10-15 [Java - LocalDate]

소유자	종수 김
태그	

날짜와 시간

날짜와 시간 라이브러리가 필요한 이유

날짜와 시간을 계산하는 것이 복잡하고 어려움.

1. 날짜와 시간 차이 계산
    a. 특정 날짜에서 다른 날짜 까지의 정확한 일수를 계산하는 것은 생각보다 복잡한데, 윤년, 각 달의 일 수 처럼 고려할 사항이 들어가기 때문.
2. 윤년 계산
3. 일광 절약 시간(Daylight Saving Time, DST = 써머타임) 변환
    a. 태양이 일찍 뜨는 3 ~ 10월에 맞추어 1시간 앞당기거나 뒤로 미루는 제도
    b. 써머타임은 각 국가나 지역에 따라 적용 여부와 시작 및 종료 날짜가 다름.
4. 타임존 계산
    a. 세계는 다양한 타임존으로 나뉘어 있으며, 각 타임존은 UTC(협정 세계시)로부터의 시간 차이로 정의
    b. London / UTC / GMT는 세계 시간의 기준이 되는 00:00
    c. Asia/Seoul의 타임존은 UTC + 9

위와 같은 복잡성으로 인해, 잘 설계된 날짜 및 시간 관련 라이브러리를 잘 사용하는 것이 중요.

자바 날짜와 시간 라이브러리

Class or Enum	Year	Month	Day	Hours	Minutes	Seconds*	Zone Offset	Zone ID	toString Output
LocalDate	V	V	V						2013-08-20
LocalTime				V	V	V			08:16:26.943
LocalDateTime	V	V	V	V	V	V			2013-08-20T08:16:26.937
ZonedDateTime	V	V	V	V	V	V	V	V	2013-08-21T00:16:26.941+09:00[Asia/Tokyo]
OffsetDateTime	V	V	V	V	V	V	V		2013-08-20T08:16:26.954-07:00
OffsetTime				V	V	V	V		08:16:26.957-07:00
Year	V								2013
Month		V							AUGUST
YearMonth	V	V							2013-08
MonthDay		V	V						--08-20
Instant						V			2013-08-20T15:16:26.355Z
Period	V	V	V			***	***		P10D (10 days)
Duration			**	**	**	V			PT20H (20 hours)

- 원문: https://docs.oracle.com/javase/tutorial/datetime/iso/overview.html
- *: 초는 나노초 단위의 정밀도로 캡처된다. (밀리초, 나노초 가능)
- **: 이 클래스는 이 정보를 저장하지는 않지만 이러한 단위로 시간을 제공하는 메서드가 있다.
- ***: `ZonedDateTime`에 `Period`를 추가하면 서머타임 또는 기타 현지 시간 차이를 준수한다.

LocalDate, LocalTime, LocalDateTime

LocalDate : 날짜만 표현 - 년,월,일

LocalTime : 시간만 표현 - 시,분,초

LocalDateTime : LocalDate + LocalTime

- Local : 세계 시간대를 고려하지 않기에 타임존이 적용되지 않음.
  특정 지역의 날짜와 시간만 고려할 때 사용.
  Ex) 국내 서비스만을 고려할 때

ZonedDateTime, OffsetDateTime

ZonedDateTime : 시간대를 고려한 날짜와 시간을 표현할 때 사용. 여기에는 시간대를 표현하는 타임존이 포함.

OffsetDateTime : 시간대를 고려한 날짜와 시간을 표현할 때 사용. 타임존은 없고 UTC로부터의 시간대 차이인 고정된 오프셋만을 포함.

- UTC를 기준으로 몇 시간 차이가 나는지를 고려하여 표현.

Instant

UTC를 기준으로 하는, 시간의 한 지점.
Instant는 날짜와 시간을 나노초 정밀도로 표현하며 1970년 1월 1일 0시 0분 0초(UTC)를 기준으로 경과한 시간으로 계산.

- 초 데이터만 들어있음. 저 시간을 기준으로 몇 초가 흘렀는지만 체크가 가능.

Period, Duration

시간의 개념은 두 가지로 표현할 수 있음.

1. 특정 시점의 시간(시각)
    a. 특정 어떤 시점을 얘기할 때 사용하는 시간
2. 시간의 간격(기간)
    a. 사건의 시작 ~ 사건의 종료 까지의 간격

Period : 두 날짜 사이의 간격을 년,월,일로 표현

Duration : 두 시간 사이의 간격을 시,분,초로 표현

기본 날짜와 시간 - LocalDateTime

모든 날짜 클래스는 '불변'.
변경이 발생하는 경우 새로운 객체를 생성해서 반환하므로 반환 값이 필수.

```
package time;

import java.time.LocalDate;

public class LocalDateMain {
    public static void main(String[] args) {
        LocalDate nowDate = LocalDate.now();
        LocalDate ofDate = LocalDate.of(1996, 9, 19);
        System.out.println("nowDate = " + nowDate);
        System.out.println("ofDate = " + ofDate);

        //계산(불변)
        ofDate = ofDate.plusDays(10);
        System.out.println("ofDate = " + ofDate);
    }
}
```

```
package time;

import java.time.LocalTime;

public class LocalTimeMain {
    public static void main(String[] args) {
        LocalTime nowTime = LocalTime.now();
        LocalTime localTime = LocalTime.of(16, 10, 10);
        System.out.println("nowTime = " + nowTime);
        System.out.println("localTime = " + localTime);

        localTime = localTime.plusHours(2);
        System.out.println("localTime = " + localTime);
    }
}

package time;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

public class LocalDateTimeMain {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        System.out.println("now = " + now);
        LocalDateTime localDateTime = LocalDateTime.of(LocalD
        System.out.println("localDateTime = " + localDateTime

        //날짜와 시간 분리
        LocalDate localDate = localDateTime.toLocalDate();
        LocalTime localTime = localDateTime.toLocalTime();
        System.out.println("localDate = " + localDate);
        System.out.println("localTime = " + localTime);

        LocalDateTime localDateTime1 = localDateTime.plusWeek
        System.out.println("localDateTime1 = " + localDateTim
```

```
    }
}
```

- LocalDateTime 객체는 내부에 LocalDate, LocalTime을 가지고 있음.

비교

- **isBefore()**: 다른 날짜시간과 비교한다. 현재 날짜와 시간이 이전이라면 `true`를 반환한다.
- **isAfter()**: 다른 날짜시간과 비교한다. 현재 날짜와 시간이 이후라면 `true`를 반환한다.
- **isEqual()**: 다른 날짜시간과 시간적으로 동일한지 비교한다. 시간이 같으면 `true`를 반환한다.

isEqual() vs equals()

- `isEqual()`는 단순히 비교 대상이 시간적으로 같으면 `true`를 반환한다. 객체가 다르고, 타임존이 달라도 시간적으로 같으면 `true`를 반환한다. 쉽게 이야기해서 시간을 계산해서 시간으로만 둘을 비교한다.
  - 예) 서울의 9시와 UTC의 0시는 시간적으로 같다. 이 둘을 비교하면 `true`를 반환한다.
- `equals()` 객체의 타입, 타임존 등등 내부 데이터의 모든 구성요소가 같아야 `true`를 반환한다.
  - 예) 서울의 9시와 UTC의 0시는 시간적으로 같다. 이 둘을 비교하면 타임존의 데이터가 다르기 때문에 `false`를 반환한다.

타임존 - ZonedDateTime

Asia/Seoul 같은 타임존 안에는 써머타임에 대한 정보와, UTC+9:00와 같은 UTC로 부터 시간 차이인 오프셋 정보를 모두 포함하고 있다.

LocalDateTime + ZoneId

```
package time;

import java.time.ZoneId;
import java.util.Set;

public class ZoneIdMain {
    public static void main(String[] args) {
        //사용 가능한 타임존 목록
        Set<String> availableZoneIds = ZoneId.getAvailableZon
        for(String availableZoneId : availableZoneIds) {
            System.out.println("availableZoneId = " + availab
            ZoneId zoneId = ZoneId.of(availableZoneId);
            System.out.println(zoneId + " | " + zoneId.getRul
        }
```

```
        ZoneId zoneId = ZoneId.systemDefault();
        System.out.println("zoneId = " + zoneId);

        ZoneId seoulZoneId = ZoneId.of("Asia/Seoul");
        System.out.println("seoulZoneId = " + seoulZoneId);
    }
}


package time;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class ZoneDateTimeMain {
    public static void main(String[] args) {
        ZonedDateTime nowZdt = ZonedDateTime.now();
        System.out.println("nowZdt = " + nowZdt);

        LocalDateTime ldt = LocalDateTime.of(2030, 1, 1, 13,
        ZonedDateTime zdt1 = ZonedDateTime.of(ldt, ZoneId.of(
        System.out.println("zdt1 = " + zdt1);

        ZonedDateTime zonedDateTime = ZonedDateTime.of(2030,
        System.out.println("zonedDateTime = " + zonedDateTime

        ZonedDateTime utc = zonedDateTime.withZoneSameInstant
        System.out.println("utc = " + utc);


    }
}
```

OffsetDateTime

LocalDateTime에 UTC 오프셋 정보인 ZoneOffset이 합쳐진 것.

```
package time;

import java.time.LocalDateTime;
import java.time.OffsetDateTime;
import java.time.ZoneOffset;

public class OffsetDateTimeMain {
    public static void main(String[] args) {
        OffsetDateTime nowOdt = OffsetDateTime.now();
        System.out.println("nowOdt = " + nowOdt);

        LocalDateTime ldt = LocalDateTime.of(2030, 1, 1, 13,
        System.out.println("ldt = " + ldt);
        OffsetDateTime odt = OffsetDateTime.of(ldt, ZoneOffse
        System.out.println("odt = " + odt);
    }
}
```

ZonedDateTime vs OffsetDateTime

- `ZonedDateTime`은 구체적인 지역 시간대를 다룰 때 사용하며, 일광 절약 시간을 자동으로 처리할 수 있다. 사용자 지정 시간대에 따른 시간 계산이 필요할 때 적합하다.
- `OffsetDateTime`은 UTC와의 시간 차이만을 나타낼 때 사용하며, 지역 시간대의 복잡성을 고려하지 않는다. 시간대 변환 없이 로그를 기록하고, 데이터를 저장하고 처리할 때 적합하다.

참고

`ZonedDateTime`이나 `OffsetDateTime`은 글로벌 서비스를 하지 않으면 잘 사용하지 않는다. 따라서 너무 깊이있게 파기 보다는 대략 이런 것이 있다 정도로만 알아두면 된다. 실무에서 개발하면서 글로벌 서비스를 개발할 기회가 있다면 그때 필요한 부분을 찾아서 깊이있게 학습하면 된다.

기계 중심의 시간 - Instant

UTC를 기준으로 하는, 시간의 한 지점.
날짜와 시간을 나노초 정밀도로 표현하며, 1970년 1월 1일 0시 0분 0초(UTC)를 기준으로 경과한 시간을 계산

```
package time;
```

```
import java.time.Instant;
import java.time.ZonedDateTime;

public class InstantMain {
    public static void main(String[] args) {
        //생성
        Instant now = Instant.now();
        System.out.println("now = " + now);

        ZonedDateTime zdt = ZonedDateTime.now();
        Instant from = Instant.from(zdt);
        System.out.println("from = " + from);

        Instant epochSecond = Instant.ofEpochSecond(100);
        System.out.println("epochSecond = " + epochSecond);

        Instant later = epochSecond.plusSeconds(3600);
        System.out.println("later = " + later);

        long laterEpochSecond = later.getEpochSecond();
        System.out.println("laterEpochSecond = " + laterEpoch
    }
}
```

참고 - Epoch 시간

Epoch time(에포크 시간), 또는 Unix timestamp는 컴퓨터 시스템에서 시간을 나타내는 방법 중 하나이다. 이는 1970년 1월 1일 00:00:00 UTC부터 현재까지 경과된 시간을 초 단위로 표현한 것이다. 즉, Unix 시간은 1970년 1월 1일 이후로 경과한 전체 초의 수로, 시간대에 영향을 받지 않는 절대적인 시간 표현 방식이다.

참고로 Epoch라는 뜻은 어떤 중요한 사건이 발생한 시점을 기준으로 삼는 시작점을 뜻하는 용어다.

`Instant`는 바로 이 Epoch 시간을 다루는 클래스이다.

Instant의 특징

장점

- 시간대 독립성 : UTC를 기준으로 하므로, 시간대에 영향을 받지 않음. 전 세계 어디서나 동일한 시점

고정된 기준점 : 1970년 1월 1일을 기준으로 하기 때문에 시간 계산 및 비교가 명확하고 일관됨.

단점

사용자 친화적이지 않음 : 기계적인 시간 처리에는 적합하나 사람이 읽고 이해하기는 직관적이지 않음.

시간대 정보 부재 : 시간대 정보가 포함되어 있지 않아, 특정 지역의 날짜와 시간으로 변환하려면 추가적인 작업 필요

사용 예

전 세계적인 시간 기준 필요시 : 전 세계적 일관된 시점을 표현할 때 사용. 로그 기록, 트랜잭션 타임스탬프, 서버 간의 시간 동기화 등.

시간대 변환 없이 시간 계산 필요 시 : 시간대의 변화 없이 순수하게 시간의 흐름(예 : 지속 시간 계산)만을 다루고 싶을 때 적합. 시간대 변환의 복잡성 없이 시간 계산 가능

데이터 저장 및 교환 : 데이터베이스에 날짜와 시간 정보를 저장하거나, 다른 시스템과 날짜와 시간 정보를 교환할 때 사용하면 모든 시스템에서 동일한 기준점을 사용하게 되므로 데이터 일관성 유지가 가능.

기간, 시간의 간격 - Duration, Period

시간의 개념은 두 가지로 표현할 수 있음.
두 날짜 사이의 기간을 나타내기 위한 객체

1. 특정 시점의 시간(시각)
    a. 특정 어떤 시점을 얘기할 때 사용하는 시간
2. 시간의 간격(기간)
    a. 사건의 시작 ~ 사건의 종료 까지의 간격

Period : 두 날짜 사이의 간격을 년,월,일로 표현

Duration : 두 시간 사이의 간격을 시,분,초로 표현

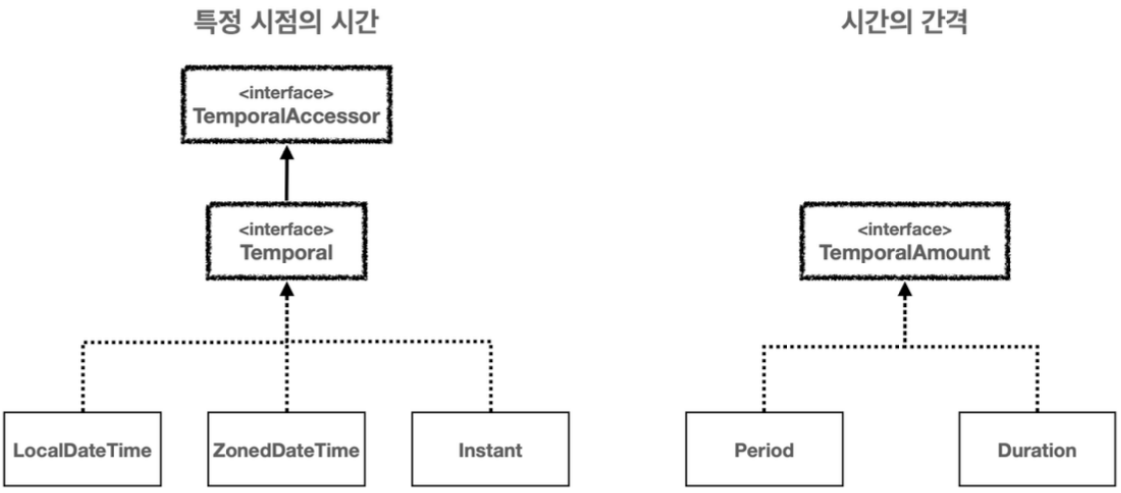
```
package time;

import java.time.Duration;
import java.time.LocalTime;

public class DurationMain {
    public static void main(String[] args) {
        //생성
```

```
        Duration duration = Duration.ofMinutes(30);
        System.out.println("duration = " + duration);

        LocalTime lt = LocalTime.of(1, 0);
        System.out.println("lt = " + lt);

        //계산에 사용
        LocalTime plusTime = lt.plus(duration);
        System.out.println("plusTime = " + plusTime);

        LocalTime startTime = LocalTime.of(9, 0);
        LocalTime endTime = LocalTime.of(10, 0);
        Duration between = Duration.between(startTime, endTim
        System.out.println(between.toMinutes());

    }
}
```

구분	Period	Duration
단위	년, 월, 일	시간, 분, 초, 나노초
사용 대상	날짜	시간
주요 메소드	getYears(), getMonths(), getDays()	toHours(), toMinutes(), getSeconds(), getNano()

- get : 보통 내가 가지고 있는 속성을 바로 반환
- to : 특정 값을 계산해서 반환

날짜와 시간의 핵심 인터페이스



- 특정 시점의 시간: `Temporal`(`TemporalAccessor` 포함) 인터페이스를 구현한다.
  - 구현으로 `LocalDateTime`, `LocalDate`, `LocalTime`, `ZonedDateTime`, `OffsetDateTime`, `Instant` 등이 있다.
- 시간의 간격(기간): `TemporalAmount` 인터페이스를 구현한다.
  - 구현으로 `Period`, `Duration`이 있다.

TemporalAccessor 인터페이스

1. 날짜와 시간을 읽기 위한 기본 인터페이스
2. 이 인터페이스는 특정 시점의 날짜와 시간 정보를 읽을 수 있는 최소한의 기능을 제공
3. 읽기전용 접근

Temporal 인터페이스

1. TemporalAccessor의 하위 인터페이스로 날짜와 시간을 조작(추가, 빼기)하기 위한 기능을 제공
2. 읽기와 쓰기(조작) 모두를 지원

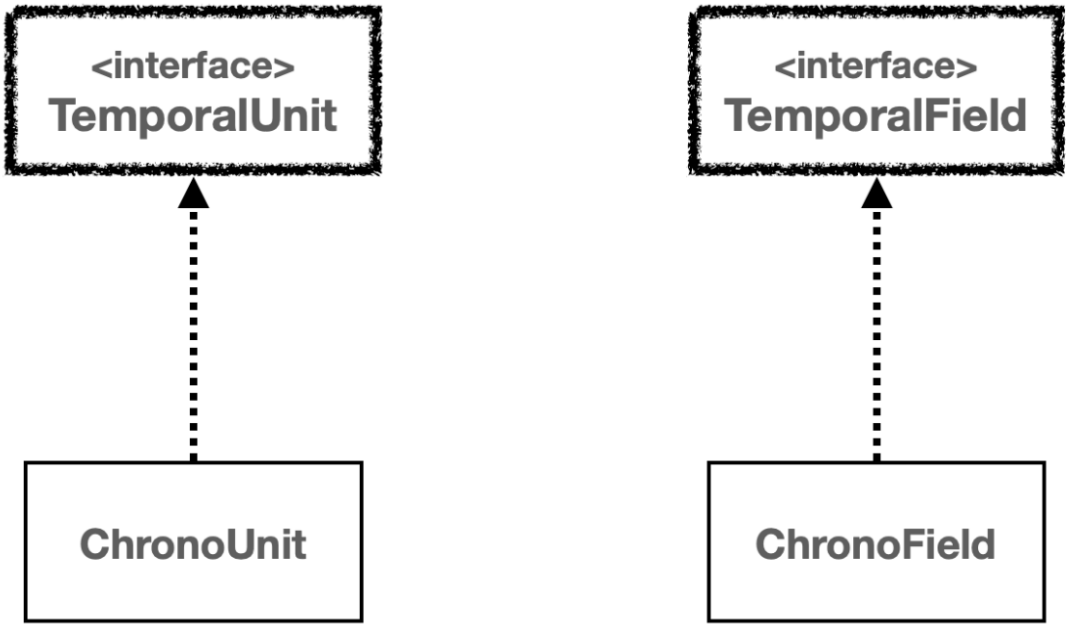
TemporalAmount 인터페이스

1. 시간의 간격(시간의 양, 기간)을 나타내며 날짜와 시간 객체에 적용하여 그 객체를 조정. 특정 날짜에 일정 기간을 더하거나 빼는 것이 가능.

시간의 단위와 시간 필드

```
//시간의 단위 ChronoUnit
package time;
```

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

public class ChronoUnitMain {
    public static void main(String[] args) {
        ChronoUnit[] values = ChronoUnit.values();
        for (ChronoUnit value : values) {
            System.out.println("value = " + value);
        }

        System.out.println("ChronoUnit.HOURS = " + ChronoUnit
        System.out.println("ChronoUnit.HOURS.getDuration().ge
        System.out.println("ChronoUnit.DAYS = " + ChronoUnit.
        System.out.println("ChronoUnit.DAYS.getDuration().get

        //차이 구하기
        LocalTime lt1 = LocalTime.of(1, 10, 0);
        LocalTime lt2 = LocalTime.of(1, 20, 0);

        long secondsBetween = ChronoUnit.SECONDS.between(lt1,
        System.out.println("secondsBetween = " + secondsBetwe

        System.out.println("ChronoUnit.MINUTES.between(lt1, l
    }
}
```



시간의 단위 - TemporalUnit, ChronoUnit

- `TemporalUnit` 인터페이스는 날짜와 시간을 측정하는 단위를 나타내며, 주로 사용되는 구현체는 `java.time.temporal.ChronoUnit` 열거형으로 구현되어 있다.
- `ChronoUnit`은 다양한 시간 단위를 제공한다.
- 여기서 `Unit`이라는 뜻을 번역하면 단위이다. 따라서 시간의 단위 하나하나를 나타낸다.

```
//시간 필드 ChronoField
package time;

import java.time.temporal.ChronoField;

public class ChronoFieldMain {
    public static void main(String[] args) {
        ChronoField[] values = ChronoField.values();
        for (ChronoField field : values) {
            System.out.println("field = " + field + "range =
        }

        System.out.println("ChronoField.MONTH_OF_YEAR.range()
        System.out.println("ChronoField.DAY_OF_MONTH.range() =
```

```
    }
}
```

시간 필드 - ChronoField

`ChronoField`는 날짜 및 시간을 나타내는 데 사용되는 열거형이다. 이 열거형은 다양한 필드를 통해 **날짜와 시간의 특정 부분을 나타낸다.** 여기에는 연도, 월, 일, 시간, 분 등이 포함된다.

- `TemporalField` 인터페이스는 날짜와 시간을 나타내는데 사용된다. 주로 사용되는 구현체는 `java.time.temporal.ChronoField` 열거형으로 구현되어 있다.
- `ChronoField`는 다양한 필드를 통해 날짜와 시간의 특정 부분을 나타낸다. 여기에는 연도, 월, 일, 시간, 분 등이 포함된다.
- 여기서 필드(Field)라는 뜻이 날짜와 시간 중에 있는 특정 필드들을 뜻한다. 각각의 필드 항목은 다음을 참고하자.
  - 예를 들어 2024년 8월 16일이라고 하면 각각의 필드는 다음과 같다.
    - `YEAR`: 2024
    - `MONTH_OF_YEAR`: 8
    - `DAY_OF_MONTH`: 16
- 단순히 시간의 단위 하나하나를 뜻하는 `ChronoUnit`과는 다른 것을 알 수 있다. `ChronoField`를 사용해야 날짜와 시간의 각 필드 중에 원하는 데이터를 조회할 수 있다.

TemporalUnit(Chrono Unit), TemporalField(ChronoField)는 단독으로 사용하기 보다, 주로 날짜와 시간을 조회하거나 조작할 때 사용.

날짜와 시간 조회하고 조작하기1

날짜와 시간 조회하기

```
package time;

import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.temporal.ChronoField;

public class GetTimeMain {
    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2030, 1, 1, 13, 3
        System.out.println("YEAR = " + dt.get(ChronoField.YEA
        System.out.println("MONTH_OF_YEAR = " + dt.get(Chrono
        System.out.println("DAY_OF_MONTH = " + dt.get(ChronoF
```

```
        System.out.println("HOUR_OF_DAY = " + dt.get(ChronoFi
        System.out.println("MINUTE_OF_HOUR = " + dt.get(Chron
        System.out.println("SECOND_OF_MINUTE = " + dt.get(Chr

        // 편의 메서드 제공
        System.out.println("YEAR = " + dt.getYear());
        System.out.println("MONTH_OF_YEAR = " + dt.getMonthVa
        System.out.println("DAY_OF_MONTH = " + dt.getDayOfMon
        System.out.println("HOUR_OF_DAY = " + dt.getHour());
        System.out.println("MINUTE_OF_HOUR = " + dt.getMinute
        System.out.println("SECOND_OF_MINUTE = " + dt.getSeco

        // 편의 메서드에 없는 경우, ChronoField 열거형을 통해 구할 수
        System.out.println("MINUTE_OF_DAY = " + dt.get(Chrono
        System.out.println("SECOND_OF_DAY = " + dt.get(Chrono

    }
}
```

TemporalAccessor.get(TemporalField field)

- `LocalDateTime`을 포함한 특정 시점의 시간을 제공하는 클래스는 모두 `TemporalAccessor` 인터페이스를 구현한다.
- `TemporalAccessor`는 특정 시점의 시간을 조회하는 기능을 제공한다.
- `get(TemporalField field)`을 호출할 때 어떤 날짜와 시간 필드를 조회할 지 `TemporalField`의 구현인 `ChronoField`를 인수로 전달하면 된다.

날짜와 시간 조작하기

```
package time;

import java.time.LocalDateTime;
import java.time.Period;
import java.time.temporal.ChronoUnit;

public class ChangeTimePlusMain {
    public static void main(String[] args) {
```

```
        LocalDateTime dt = LocalDateTime.of(2018, 1, 1, 13, 3
        System.out.println("dt = " + dt);

        LocalDateTime plusDt1 = dt.plus(10, ChronoUnit.YEARS)
        System.out.println("plusDt1 = " + plusDt1);

        LocalDateTime plusDt2 = dt.plusYears(10);
        System.out.println("plusDt2 = " + plusDt2);

        Period period = Period.ofYears(10);
        LocalDateTime plusDt3 = dt.plus(period);
        System.out.println("plusDt3 = " + plusDt3);
    }
}
```

Temporal plus(long amountToAdd, TemporalUnit unit)

- `LocalDateTime`을 포함한 특정 시점의 시간을 제공하는 클래스는 모두 `Temporal` 인터페이스를 구현한다.
- `Temporal`은 특정 시점의 시간을 조작하는 기능을 제공한다.
- `plus(long amountToAdd, TemporalUnit unit)`를 호출할 때 더하기 할 숫자와 시간의 단위(Unit)를 전달하면 된다. 이때 `TemporalUnit`의 구현인 `ChronoUnit`을 인수로 전달하면 된다.
- 불변이므로 반환 값을 받아야 한다.
- 참고로 `minus()`도 존재한다.

정리

시간을 조회하고 조작하는 부분을 보면 TemporalAccessor.get(), Temporal.plus()와 같은 인터페이스를 통해 특정 구현 클래스와 무관하게 아주 일관성 있는 시간 조회, 조작이 가능.

때문에, LocalTime, LocalDate, LocalDateTime 등 인터페이스를 구현하여 일관성 있는 조작이 가능.

한계

```
package time;

import java.time.LocalDate;
import java.time.temporal.ChronoField;
```

```
public class InSupportedMain1 {
    public static void main(String[] args) {
        LocalDate now = LocalDate.now();
        int minute = now.get(ChronoField.SECOND_OF_MINUTE);
        System.out.println("minute = " + minute);
    }
}
```

LocalDate는 날짜만 가지고 있으므로 시간에 관련된 내용은 없기에 해당 코드를 실행하면 UnSpportedException이 터짐.

이를 예방하기 위해, boolean isSupported(Temporal field)와 같은 메서드를 제공

이처럼 구현체에 상관없이 스펙이 잘 정의되어 있는 경우 설계가 잘된 것.

날짜와 시간 조회하고 조작하기2

날짜와 시간을 조작하는 with() 메서드

```
package time;

import java.time.DayOfWeek;
import java.time.LocalDateTime;
import java.time.temporal.ChronoField;
import java.time.temporal.TemporalAdjuster;
import java.time.temporal.TemporalAdjusters;

public class ChangeTimeWithMain {
    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2018, 1, 1, 13, 3
        System.out.println("dt = " + dt);

        LocalDateTime changedDt1 = dt.with(ChronoField.YEAR,
        System.out.println("changedDt1 = " + changedDt1);

        LocalDateTime changedDt2 = dt.withYear(2020);
        System.out.println("changedDt2 = " + changedDt2);

        // TemporalAdjuster 사용
        // 다음주 금요일과 같은 복잡한 날짜 변경
```

```
        LocalDateTime with1 = dt.with(TemporalAdjusters.next(
        System.out.println("기준 날짜 : " + dt);
        System.out.println("다음 주 금요일 : " + with1);

        //이번 달의 마지막 일요일
        LocalDateTime with2 = dt.with(TemporalAdjusters.lastI
        System.out.println("같은 달의 마지막 일요일 : " + with2);

    }
}
```

Temporal with(TemporalField field, long newValue)

- `Temporal.with()` 를 사용하면 날짜와 시간의 특정 필드의 값만 변경할 수 있다.
- 불변이므로 반환 값을 받아야 한다.

편의 메서드

- 자주 사용하는 메서드는 편의 메서드가 제공된다.
- `dt.with(ChronoField.YEAR, 2020)` ➜ `dt.withYear(2020)`

TemporalAdjuster 사용

- `with()` 는 아주 단순한 날짜만 변경할 수 있다. 다음 금요일, 이번 달의 마지막 일요일 같은 복잡한 날짜를 계산하고 싶다면 `TemporalAdjuster` 를 사용하면 된다.

TemporalAdjuster 인터페이스

```
public interface TemporalAdjuster {
    Temporal adjustInto(Temporal temporal);
}
```

원래대로 하면 이 인터페이스를 직접 구현해야겠지만, 자바는 이미 필요한 구현체들을 `TemporalAdjusters` 에 다 만들어두었다. 우리는 단순히 구현체들을 모아둔 `TemporalAdjusters` 를 사용하면 된다.

- `TemporalAdjusters.next(DayOfWeek.FRIDAY)` : 다음 금요일을 구한다.
- `TemporalAdjusters.lastInMonth(DayOfWeek.SUNDAY)` : 이번 달의 마지막 일요일을 구한다.

필요하면 절대 구현하지 말고(어렵고, 시간이 아까우며, 잘 구현된 라이브러리가 있음) TemporalAdjuster 쓰자.

날짜와 시간 문자열 파싱과 포맷팅

포맷팅 - 날짜와 시간 데이터를 원하는 포맷의 문자열로 변경 Date to String

파싱 : 문자열을 날짜와 시간 데이터로 변경하는 것 String to Date

```
package time;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class FormattingMain2 {
    public static void main(String[] args) {
        // 포맷팅 : 날짜와 시간을 문자로
        // 포맷팅 : Date to String
        LocalDateTime now = LocalDateTime.of(2024, 12, 31, 13
        DateTimeFormatter formatter = DateTimeFormatter.ofPat
        String formattedDate = now.format(formatter);
        System.out.println(now);
        System.out.println("날짜와 시간 포맷팅 : " + formattedDat

        // 파싱 : 문자를 날짜와 시간으로
        // 파싱 : String to Date
        String dateTimeString = "2030-01-01 11:30:00";
        LocalDateTime parsedDateTime = LocalDateTime.parse(da
        System.out.println("문자열 파싱 날짜와 시간 : " + parsedDa

    }
}
```

`LocalDate`과 같은 날짜 객체를 원하는 형태의 문자로 변경하려면 `DateTimeFormatter`를 사용하면 된다. 여기에 `ofPattern()`으로 원하는 포맷을 지정하면 된다. 여기서는 `yyyy년 MM월 dd일` 포맷을 지정했다.