [Unix-time](https://ru.wikipedia.org/wiki/UNIX-%D0%B2%D1%80%D0%B5%D0%BC%D1%8F) в виде двух полей: long с количеством секунд, и int с количеством наносекунд внутри текущей секунды.   
  
Значение обоих полей можно запросить напрямую, а также можно попросить посчитать более привычное для старого API представление Unix-time в виде миллисекунд:

@Test

public void testInstantFields() throws Exception {

Instant instant = Clock.systemDefaultZone().instant();

System.out.println(instant.getEpochSecond());

System.out.println(instant.getNano());

System.out.println(instant.toEpochMilli());

}

Также как и java.util.Date (при правильном его использовании), объект класса java.time.Instant ничего не знает про временную зону.  
  
Отдельно стоит сказать про метод java.time.Instant.toString(). Если раньше java.util.Date.toString() работал с учетом текущей локали и временной зоны по умолчанию, то новый java.time.Instant.toString() всегда формирует текстовое представление во временной зоне UTC и одинаковым форматом ISO-8601 — это касается и вывода переменных в IDE при отладке:

@Test

public void testInstantString() throws Exception {

Instant instant1 = Clock.system(ZoneId.of("Europe/Paris")).instant();

System.out.println(instant1.toString());

Instant instant2 = Clock.systemUTC().instant();

System.out.println(instant2.toString());

Instant instant3 = Clock.systemDefaultZone().instant();

System.out.println(instant3.toString());

}

2016-01-06T15:22:53.403Z

2016-01-06T15:22:53.417Z

2016-01-06T15:22:53.423Z

# Базовые интерфейсы

Посмотрим на базовый интерфейс [java.time.temporal.TemporalAccessor](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAccessor.html). Интерфейс TemporalAccessor — это справочник для запроса отдельной частичной информации по текущей точке или метке и его реализуют все временные классы нового API.  
  
Попросим значение Unix-time у java.time.Instant:

@Test(expected = DateTimeException.class)

public void testTemporalAccessor2() throws Exception {

TemporalAccessor ta = Clock.systemUTC().instant();

*// java.time.DateTimeException: Invalid value for InstantSeconds \*

*// (valid values -9223372036854775808 - 9223372036854775807): 1451983908*

System.out.println(ta.get(ChronoField.INSTANT\_SECONDS));

}

Получаем исключение с совершенно необъяснимым сообщением:

java.time.DateTimeException: Invalid value for InstantSeconds \

(valid values -9223372036854775808 - 9223372036854775807): 1451983908

Немного подебажив, становится ясна причина исключения: результат теоретически может не помещаться в диапазон int (хотя в данный момент помещается). Поле INSTANT\_SECONDS надо запрашивать как long. Исправим запрос, попутно запросим дополнительную мета-информацию:

@Test

public void testTemporalAccessor3() throws Exception {

TemporalAccessor ta = Clock.systemUTC().instant();

System.out.println(ta.getLong(ChronoField.INSTANT\_SECONDS));

ValueRange vr = ta.range(ChronoField.INSTANT\_SECONDS);

System.out.println(vr.getMinimum());

System.out.println(vr.getMaximum());

System.out.println(ta.isSupported(ChronoField.INSTANT\_SECONDS));

System.out.println(ta.isSupported(ChronoField.CLOCK\_HOUR\_OF\_DAY));

}

1452094053

-9223372036854775808

9223372036854775807

true

false

Поле CLOCK\_HOUR\_OF\_DAY не поддерживается типом Instant. Это совершенно ожидаемо, поскольку для выяснения часа дня по временной точке нам нужно указать временную зону, которой в java.time.Instant нет. Попробуем все таки запросить это значение:

@Test(expected = UnsupportedTemporalTypeException.class)

public void testTemporalAccessor1() throws Exception {

TemporalAccessor ta = Clock.systemUTC().instant();

*// java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: ClockHourOfDay*

System.out.println(ta.getLong(ChronoField.CLOCK\_HOUR\_OF\_DAY));

}

Все правильно — при запросе часа дня мы получаем исключение. Прекрасно, что метод запроса не стал использовать временную зону по умолчанию (которой в новом API и нет).  
  
Кроме запроса отдельных полей можно запрашивать значения с помощью более сложных алгоритмов-стратегий наследующих интерфейс [java.time.TemporalQuery](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalQuery.html):

@Test

public void testTemporalAccessor4() throws Exception {

TemporalAccessor ta = Clock.systemUTC().instant();

ZoneId zoneId1 = ta.query(TemporalQueries.zone());

ZoneId zoneId2 = TemporalQueries.zone().queryFrom(ta);

Assert.assertEquals(zoneId1, zoneId2);

TemporalUnit unit1 = ta.query(TemporalQueries.precision());

TemporalUnit unit2 = TemporalQueries.precision().queryFrom(ta);

Assert.assertEquals(unit1, unit2);

}

[java.time.temporal.Temporal](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/Temporal.html) — интерфейс является наследником интерфейса TemporalAccessor. Вводит операции сдвига временной точки/метки вперед и назад, операцию замены части временной информации, а также операцию вычисления расстояния до другой временной точки/метки. Реализуется почти всеми «полноценными» временными классами нового API.  
  
Пробуем сдвинуть метку на день вперед и посчитаем разницу:

@Test

public void testTemporal1() throws Exception {

Temporal t1 = Clock.systemUTC().instant();

Temporal t2 = t1.plus(1, ChronoUnit.DAYS);

Assert.assertEquals(Duration.ofDays(1).getSeconds(),

t2.getLong(ChronoField.INSTANT\_SECONDS) - t1.getLong(ChronoField.INSTANT\_SECONDS));

Assert.assertEquals(24, t1.until(t2, ChronoUnit.HOURS));

Assert.assertEquals(24, Duration.between(t1, t2).get(ChronoUnit.HOURS));

}

Поскольку все классы наконец-то стали неизменяемыми, то результаты операций надо не забыть присвоить другой переменной, поскольку оригинальная при операции не изменяется — все аналогично java.lang.String или java.math.BigDecimal.  
  
Попробуем изменить час дня в java.time.Instant:

@Test(expected = UnsupportedTemporalTypeException.class)

public void testTemporal2() throws Exception {

Temporal t = Clock.systemUTC().instant();

*// java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: HourOfDay*

t.with(ChronoField.HOUR\_OF\_DAY, 2);

}

Ожидаемо получаем по рукам, поскольку для этой операции необходима временная зона.  
  
[java.time.temporal.TemporalAdjuster](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjusters.html) — интерфейс стратегии коррекции временной точки/метки, например перемещение в первый день текущего кода. Раньше приходилось для этого писать свои вспомогательные классы для работы с полями java.util.Calendar — сейчас весь код можно оформить в виде стратегии, если нужной еще нет в стандартной поставке:

@Test

public void testTemporalAdjuster() throws Exception {

ZonedDateTime zdt = ZonedDateTime.of(2005, 10, 30, 0, 0, 0, 0, ZoneId.of("Europe/Moscow"));

ZonedDateTime zdt1 = zdt.with(TemporalAdjusters.firstDayOfYear());

ZonedDateTime zdt2 = (ZonedDateTime) TemporalAdjusters.firstDayOfYear().adjustInto(zdt);

Assert.assertEquals(zdt1, zdt2);

Assert.assertEquals(2005, zdt1.get(ChronoField.YEAR));

Assert.assertEquals(1, zdt1.get(ChronoField.MONTH\_OF\_YEAR));

Assert.assertEquals(1, zdt1.get(ChronoField.DAY\_OF\_MONTH));

}

Теперь можно перейти к временным классам.

# LocalTime, LocalDate, LocalDateTime

[java.time.LocalTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) — это кортеж (час, минуты, секунды, наносекунды)  
[java.time.LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) — это кортеж (год, месяц, день месяца)  
[java.time.LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) — оба кортежа вместе  
  
К этим же классам я бы отнес еще и специфические классы для хранения части информации: [java.time.MonthDay](https://docs.oracle.com/javase/8/docs/api/java/time/MonthDay.html), [java.time.Year](https://docs.oracle.com/javase/8/docs/api/java/time/Year.html), [java.time.YearMonth](https://docs.oracle.com/javase/8/docs/api/java/time/YearMonth.html)  
  
Все эти классы объединяет то, что они содержат временные метки или их части, но временные точки на временной оси сами по себе определить не в состоянии (даже LocalDateTime) — поскольку ни в одном из них нет ни временной зоны, ни даже смещения.  
  
Эти классы, как и все другие, поддерживают интерфейс java.lang.Comparable, но нужно понимать, что это именно сравнение временных меток, а не временных точек:

@Test

public void testLocalDateTime() throws Exception {

ZonedDateTime zdt1 = ZonedDateTime.of(2015, 1, 10, 15, 0, 0, 0, ZoneId.of("Europe/Moscow"));

ZonedDateTime zdt2 = ZonedDateTime.of(2015, 1, 10, 14, 0, 0, 0, ZoneId.of("Europe/London"));

Assert.assertEquals(-1, zdt1.compareTo(zdt2));

LocalDateTime ldt1 = zdt1.toLocalDateTime();

LocalDateTime ldt2 = zdt2.toLocalDateTime();

Assert.assertEquals(+1, ldt1.compareTo(ldt2));

}

Нужно сказать, что несмотря на неизбежные параллели в использовании между java.time.LocalTime и java.sql.Time, а также между java.time.LocalDate и java.sql.Date — это совершенно различные классы. В старом API классы java.sql.Time и java.sql.Date являются наследниками java.util.Date, а это значит, что их интерпретация (получение значения часа например) зависит от временной зоны в которой объект этого класса был создан и от временной зоны в которой этот объект будет прочитан. В новом API классы java.time.LocalTime и java.time.LocalDate — это честные кортежи значений и при записи и чтении значения часа временная зона никак не участвует.  
  
Однако временная зона необходима при создании их из временной точки, поскольку интерпретация дней-часов от нее зависит:

@Test(expected = DateTimeException.class)

public void testLocalDateTimeCreate1() throws Exception {

Clock clock = Clock.system(ZoneId.of("Europe/Moscow"));

*// java.time.DateTimeException: Unable to obtain LocalDateTime \*

*// from TemporalAccessor: 2016-01-11T15:15:03.180Z of type java.time.Instant*

LocalDateTime ldt = LocalDateTime.from(clock.instant());

}

Исключение выбрасывается, по причине того, что временную зону взять просто неоткуда (в Instant ее нет, а зону по-умолчанию не берем). Но ее можно получить либо из часов java.time.Clock, либо передать дополнительно:

@Test

public void testLocalDateTimeCreate2() throws Exception {

Clock clock = Clock.system(ZoneId.of("Europe/Moscow"));

LocalDateTime ldt1 = LocalDateTime.ofInstant(clock.instant(), ZoneId.of("UTC"));

System.out.println(ldt1);

LocalDateTime ldt2 = LocalDateTime.now(clock);

System.out.println(ldt2);

}

Теперь все работает, но легкость, с которой можно сделать ошибку, несколько настораживает.   
  
В комментариях к [предыдущей статье](http://habrahabr.ru/post/274811/) упомянули, что настоящие параноики должны еще указывать календарь при операциях с календарными значениями (что включает создание объектов всех временных классов кроме Instant). В новом API есть несколько календарей, которые названы хронологиями:

@Test

public void testChronology() throws Exception {

Clock clock = Clock.system(ZoneId.of("Europe/Moscow"));

ZonedDateTime zdt = ZonedDateTime.now(clock);

ChronoLocalDateTime dt1 = IsoChronology.INSTANCE.localDateTime(zdt);

System.out.println(dt1); *// 2016-01-11T18:48:15.145*

ChronoLocalDateTime dt2 = JapaneseChronology.INSTANCE.localDateTime(zdt);

System.out.println(dt2); *// Japanese Heisei 28-01-11T18:48:15.145*

ChronoLocalDateTime dt3 = ThaiBuddhistChronology.INSTANCE.localDateTime(zdt);

System.out.println(dt3); *// ThaiBuddhist BE 2559-01-11T18:48:15.145*

}

Вообще сложно представить кейс, где может потребоваться отличная от [ISO-8601](https://en.wikipedia.org/wiki/ISO_8601) хронология [IsoChronology](https://docs.oracle.com/javase/8/docs/api/java/time/chrono/IsoChronology.html) (которая практически эквивалентна грегорианскому календарю), но, если что, новый API это поддерживает. 

# ZonedDateTime

[java.time.ZonedDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html) — аналог java.util.Calendar. Это самый мощный класс с полной информацией о временном контексте, включает временную зону, поэтому все операции со сдвигами этот класс проводит правильно.  
  
Попробуем создать ZonedDateTime из LocalDateTime:

@Test(expected = DateTimeException.class)

public void testZoned1() throws Exception {

LocalDateTime ldt = LocalDateTime.of(2015, 1, 10, 0, 0, 0, 0);

*// java.time.DateTimeException: Unable to obtain ZonedDateTime from TemporalAccessor: 2015-01-10T00:00 of type java.time.LocalDateTime*

ZonedDateTime zdt = ZonedDateTime.from(ldt);

}

Сразу же получаем по рукам за то, что в операции (в LocalDateTime) нет временной зоны, а использовать временную зону по-умолчанию новое API опять отказывается (это очень хорошо).   
  
Правильный вариант:

@Test

public void testZoned2() throws Exception {

LocalDateTime ldt = LocalDateTime.of(2015, 1, 10, 0, 0, 0, 0);

ZonedDateTime zdt = ZonedDateTime.of(ldt, ZoneId.of("Europe/Moscow"));

}

Посмотрим, насколько ZonedDateTime строг по отношению к некорректно указанным датам. В java.util.Calendar есть переключатель lenient, который можно настроить как на «строгий», так и на «мягкий» режим. В новом API такого переключателя нет.  
  
29-е февраля не в високосном году не пройдет:

@Test(expected = DateTimeException.class)

public void testLenient2() throws Exception {

*// java.time.DateTimeException: Invalid date 'February 29' as '2005' is not a leap year*

ZonedDateTime.of(2005, 2, 29, 2, 30, 0, 0, ZoneId.of("Europe/Moscow"));

}

60-ю секунду указать нельзя:

@Test(expected = DateTimeException.class)

public void testLenient3() throws Exception {

*// java.time.DateTimeException: Invalid value for SecondOfMinute (valid values 0 - 59): 60*

ZonedDateTime.of(2005, 2, 20, 2, 30, 60, 0, ZoneId.of("Europe/Moscow"));

}

Но указание метки в момент перевода стрелок на летнее время успешно проходит, а результат отличается от ожидаемого. В строгом режиме java.util.Calendar такое не пропускал (см. [предыдущую статью](http://habrahabr.ru/post/274811/)).

@Test

public void testLenient1() throws Exception {

ZonedDateTime zdt = ZonedDateTime.of(2005, 3, 27, 2, 30, 0, 0, ZoneId.of("Europe/Moscow"));

Assert.assertEquals(3, zdt.getLong(ChronoField.HOUR\_OF\_DAY));

Assert.assertEquals(30, zdt.getLong(ChronoField.MINUTE\_OF\_HOUR));

}

Про операции в ZonedDateTime я ничего писать не буду — можно посмотреть документацию.

# OffsetTime, OffsetDateTime

[java.time.OffsetTime](https://docs.oracle.com/javase/8/docs/api/java/time/OffsetTime.html) — это LocalTime + ZoneOffset  
[java.time.OffsetDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/OffsetDateTime.html) — это LocalDateTime + ZoneOffset  
  
Надо сказать, что также как смещение не является временной зоной (временная зона — это историясмещений, плюс еще дополнительная информация), то также OffsetDateTime хранит меньше информации чем ZonedDateTime. OffsetDateTime может полноценно обозначать временную точку на временной оси, но полностью корректные сдвиги производить не в силах, поскольку о будущих и прошлых переводах стрелок этот класс ничего не знает.   
  
Эти классы можно использовать, если по ситуации известно только текущее смещение пользователя (например [через JavaScript](http://www.w3schools.com/jsref/jsref_gettimezoneoffset.asp)). Полностью корректные операции сдвигов они не позволяют сделать, поэтому лучше использовать [ZonedDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html) — если есть способ выяснить полноценную временную зону пользователя. С другой стороны, между двумя экземплярами OffsetDateTime всегда можно успешно и правильно посчитать разницу в секундах.

# Модификации времени

Из всех классов нового API временную точку на временной оси однозначно определяют только три: java.time.Instant, java.time.ZonedDateTime и java.time.OffsetTime.  
  
Операции сдвига и модификации времени в общем случае выполняются корректно только в java.time.ZonedDateTime, поскольку только он один знает про временные зоны.   
  
Выполним пример с расчетом прошедших часов в день перевода стрелок на зимнее время:

@Test

public void testWinterDay() throws Exception {

ZonedDateTime zdt1 = ZonedDateTime.of(2005, 10, 30, 0, 0, 0, 0, ZoneId.of("Europe/Moscow"));

*// case #1 - ok*

ZonedDateTime zdt2 = zdt1.plusDays(1);

Assert.assertEquals(25, Duration.between(zdt1, zdt2).toHours());

*// case #2 - ok*

ZonedDateTime zdt3 = zdt1.plus(1, ChronoUnit.DAYS);

Assert.assertEquals(25, Duration.between(zdt1, zdt3).toHours());

*// case #3 - ok*

OffsetDateTime odt1 = zdt1.toOffsetDateTime();

OffsetDateTime odt2 = zdt2.toOffsetDateTime();

Assert.assertEquals(25, Duration.between(odt1, odt2).toHours());

*// case #4 - ???*

OffsetDateTime odt3 = zdt1.toOffsetDateTime();

OffsetDateTime odt4 = odt3.plus(1, ChronoUnit.DAYS);

Assert.assertEquals(24, Duration.between(odt3, odt4).toHours());

*// case #5 - ok*

Instant instant1 = Instant.from(zdt1);

Instant instant2 = Instant.from(zdt2);

Assert.assertEquals(25, Duration.between(instant1, instant2).toHours());

*// case #6 - ???*

Instant instant3 = Instant.from(zdt1);

Instant instant4 = instant3.plus(1, ChronoUnit.DAYS);

Assert.assertEquals(24, Duration.between(instant3, instant4).toHours());

*// case #7 - ???*

LocalDateTime localDateTime1 = LocalDateTime.from(zdt1);

LocalDateTime localDateTime2 = localDateTime1.plus(1, ChronoUnit.DAYS);

Assert.assertEquals(24, Duration.between(localDateTime1, localDateTime2).toHours());

*// case #8 - ???*

LocalDateTime localDateTime3 = LocalDateTime.from(zdt1);

LocalDateTime localDateTime4 = LocalDateTime.from(zdt2);

Assert.assertEquals(24, Duration.between(localDateTime3, localDateTime4).toHours());

}

Кейсы case#1 и case#2 выполняются на полноценном классе ZonedDateTime и выдают правильный результат, поскольку в этот день стрелки переводили назад в итоге получается 25 часов.  
  
Кейс case#3 показывает, что OffsetDateTime полноценно сохраняет информацию о точке на временной оси, но кейс case#4 показывает, что с потерей временной зоны этот класс производит вычисления уже по другому.   
  
То же с кейсами case#5 и case#6 — несмотря на то, что Instant полноценно определяет точку на временной оси, расчеты он производит без временной зоны.  
  
Кейсы case#7 и case#8 — показывают, что LocalDateTime не может ни полноценно отразить временную точку, ни произвести расчеты без временной зоны.  
  
Я ни в коем случае не хочу сказать, что эти примеры показывают ошибки в новом API (если кто-то так подумал). Все эффекты ожидаемы и объяснимы. Напрягает другое — насколько такое поведение будет осознано армией Java-разработчиков. В старом API такие потенциальные проблемы были невозможны, поскольку всеми расчетами занимался только один класс java.util.Calendar, а единственное, что в нем можно было сделать неправильно — забыть явно указать временную зону.  
  
Возможно стоило запретить большинство операций со временем во всех классах кроме ZonedDateTime, поскольку только он один в курсе переводов стрелок. Возможно стоило запретить расчет Duration с использованием LocalDateTime, поскольку без временной зоны он не определяет временную точку. Я не готов сейчас как-то серьезно дискутировать на тему возможности или невозможности таких решений, но ощущение опасности от нового API у меня есть. 

# Period, Duration

В новом API есть два класса для определения длительности.  
  
[java.time.Period](https://docs.oracle.com/javase/8/docs/api/java/time/Period.html) — описание календарной длительности (периода) в виде кортежа (год, месяц, день).  
  
[java.time.Duration](https://docs.oracle.com/javase/8/docs/api/java/time/Duration.html) — описание точной длительности в виде целого количества секунд и долей текущей секунды в виде наносекунд.   
  
Разницу между двумя можно показать в примере с днем перевода стрелок на зимнее время. Из-за перевода стрелок назад этот календарный день состоит из 25 часов. 

@Test

public void testDuration() throws Exception {

Period period = Period.of(0, 0, 1);

Duration duration = Duration.of(1, ChronoUnit.DAYS);

ZonedDateTime zdt1 = ZonedDateTime.of(2005, 10, 30, 0, 0, 0, 0, ZoneId.of("Europe/Moscow"));

ZonedDateTime ztd2 = zdt1.plus(period);

Assert.assertEquals(ZonedDateTime.of(2005, 10, 31, 0, 0, 0, 0, ZoneId.of("Europe/Moscow")),

ztd2);

ZonedDateTime ztd3 = zdt1.plus(duration);

Assert.assertEquals(ZonedDateTime.of(2005, 10, 30, 23, 0, 0, 0, ZoneId.of("Europe/Moscow")),

ztd3);

}

При добавлении Period.of(0, 0, 1) мы корректно переходим на следующий календарный день. В случае добавления Duration.of(1, ChronoUnit.DAYS) мы фактически добавляем 24 часа и в следующий календарный день не переходим. 

# Форматирование и парсинг

В старом API всегда сюрпризом было то, что java.text.SimpleDateFormat не являлся потоко-безопасным. Интуитивно потоко-безопасность ожидалась, поскольку SimpleDateFormat вроде как не должен хранить какое-либо состояние.  
  
В новом API эта проблема решена.  
  
[java.time.format.DateTimeFormatter](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html) — класс определяет настройки форматирования и парсинга. 

@Test

public void testFormat() throws Exception {

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:dd z", Locale.ENGLISH);

ZonedDateTime zdt1 = ZonedDateTime.of(2005, 10, 30, 0, 0, 0, 0, ZoneId.of("Europe/Moscow"));

String text = zdt1.format(formatter);

System.out.println(text);

TemporalAccessor ta = formatter.parse(text); *// java.time.format.Parsed*

ZonedDateTime zdt2 = ZonedDateTime.from(ta);

Assert.assertEquals(zdt1, zdt2);

}

Если посмотреть в JavaDoc то видно, что в новом API добавили больше опций для форматирования. Также интересно, что парсинг возвращает не конкретный временной класс, а абстрактный java.time.Temporal (java.time.format.Parsed как реализация), а уже из него, как из сумки с запчастями, мы можем собрать объект того класса, который нам нужен.

# Диаграмма классов

Приведу диаграмму классов нового API. Некоторые второстепенные классы не приведены, также как и реализация таких интерфейсов как java.util.Serializable и java.lang.Comparable.

Базовые интерфейсы

Эра

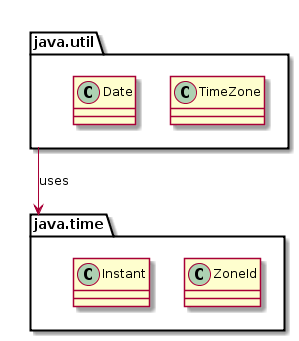
Временная зона

Длительности и периоды

Хронология

Временные классы

# Совместимость

Для обмена информацией между старым и новым API реализовано несколько методов. Причем реализовано достаточно грамотно: старый API знает о новом API, а вот новый API ничего о старом API не знает вообще. Чисто теоретически это позволит когда-нибудь выкинуть все старые классы, но я сомневаюсь, что это случится при нашей жизни.  
  


@Test

public void testTimeZoneCompat() throws Exception {

ZoneId zoneId1 = ZoneId.of("Europe/Moscow");

TimeZone timeZone = TimeZone.getTimeZone(zoneId1);

ZoneId zoneId2 = timeZone.toZoneId();

Assert.assertEquals(zoneId1, zoneId2);

}

@Test

public void testDateCompat() throws Exception {

Instant instant1 = Clock.systemUTC().instant();

Date date = Date.from(instant1);

Instant instant2 = date.toInstant();

Assert.assertEquals(instant1, instant2);

}

И снова есть нюанс: в случае когда мы гоняем время в java.util.Date и обратно у нас безвозвратно теряется точность, поскольку старое API оперирует миллисекундами, а новое оперирует наносекундами. Это не критично, пока у нас есть единственный миллисекундный источник текущего времени в виде java.lang.System#currentTimeMillis, но в будущем это может стать проблемой, особенно для тестов.

# Выводы

У меня осталось смешанное ощущение от нового API. С одной стороны, есть существенные улучшения, с другой - мы получили две главные проблемы: возможность выполнения неожиданно некорректных временных сдвигов при использовании классов отличных от ZonedDateTime, а также возможность неожиданно получить исключение в runtime, при недоступности временной зоны в операциях. Кроме того, новое API несколько сложнее старого. Насколько это будет критично — покажет массовая практика.