ЛАБОРАТОРНАЯ РАБОТА № 1. РАБОТА С ДАТОЙ И ВРЕМЕНЕМ В JAVA

Цель работы:

1. Изучить основные классы пакета java.time для работы с датой и временем
2. Получить практические навыки работы с датой и временем
3. Изучить возможности форматирования даты и времени в Java. Класс DateTimeFormatter

Задание на лабораторную работу

1. Создать новый пакет.
2. Добавить в него новый класс
3. Создать в классе следующие методы:
    a. Метод, возвращающий объект LocalDate из строкового представления даты
    b. Метод, возвращающий объект LocalTime из строкового представления времени
    c. Метод, возвращающий объект LocalDateTime из строкового представления даты и времени
    d. Метод printDate(String date), который может принимать дату в одном из трех форматов: только дата, только время, дата и время. Данный метод должен выводить переданную в него информацию следующим образом:
        i. если исходная строка содержит дату, то вывести по ней: День, День недели, День месяца, Неделя года, Месяц, Год
        ii. если исходная строка содержит время, то вывести: AM или PM, Часы, Часы дня, Минуты, Секунды
        iii. если строка содержит и дату, и время, то вывод должен включать всю перечисленную выше информацию
    e. Метод, определяющий, когда наступит (или уже наступил) ваш день рождения относительно текущей даты (требуется использовать методы сравнения дат isAfter(), isBefore()
    f. Метод, который позволяет выбрать из исходной строки (переданной в качестве входного параметра), все даты, соответствующие выходным дням. В строке даты разделены пробелами
    g. Метод, определяющий, сколько дней осталось до Нового года.
    h. Метод, определяющий, сколько дней осталось до вашего дня рождения
    i. Метод, определяющий день недели, соответствующий переданной в него дате.
    j. Метод, определяющий возраст человека в днях (месяцах). Дата рождений передается в качестве исходного параметра.
    k. Метод, выводящий на экран дни недели, на которые выпадет ваш день рождения в ближайшие 10 лет.
    l. Метод isDateOdd(String date) так, чтобы он возвращал true, если количество дней с начала года - нечетное число, иначе false 2. String date передается в формате FEBRUARY 1 2013 Не забудьте учесть первый день года. Пример: JANUARY 1 2000 = true JANUARY 2 2020 = false

ПРИЛОЖЕНИЕ

Дата и время в Java

Дата и время в Java представлена четырьмя классами. Для того, чтобы начать работу с новыми классами нужно их импортировать:

```
1 import java.time.*;
```

Рассмотрим три класса для работы с датами и временем:

• LocalDate. Позволяет работать только с датами, без времени.

• LocalTime. Позволяет работать только со временем, без дат.

• LocalDateTime. Позволяет работать и с датами, и с временем.

Давайте попробуем получить информацию о текущем времени:

```
1 System.out.println(LocalDate.now());
2 System.out.println(LocalTime.now());
3 System.out.println(LocalDateTime.now());
```

Вывод:

2021-02-15

10:49:45.690

2021-02-15T10:49:45.690

Все три класса имеют статические методы *now()*, которые возвращают соответствующие объекты содержащие текущую дату и время. Первая строчка вывода содержит только дату, вторая только время и третья дату и время (буква T используется для разделения даты и времени). Давайте теперь попробуем создать объект не с текущей датой, а какой-то определенной, например 01.01.2016:

```
1 LocalDate date = LocalDate.of(2016, 1, 1);
2 System.out.println(date);
```

Вывод:

2016-01-01

Метод of можно вызвать по-другому:

```
1 LocalDate date = LocalDate.of(2016, Month.JANUARY, 1);
2 System.out.println(date);
```

Вывод:

2016-01-01

Если обратиться к документации Oracle то можно посмотреть в чем именно заключается отличие в этих двух вызовах:

```
1 public static LocalDate of(int year, int month, int dayOfMonth)
2 public static LocalDate of(int year, Month month, int dayOfMonth)
```

В первом случае для месяца используется переменная с типом *int*, во втором константа перечисляемого типа. Обратите внимание, что нумерация месяцев начинается с единицы, а не нуля. Использования нуля в качестве параметра для месяца сгенерирует исключение класса DateTimeException.

Настало время попробовать создать объект, содержащий только время:

```
1 LocalTime time = LocalTime.of(8, 23);
2 System.out.println(time);
```

Вывод:

08:23

Как и для класса LocalDate метод *of* для класса LocalTime тоже перегружен, например, добавим к нашему времени секунды:

```
1 LocalTime time = LocalTime.of(8, 23, 15);
2 System.out.println(time);
```

Вывод:

08:23:15

Добавим наносекунды:

```
1 LocalTime time = LocalTime.of(8, 23, 15, 150);
2 System.out.println(time);
```

Вывод:

08:23:15.000000150

Напоследок создадим объект класса LocalDateTime, который содержит в себе как дату, так и время:

```
1 LocalDateTime dateTime = LocalDateTime.of(2016, 1, 1, 12, 30, 45);
2 System.out.println(dateTime);
```

Вывод:

2016-01-01T12:30:45

Как можно заметить в методе *of* использовалась комбинация из даты и времени, поэтому выше приведенный код можно написать иначе:

```
1 LocalDate date = LocalDate.of(2016, 1, 1);
2 LocalTime time = LocalTime.of(12, 30, 45);
3 LocalDateTime dateTime = LocalDateTime.of(date, time);
4 System.out.println(dateTime);
```

Вывод:

2016-01-01T12:30:45

Как видите, вывод абсолютно идентичен. Вариаций на тему «как создать объект LocalDateTime» много и описываться в рамках этой статьи они не будут, как правило, это различные комбинации из даты и времени.

нумерация месяцев в java date time api начинается с 1

Учтите, что все три класса LocalDate, LocalTime, LocalDateTime имеют приватные конструкторы и создать экземпляры этих классов напрямую не получится. Следующий код вызовет ошибку компиляции:

```
1 LocalDate date = new LocalDate();
```

Работа с датой и временем.

Работать с классами даты и времени так же просто, как и их создавать, только следует учитывать один нюанс – так же, как и класс String классы даты и времени неизменяемые (Immutable). Рассмотрим пару примеров, создадим объект класса LocalDate:

```
1 LocalDate date = LocalDate.of(2016, Month.MARCH, 1);
2 System.out.println(date);   // 2016-03-01
```

Прибавим к созданной дате один день:

```
1 date = date.plusDays(1);
2 System.out.println(date);   // 2016-03-02
```

Как видите все очень просто, название метода plusDays() говорит само за себя. Для добавления недели можно передать в метод plusDays() целочисленную константу 7, а можно сделать проще:

```
1 date = date.plusWeeks(1);
2 System.out.println(date);   // 2016-03-09
```

метод plusWeeks() удобно использовать, если необходимо добавить несколько недель. Теперь месяцы:

```
1 date = date.plusMonths(1);
2 System.out.println(date);   // 2016-04-09
```

И года:

```
1 date = date.plusYears(1);
2 System.out.println(date);   // 2017-04-09
```

А теперь попробуем вернуться в прошлое:

```
 1  date = date.minusYears(1);
 2  System.out.println(date);   // 2016-04-09
 3
 4  date = date.minusMonths(1);
 5  System.out.println(date);   // 2017-03-09
 6
 7  date = date.minusWeeks(1);
 8  System.out.println(date);   // 2017-03-02
 9
10 date = date.minusDays(1);
11 System.out.println(date);   // 2017-03-01
```

Рассмотрим более интересный пример:

```
1 LocalDate date = LocalDate.of(2016, Month.OCTOBER, 31);
2 date = date.plusDays(1);
3 System.out.println(date);
```

Можно предположить, что в результате выполнения этого кода будет сгенерировано исключение, но это не так. Java прибавит один день к 31 октября и переменная *date* будет содержать дату 1 ноября 2016. Ещё один интересный пример, касающийся дат:

```
1 LocalDate date = LocalDate.of(2015, Month.JANUARY, 29);
2 date = date.plusMonths(1);
3 System.out.println(date);
```

В 2015 году в феврале было 28 дней, но об этом беспокоиться вам не надо, Java сделает все за вас, в результате выполнения этого кода получим 2015-02-28.

Методы для работы с классом LocalTime схожи с теми, которые мы рассматривали выше:

```
 1  LocalTime time = LocalTime.of(1, 15, 30);
 2  System.out.println(time);   // 01:15:30
 3  time = time.plusHours(1);
 4  System.out.println(time);   // 02:15:30
 5  time = time.plusMinutes(10);
 6  System.out.println(time);   // 02:25:30
 7  time = time.plusSeconds(8);
 8  System.out.println(time);   // 02:25:38
 9  time = time.minusSeconds(8);
10 System.out.println(time);   // 02:25:30
11 time = time.minusMinutes(10);
12 System.out.println(time);   // 02:15:30
13 time = time.minusHours(1);
14 System.out.println(time);   // 01:15:30
```

Код прост и интуитивно понятен и подробно рассматриваться не будет, лучше рассмотрим более интересный пример:

```
1 LocalTime time = LocalTime.of(1, 15);
2 System.out.println(time);   // 01:15
3 time = time.plusSeconds(30);
4 System.out.println(time);
```

Можно предположить, что первоначально объект LocalTime содержит в себе только часы и минуты, и при добавлении секунд будет сгенерировано исключение, но нет, в результате выполнения кода получим: 01:15:30

Класс LocalDateTime содержит в себе и дату и время, поэтому к нему можно применить все выше описанные методы классов LocalDate и LocalTime:

```
1 LocalDateTime dateTime = LocalDateTime.of(2016, Month.NOVEMBER, 25, 17, 06);
2 System.out.println(dateTime);   // 2016-11-25T17:06
3 dateTime = dateTime.plusDays(3);
4 System.out.println(dateTime);   // 2016-11-28T17:06
5 dateTime = dateTime.plusMinutes(20);
```

```
6 System.out.println(dateTime);   // 2016-11-28T17:26
```

Приведенный выше код можно записать в три строчки:

```
1 LocalDateTime dateTime = LocalDateTime.of(2016, Month.NOVEMBER, 25, 17, 06);
2 dateTime = dateTime.plusDays(3).plusMinutes(20);
3 System.out.println(dateTime);   // 2016-11-28T17:26
```

Хотелось бы еще раз напомнить о том, что
классы LocalDate, LocalTime и LocalDateTime неизменяемые. Рассмотрим пример:

```
1 LocalDate date = LocalDate.of(2016, Month.NOVEMBER, 25);
2 date.plusDays(3);
3 System.out.println(date);
```

Вывод:
2016-11-25

При добавлении трех дней во второй строчке мы проигнорировали результат и не присвоили его ни одной переменной, как делали до этого.

Работа с временными зонами.

Не смотря, на то, что Oracle настоятельно рекомендует избегать работы с временными зонами (или часовыми поясами) и применять их в случаях крайней нужды, ознакомиться с ними в любом случае нужно. Настало время к рассмотренным ранее трем классам для работы с датой и временем добавить четвертый:
ZonedDateTime – содержит в себе дату, время и временную зону.

Пример:

```
1 System.out.print(ZonedDateTime.now());
```

Вывод:
2016-12-19T13:31:26.103+12:00[Asia/Anadyr]

Как можно заметить часовой пояс был добавлен в конце сообщения *+12:00[Asia/Anadyr]*, иначе говоря, я нахожусь в 12 часовых поясах от Гринвича. Как и с датами объект класса ZonedDateTime можно создать несколькими способами:

```
1 public static LocalDate of(int year, int month, int dayOfMonth, int hour, int minute, int
2 second, int nanos, ZoneId zone)
```

```
3 public static LocalDate of(LocaleDate date, LocaleTime time, ZoneId zone)
  public static LocalDate of(LocaleDateTime dateTime, ZoneId zone)
```

Пример:

```
 1  import java.time.*;
 2
 3  public class Example{
 4      public static void main(String... args){
 5          LocalDate date = LocalDate.of(2016, 12, 30);
 6          LocalTime time = LocalTime.of(12, 30);
 7          ZoneId zone = ZoneId.of("Europe/Moscow");
 8          ZonedDateTime zoned = ZonedDateTime.of(date, time, zone);
 9          System.out.print(zoned);
10      }
11  }
```

Вывод:

2016-12-30T12:30+03:00[Europe/Moscow]

Класс ZonedDateTime учитывает не только временные зоны, но и летнее/зимнее время. В России от переходов на летнее и зимнее время отказались, а вот, к примеру, в США нет. Давайте посмотрим, как это работает. В США переход на зимнее время осуществляется в ноябре в 1:59 отсчет времени начинает вновь с 1:00, иначе говоря от времени отнимается один час:

```
 1  import java.time.*;
 2
 3  public class Example{
 4      public static void main(String... args){
 5          LocalDate date = LocalDate.of(2016, Month.NOVEMBER, 6);
 6          LocalTime time = LocalTime.of(1, 30);
 7          ZoneId zone = ZoneId.of("US/Eastern");
 8          ZonedDateTime zoned = ZonedDateTime.of(date, time, zone);
 9          System.out.println(zoned); // 2016-11-06T01:30-04:00[US/Eastern]
10          zoned = zoned.plusHours(1);
11          System.out.println(zoned); // 2016-11-06T01:30-05:00[US/Eastern]
12          zoned = zoned.plusHours(1);
13          System.out.println(zoned); // 2016-11-06T02:30-05:00[US/Eastern]
14      }
15  }
```

Как видите, при первом добавлении часа, время не меняется. Давайте посмотрим, как дела обстоят с переходом на летнее время. В США переход на летнее время происходит в марте после 1:59 наступает 3:00, другими словами прибавляется один час.

```
 1  import java.time.*;
 2
 3  public class Example{
 4      public static void main(String... args){
 5          LocalDate date = LocalDate.of(2016, Month.MARCH, 13);
 6          LocalTime time = LocalTime.of(1, 30);
 7          ZoneId zone = ZoneId.of("US/Eastern");
 8          ZonedDateTime zoned = ZonedDateTime.of(date, time, zone);
 9          System.out.println(zoned); // 2016-03-13T01:30-05:00[US/Eastern]
10          zoned = zoned.plusHours(1);
11          System.out.println(zoned); // 2016-03-13T03:30-04:00[US/Eastern]
12          zoned = zoned.plusHours(1);
13          System.out.println(zoned); // 2016-03-13T04:30-04:00[US/Eastern]
14      }
15  }
```

Час из жизни американцев был бессовестно украден.

Работа с периодами.

Для работы с временными периодами в Java присутствует замечательный класс Period. Давайте создадим экземпляр класса и более рассмотрим его методы:

```
1 Period period1 = Period.ofDays(1);
2 Period period2 = Period.ofWeeks(1);
3 Period period3 = Period.ofMonths(1);
4 Period period4 = Period.ofYears(1);
5 Period period5 = Period.of(2, 3, 1);
```

Так же, как классы даты и времени, рассмотренные выше, периоды имеют приватные конструкторы, а для создания экземпляров класса используются статичные методы. В примере первой строчкой мы создали период из одного дня, второй строчкой – из одной недели, третьей – из одного месяца, четвертной — из одного года и последней – из двух лет, трех месяцев и одного дня. Период можно создать и по-другому:

```
1 LocalDate dateStart = LocalDate.of(2016, Month.SEPTEMBER, 1);
2 LocalDate dateEnd = LocalDate.of(2016, Month.SEPTEMBER, 2);
3 Period period = Period.between(dateStart, dateEnd);
```

В примере мы создали период из одного дня использовав экземпляры класса LocalDate в качестве начальной и конечной даты нашего периода. При создании методов можно использовать цепочку вызовов (method chaining), но делать этого не стоит и сейчас мы рассмотрим почему:

```
1 Period period = Period.ofDays(1).ofMonths(1);
```

Логично предположить, что в примере был создан период из одного месяца и одного дня, но это не так. При использовании цепочки вызовов для периодов учитывается только последний вызов – в наше случае ofMonths. Иначе говоря выше приведенный код эквивалентен:

```
1 Period period = Period.ofDays(1);
2 period = Period.ofMonths(1);
```

Первой строчкой мы создаем период в один день, а второй заменяем его на период в один месяц.

сцепка методов на периодах не работает, выполняется только последний метод

Рассмотрим теперь работу с периодами с практичной стороны:

```
1 LocalDate date = LocalDate.of(2016, Month.MARCH, 1);
2 Period period = Period.ofDays(10);
3 date = date.plus(period);
4 System.out.println(date); //2016-03-11
```

Первыми двумя строчками мы создаем дату 1-03-2016 и период в 10 дней, третьей строчкой с помощью метода plus мы прибавляем к созданной дате период в 10 дней и получаем 11-03-2016. Абсолютно так же периоды работают и с классом LocalDatetime:

```
1 LocalDateTime dateTime = LocalDateTime.of(2016, Month.MARCH, 1, 12, 30);
2 Period period = Period.ofDays(10);
3 dateTime = dateTime.plus(period);
4 System.out.println(dateTime); //2016-03-11T12:30
```

Разница лишь только в выводе дополнительной информации о времени. При попытке прибавить период в десять дней к объекту класса LocalTime будет сгенерировано исключение:

```
1 LocalTime time = LocalTime.of(12, 30);
2 Period period = Period.ofDays(10);
3 time = time.plus(period);  //UnsupportedTemporalTypeException
```

Исключение генерируется, потому что объект LocalTime не содержит в себе никакой информации о дате, только время.

Работа с промежутками.

Как вы могли заметить периоды, не работают с классами времени и создать период из 10 минут не получится, в таких ситуациях на помощь нам придут промежутки (Duration, если переводить этот термин «в лоб», то дословно получится продолжительность/длительность, но мне больше нравится термин промежуток). Работа с промежутками очень схожа с тем, что мы делали с периодами, за исключением того, что промежутки работают со временем и не работают с датами:

```
1  import java.time.*;
2
3  public class Example{
4      public static void main(String... args){
5          Duration day = Duration.ofDays(1);
6          System.out.println(day); // PT24H
7          Duration hour = Duration.ofHours(1);
8          System.out.println(hour); // PT1H
9          Duration minute = Duration.ofMinutes(30);
10         System.out.println(minute); // PT30M
11         Duration second = Duration.ofSeconds(20);
12         System.out.println(second); // PT20S
13         Duration millis = Duration.ofMillis(20);
14         System.out.println(millis); // PT0.02S
15         Duration nanos = Duration.ofNanos(20);
16         System.out.println(nanos); // PT0.00000002S
17     }
18 }
```

В примере выше, как вы можете заметить, самый большой промежуток — это день, который измеряется в часах. Конечно никто вам не помешает создать промежуток из 168 часов, что будет эквивалентно недели, но лучше всё-таки для этой цели воспользоваться периодами. Так же обратите внимание на то как представлена информация в потоке вывода, если в периодах перед выводом стояла буква *P*, в промежутках указывается *PT* (Period of Time – период времени).

Промежутки можно создать и другим способом:

```
1  import java.time.*;
2  import java.time.temporal.ChronoUnit;
3
4  public class Example{
5      public static void main(String... args){
6          Duration day = Duration.of(1, ChronoUnit.DAYS);
7          System.out.println(day); // PT24H
```

```
8         Duration hour = Duration.of(1, ChronoUnit.HOURS);
9         System.out.println(hour); // PT1H
10        Duration minute = Duration.of(30, ChronoUnit.MINUTES);
11        System.out.println(minute); // PT30M
12        Duration second = Duration.of(20, ChronoUnit.SECONDS);
13        System.out.println(second); // PT20S
14        Duration millis = Duration.of(20, ChronoUnit.MILLIS);
15        System.out.println(millis); // PT0.02S
16        Duration nanos = Duration.of(20, ChronoUnit.NANOS);
17        System.out.println(nanos); // PT0.00000002S
18    }
19 }
```

Результат работы этой программы абсолютно идентичен предыдущему, только в этот раз мы воспользовались классом перечислений ChronoUnit. Этот класс может быть полезен если вы хотите задать какой-нибудь нестандартный промежуток, например:

```
1 Duration halfDay = Duration.of(1, ChronoUnit.HALF_DAYS);
2 System.out.println(halfDay); // PT12H
```

Как уже говорилось работа с промежутками абсолютно идентична работе с периодами:

```
1  import java.time.*;
2
3  public class Example{
4     public static void main(String... args){
5         LocalDate date = LocalDate.of(2016, Month.DECEMBER, 17);
6         LocalTime time = LocalTime.of(12, 30);
7         LocalDateTime dateTime = LocalDateTime.of(date, time);
8         Duration duration = Duration.ofHours(8);
9         System.out.println(dateTime.plus(duration)); // 2016-12-17T20:30
10        System.out.println(time.plus(duration)); // 20:30
11        System.out.println(date.plus(duration)); // UnsupportedTemporalTypeException
12    }
13 }
```

Еще раз повторюсь, промежутки не работают с датами, поэтому в последней строчке было сгенерировано исключение. Подытожим рассмотрение классов периода и промежутка следующей таблицей:

Классы java.time	Применение периода	Применение промежутка
LocalDate	Да	Нет
LocalTime	Нет	Да
LocalDateTime	Да	Да
ZonedDateTime	Да	Да

Работа с моментами.

Да в Java присутствует и такое В оригинале – Instant. Класс Instant представляет собой какой-то конкретный момент во времени и, как правило, используется для запуска какого-либо счетчика:

```
1  import java.time.*;
2  import java.util.concurrent.TimeUnit;
3
4  public class Example{
5      public static void main(String... args) throws InterruptedException{
6          Instant now = Instant.now();
7          TimeUnit.SECONDS.sleep(10);
8          Instant later = Instant.now();
9          Duration duration = Duration.between(now, later);
10         System.out.println(duration); // PT10.013S
11     }
12 }
```

В этом примере используется метод sleep() для остановки основного потока на 10 секунд, что мы и видим создав промежуток между двумя моментами до и после остановки потока. Если у вас есть объект класса ZonedDateTime вы его можете с легкостью конвертировать в Instant:

```
1  import java.time.*;
2
3  public class Example{
4      public static void main(String... args) {
5          LocalDate date = LocalDate.of(2016, Month.MARCH, 16);
6          LocalTime time = LocalTime.of(12, 30);
7          ZoneId zone = ZoneId.of("Europe/Moscow");
8          ZonedDateTime zoned = ZonedDateTime.of(date, time, zone);
9          Instant instant = zoned.toInstant();
10         System.out.println(zoned); // 2016-03-16T12:30+03:00[Europe/Moscow]
11         System.out.println(instant); // 2016-03-16T09:30:00Z
12     }
13 }
```

Последние две строчки кода выводят один и тот же момент во времени, только переменная Instant содержит время в формате GMT. Напоследок хотелось бы отметить одну отличительную особенность класса Instant – он не умеет работать с неделями:

```
import java.time.*;
import java.time.temporal.ChronoUnit;

public class Example{
    public static void main(String... args){
        Instant instant = Instant.now();
        System.out.println(instant); // 2016-12-19T04:39:58.259Z
        Instant day = instant.plus(1, ChronoUnit.DAYS);
        System.out.println(day); // 2016-12-20T04:39:58.259Z
        Instant week = instant.plus(1, ChronoUnit.WEEKS); //
UnsupportedTemporalTypeException
    }
}
```

Форматируемый вывод.

До этого момента для получения информации о дате или времени мы пользовались только стандартным методом *println()* в качестве параметра, которому передавали ссылку на экземпляр класса даты и времени, но классы пакета java.time предоставляют гораздо больше возможностей для получения информации и ее гибкого форматирования для последующего вывода на экран. Рассмотрим следующий пример:

```
LocalDate date = LocalDate.of(2016, Month.MARCH, 1);
System.out.println(date.getDayOfMonth()); //1
System.out.println(date.getDayOfWeek()); //TUESDAY
System.out.println(date.getDayOfYear()); //61
System.out.println(date.getMonthValue()); //3
System.out.println(date.getYear()); //2016
```

Каждый из методов гэттеров возвращает определенную информацию. Первый – день месяца, второй – день недели, третий – количество дней, прошедших сначала года, четвертый – номер месяца и последний – год. Используя эти методы можно сформировать вывод информации о дате в удобном нам формате, но есть способ проще – использование класса DateTimeFormatter. Класс DateTimeFormatter хранится в пакете java.time.format. Рассмотрим подробнее как его использовать:

```
LocalDate date = LocalDate.of(2016, Month.MARCH, 1);
LocalTime time = LocalTime.of(12, 30);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE)); //2016-03-01
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME)); //12:30:00
```

```
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
//2016-03-01T12:30:00
```

В примере выше был использован международный стандарт ISO для отображения дат и времени, для преобразования выводимой информации в этот стандарт был использован метод format() в качестве параметра, которому была передана константа с указанием нужного нам формата. Если стандарт ISO нас не устраивает в Java есть несколько предустановленных форматов, рассмотрим один из них:

```
1 LocalDate date = LocalDate.of(2016, Month.MARCH, 1);
2 LocalTime time = LocalTime.of(12, 30);
3 LocalDateTime dateTime = LocalDateTime.of(date, time);
4 DateTimeFormatter shortDateTime =
5 DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
6 System.out.println(shortDateTime.format(dateTime)); //01.03.16
7 System.out.println(shortDateTime.format(date)); //01.03.16
  System.out.println(shortDateTime.format(time)); //UnsupportedTemporalTypeException
```

Как Вы можете заметить для того, чтобы воспользоваться предустановленным форматом сначала нужно создать экземпляр класса DateTimeFormatter и вызывать метод format() непосредственно этого экземпляра. Последняя строчка кода генерирует исключение, потому что время не может быть отформатирована как дата, а при создании объекта класса DateTimeFormatter мы использовали метод ofLocalizedDate, вместо этого метода можно было бы воспользоваться методом ofLocalizedDateTime и код бы выполнился без каких-либо проблем. Если Вас не устраивает ни формат ISO ни предустановленные форматы Вы можете разработать свой, делается это очень просто:

```
1 LocalDate date = LocalDate.of(2016, Month.MARCH, 1);
2 LocalTime time = LocalTime.of(12, 30);
3 LocalDateTime dateTime = LocalDateTime.of(date, time);
4 DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd MMMM yyyy, hh:mm");
5 System.out.println(dateTime.format(formatter)); //01 марта 2016, 12:30
```

Для того чтобы сделать свой шаблон вывода был использован метод ofPattern(), который в качестве параметра принимает переменную типа String. Остановимся по подробнее на самой строке шаблона:

dd обозначает день месяца, если бы мы воспользовались одной буквой d, то получили бы дату без нуля впереди (1 марта)

MMMM используется для отображения месяца, чем больше букв M используется, тем информативнее будет вывод, к примеру одна буква M отобразит 3, MM – 03, MMM – мар,

MMMM – март

yyyy отображает год, yy – отобразит две последних цифры года (16), yyyy – отобразит год полностью (2016)

Запятая после года, отображается в выводимой информации и использовалась нами для отделения даты и времени

hh отображает часы, если будет одна буква h – часы будут отображаться с одной цифрой без нуля впереди, где это возможно

: использовались нами для отделения часов и минут

mm – отображают минуты.

Рассмотренный шаблон можно так же применить для преобразования строк в дату и время. Рассмотрим пример:

```
1 DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd MM yyyy");
2 LocalDate date = LocalDate.parse("12 02 2016", formatter);
3 LocalTime time = LocalTime.parse("22:12");
4 LocalDateTime dateTime = LocalDateTime.of(date, time);
5 System.out.println(dateTime); //2016-02-12T22:12
```

В примере мы использовали два разных подхода для преобразования строк в дату и время: дату мы преобразовали с использованием нами разработанного шаблона, а вот время без. Как видите оба способа отработали корректно, но будьте готовы к тому что может быть сгенерировано исключение если в качестве строковых параметров передать некорректные значения:

```
1 DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd MM yyyy");
2 LocalDate date = LocalDate.parse("60 02 2016", formatter);  //DateTimeParseException
```

Источник: https://javanerd.ru/%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-java/%D0%B4%D0%B0%D1%82%D0%B0-%D0%B8-%D0%B2%D1%80%D0%B5%D0%BC%D1%8F-%D0%B2-java-8/