6. 날짜와 시간

#1.인강/0.자바/3.자바-중급1편

날짜와 시간 라이브러리가 필요한 이유

날짜와 시간을 계산하는 것은 단순하게 생각하면 쉬워보이지만, 실제로는 매우 어렵고 복잡하다.
왜 그런지 이유를 하나하나 살펴보자.

1. 날짜와 시간 차이 계산

특정 날짜에서 다른 날짜까지의 정확한 일수를 계산하는 것은 생각보다 복잡하다. 윤년, 각 달의 일수 등을 모두 고려해야 하며, 간단한 뺄셈 연산으로는 정확한 결과를 얻기 어렵다.

예시

2024년 1월 1일에서 2024년 2월 1일까지는 몇 일일까? 이 계산은 1월이 31일 까지라는 점을 고려해야 한다. 각각의 월 마다 날짜가 다르다.

2. 윤년 계산

지구가 태양을 한 바퀴 도는 데 걸리는 평균 시간은 대략 365.2425 일, 즉 365일 5시간 48분 45초 정도이다. 우리가 사용하는 그레고리력(현재 대부분의 세계가 사용하는 달력)은 1년이 보통 365일로 설정되어 있다. 따라서 둘의 시간이 정확히 맞지 않다. 이런 문제를 해결하기 위해 4년마다 하루(2월 29일)를 추가하는 윤년(**leap year**)을 도입한다. 쉽게

이야기해서 2월은 보통 2월 28일까지 있는데, 4년마다 한번은 2월이 29일까지 하루 더 있다.

윤년 계산은 간단해 보이지만 실제로는 매우 복잡하다. 윤년은 보통 4년마다 한 번씩 발생하지만, 100년 단위일 때는 윤년이 아니며, 400년 단위일 때는 다시 윤년이다.
이 규칙에 따라, 2000년과 2020년은 윤년이지만, 1900년과 2100년은 윤년이 아니다. 이러한 규칙을 사용함으로써 달력 연도는 태양 연도에 매우 가깝게 유지될 수 있다.

예시

2024년 1월 1일에서 2024년 3월 1일까지는 몇 일일까? 이 계산은 2024년이 윤년으로 2월이 29일 까지라는 점을 고려해야 한다.

3. 일광 절약 시간(Daylight Saving Time, DST) 변환

보통 3월에서 10월은 태양이 일찍 뜨고, 나머지는 태양이 상대적으로 늦게 뜬다. 시간도 여기에 맞추어 1시간 앞당기거나 늦추는 제도를 일광 절약 시간제 또는 썸머타임이라 한다. 일광 절약 시간은 국가나 지역에 따라 적용 여부와 시작 및 종료 날짜가 다르다. 이로 인해 날짜와 시간 계산 시 1시간의 오차가 발생할 수 있으며, 이를 정확히 계산하는 것은 복잡하다.
줄여서 DST는 각 나라마다 다르지만 보통 3월 중순 ~ 11월 초 정도까지 시행된다.
참고로 대한민국에서는 1988년 이후로는 시행하지 않는다.

예시

특정 지역에서는 3월의 마지막 일요일에 DST가 시작되어 10월의 마지막 일요일에 종료된다. 이 기간 동안 발생하는 모든 날짜와 시간 계산은 1시간을 추가하거나 빼는 로직을 포함해야 한다

4. 타임존 계산

세계는 다양한 타임존으로 나뉘어 있으며, 각 타임존은 UTC(협정 세계시)로부터의 시간 차이로 정의된다. 타임존 간의 날짜와 시간 변환을 정확히 계산하는 것은 복잡하다.

타임존 목록

- Europe/London
- GMT
- UTC
- US/Arizona -07:00
- America/New_York -05:00
- Asia/Seoul +09:00
- Asia/Dubai +04:00
- Asia/Istanbul +03:00

- Asia/Shanghai +08:00
- Europe/Paris +01:00
- Europe/Berlin +01:00

GMT, UTC

London/ UTC / GMT는 세계 시간의 기준이 되는 00:00 시간대이다.

GMT (그리니치 평균시, Greenwich Mean Time)

처음 세계 시간을 만들 때 영국 런던에 있는 그리니치 천문대를 기준으로 했다. 태양이 그리니치 천문대를 통과할 때를 정오로 한다.

UTC(협정 세계시, Universal Time Coordinated)

역사적으로 GMT가 국제적인 시간 표준으로 사용되었고, UTC가 나중에 이를 대체하기 위해 도입되었다.
둘 다 경도 0°에 위치한 영국의 그리니치 천문대를 기준으로 하며, 이로 인해 실질적으로 같은 시간대를 나타낸다. 그러나 두 시간 체계는 시간을 정의하고 유지하는 방법에서 차이가 있다.
UTC는 원자 시계를 사용하여 측정한 국제적으로 합의된 시간 체계이다. 지구의 자전 속도가 변화하는 것을 고려하여 윤초를 추가하거나 빼는 방식으로 시간을 조정함으로써, 보다 정확한 시간을 유지한다. 우리가 일반적으로 사용할 때는 GMT와 UTC는 거의 차이가 없기 때문에 GMT와 UTC가 종종 같은 의미로 사용될 수 있지만, 정밀한 시간 측정과 국제적인 표준에 관해서는 UTC가 선호된다.

예시

상황: 서울에 있는 사람이 독일 베를린에 있는 사람과 미팅을 계획하고 있다. 서울의 타임존은 Asia/Seoul, UTC+9에 위치해 있고, 베를린의 타임존은 Europe/Berlin, UTC+1에 위치해 있다. 서울에서 오후 9:00에 미팅을 하려면 베를린에서는 몇 시일까?

타임존 차이: 서울(UTC+9)와 베를린(UTC+1) 사이의 타임존 차이는 8시간이다. 이는 서울의 시간이 베를린의 시간보다 8시간 더 앞서있다는 것을 의미한다.

계산:

- 서울 시간: 오후 9시
- 베를린 시간: 오후 9시 - 8시간 = 오후 1시

주의할 점

- **일광 절약 시간(DST)**: 일광 절약 시간이 적용되는 경우, 타임존 차이가 변할 수 있다. 예를 들어, 베를린에서 DST가 적용되면 UTC+1 ➔ UTC+2가 되어, 타임존 차이는 7시간으로 줄어든다.
- 예를 들어 베를린의 경우 3월 마지막 일요일에서 10월 마지막 일요일까지 DST가 적용된다.

이러한 복잡성 때문에 대부분의 현대 개발 환경에서는 날짜와 시간을 처리하기 위해 잘 설계된 라이브러리를 사용해야 한다. 이러한 라이브러리는 위에서 언급한 복잡한 계산을 추상화하여 제공하므로, 개발자는 보다 안정적이고 정확하며 효율적인 코드를 작성할 수 있다.

자바 날짜와 시간 라이브러리의 역사

자바는 날짜와 시간 라이브러리를 지속해서 업데이트 했다.

JDK 1.0 (java.util.Date)

- **문제점**
  - **타임존 처리 부족**: 초기 `Date` 클래스는 타임존(time zone)을 제대로 처리하지 못했다.
  - **불편한 날짜 시간 연산**: 날짜 간 연산이나 시간의 증감 등을 처리하기 어려웠다.
  - **불변 객체 부재**: `Date` 객체는 변경 가능(mutable)하여, 데이터가 쉽게 변경될 수 있었고 이로 인해 버그가 발생하기 쉬웠다.
- **해결책**
  - JDK 1.1에서 `java.util.Calendar` 클래스 도입으로 타임존 지원 개선.
  - 날짜 시간 연산을 위한 추가 메소드 제공.

JDK 1.1 (java.util.Calendar)

- **문제점**
  - **사용성 저하**: `Calendar`는 사용하기 복잡하고 직관적이지 않았다.
  - **성능 문제**: 일부 사용 사례에서 성능이 저하되는 문제가 있었다.
  - **불변 객체 부재**: `Calendar` 객체도 변경 가능하여, 사이드 이펙트, 스레드 안전성 문제가 있었다.
- **해결책**
  - Joda-Time 오픈소스 라이브러리의 도입으로 사용성, 성능, 불변성 문제 해결.

Joda-Time

- **문제점**
  - **표준 라이브러리가 아님**: Joda-Time은 외부 라이브러리로, 자바 표준에 포함되지 않아 프로젝트에 별도로 추가해야 했다.
- **해결책**
  - 자바 8에서 `java.time` 패키지(JSR-310)를 표준 API로 도입.

JDK 8(1.8) (java.time 패키지)

- `java.time` 패키지는 이전 API의 문제점을 해결하면서, 사용성, 성능, 스레드 안전성, 타임존 처리 등에서 크게 개선되었다. 변경 불가능한 불변 객체 설계로 사이드 이펙트, 스레드 안전성 보장, 보다 직관적인 API 제공으로 날

짜와 시간 연산을 단순화했다.

- `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `Instant` 등의 클래스를 포함한다.
- Joda-Time의 많은 기능을 표준 자바 플랫폼으로 가져왔다.

참고

자바가 표준으로 제공했던 `Date`, `Calendar`는 사용성이 너무 떨어지고, 문제가 많은 라이브러리였다. 이런 문제를 해결하기 위해 참다참다 결국 Joda-Time이라는 오픈소스 라이브러리가 등장한다. Joda-Time의 편리함과 사용성 덕분에 이 라이브러리는 크게 대중화 되었다. 자바는 기존 날짜와 시간의 설계를 반성하고, Joda-Time을 만든 개발자를 대려와서 JSR-310(`java.time`)이라는 새로운 자바 표준 날짜와 시간 라이브러리를 정의한다.
실용적인 Joda-Time에 많은 자바 커뮤니티의 의견을 반영해서 좀 더 안정적이고 표준적인 날짜와 시간 라이브러리인 `java.time` 패키지가 성공적으로 완성되었다.
참고로 자바 표준 ORM 기술인 JPA도 비슷한 역사를 가지고 있다. 과거 자바가 표준으로 제공하는 ORM 기술이 너무 불편해서, 누군가 하이버네이트라는 ORM 오픈 소스를 만들었는데, 이 기술이 자바 표준 ORM 기술보다 더 대중화되었다. 자바는 하이버네이트를 만든 개발자를 대려와서 새로운 자바 ORM 기술 표준을 만들었는데, 이것이 바로 JPA이다.
`java.time`, JPA 모두 큰 성공을 거두고 자바의 메인 표준 기술로 완전히 자리 잡았다.

자바 날짜와 시간 라이브러리 소개

자바 날짜와 시간 라이브러리는 자바 공식 문서가 제공하는 다음 표 하나로 정리할 수 있다.

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 표현할 때 사용한다. 년, 월, 일을 다룬다. 예) `2013-11-21`
- **LocalTime**: 시간만을 표현할 때 사용한다. 시, 분, 초를 다룬다. 예) `08:20:30.213`
  - 초는 밀리초, 나노초 단위도 포함할 수 있다.
- **LocalDateTime**: `LocalDate`와 `LocalTime`을 합한 개념이다. 예) `2013-11-21T08:20:30.213`

앞에 `Local`(현지의, 특정 지역의)이 붙는 이유는 세계 시간대를 고려하지 않아서 타임존이 적용되지 않기 때문이다.
특정 지역의 날짜와 시간만 고려할 때 사용한다.
예)

- 애플리케이션 개발시 국내 서비스만 고려할 때
- 나의 생일은 2016년 8월 16일이야.

ZonedDateTime, OffsetDateTime

- **ZonedDateTime**: 시간대를 고려한 날짜와 시간을 표현할 때 사용한다. 여기에는 시간대를 표현하는 타임존이 포함된다.
  - 예) `2013-11-21T08:20:30.213+9:00[Asia/Seoul]`
  - `+9:00`은 UTC(협정 세계시)로 부터의 시간대 차이이다. 오프셋이라 한다. 한국은 UTC보다 +9:00 시간이다.
  - `Asia/Seoul`은 타임존이라 한다. 이 타임존을 알면 오프셋과 일광 절약 시간제에 대한 정보를 알 수 있다.
  - 일광 절약 시간제가 적용된다.
- **OffsetDateTime**: 시간대를 고려한 날짜와 시간을 표현할 때 사용한다. 여기에는 타임존은 없고, UTC로 부터의 시간대 차이인 고정된 오프셋만 포함한다.
  - 예) `2013-11-21T08:20:30.213+9:00`
  - 일광 절약 시간제가 적용되지 않는다.

`Asia/Seoul` 같은 타임존 안에는 일광 절약 시간제에 대한 정보와 UTC+9:00와 같은 UTC로 부터 시간 차이인 오프셋 정보를 모두 포함하고 있다.
일광 절약 시간제(DST, 썸머타임)을 알려면 타임존을 알아야 한다. 따라서 `ZonedDateTime`은 일광 절약 시간제를 함께 처리한다. 반면에 타임존을 알 수 없는 `OffsetDateTime`는 일광 절약 시간제를 처리하지 못한다.
`ZonedDateTime`은 시간대를 고려해야 할 때 실제 사용하는 날짜와 시간 정보를 나타내는 데 더 적합하고,
`OffsetDateTime`은 UTC로부터의 고정된 오프셋만을 고려해야 할 때 유용하다.

Year, Month, YearMonth, MonthDay

년, 월, 년월, 달일을 각각 다룰 때 사용한다. 자주 사용하지는 않는다.
`DayOfWeek`와 같이 월, 화, 수, 목, 금, 토, 일을 나타내는 클래스도 있다.

Instant

`Instant`는 UTC(협정 세계시)를 기준으로 하는, 시간의 한 지점을 나타낸다. `Instant`는 날짜와 시간을 나노초 정밀도로 표현하며, 1970년 1월 1일 0시 0분 0초(UTC)를 기준으로 경과한 시간으로 계산된다.
쉽게 이야기해서 Instant 내부에는 초 데이터만 들어있다. (나노초 포함)
따라서 날짜와 시간을 계산에 사용할 때는 적합하지 않다. 자세한 내용은 뒤에서 다룬다.

Period, Duration

시간의 개념은 크게 2가지로 표현할 수 있다.

- 특정 시점의 시간(시각)
  - 이 프로젝트는 2013년 8월 16일 까지 완료해야해
  - 다음 회의는 11시 30분에 진행한다.
  - 내 생일은 8월 16일이야.

- 시간의 간격(기간)
  - 앞으로 4년은 더 공부해야 해
  - 이 프로젝트는 3개월 남았어
  - 라면은 3분 동안 끓어야 해

`Period`, `Duration`은 시간의 간격(기간)을 표현하는데 사용된다.
시간의 간격은 영어로 amount of time(시간의 양)으로 불린다.

Period

두 날짜 사이의 간격을 년, 월, 일 단위로 나타낸다.

Duration

두 시간 사이의 간격을 시, 분, 초(나노초) 단위로 나타낸다.

기본 날짜와 시간 - LocalDateTime

가장 기본이 되는 날짜와 시간 클래스는 `LocalDate`, `LocalTime`, `LocalDateTime`이다.

- **LocalDate**: 날짜만 표현할 때 사용한다. 년, 월, 일을 다룬다. 예) `2013-11-21`
- **LocalTime**: 시간만을 표현할 때 사용한다. 시, 분, 초를 다룬다. 예) `08:20:30.213`
  - 초는 밀리초, 나노초 단위도 포함할 수 있다.
- **LocalDateTime**: `LocalDate`와 `LocalTime`을 합한 개념이다. 예) `2013-11-21T08:20:30.213`

앞에 `Local`(현지의, 특정 지역의)이 붙는 이유는 세계 시간대를 고려하지 않아서 타임존이 적용되지 않기 때문이다.
특정 지역의 날짜와 시간만 고려할 때 사용한다.
예)

- 애플리케이션 개발시 국내 서비스만 고려할 때
- 나의 생일은 2016년 8월 16일이야.

LocalDate

```
package time;

import java.time.LocalDate;

public class LocalDateMain {
```

```
    public static void main(String[] args) {
        LocalDate nowDate = LocalDate.now();
        LocalDate ofDate = LocalDate.of(2013, 11, 21);
        System.out.println("오늘 날짜 = " + nowDate);
        System.out.println("지정 날짜 = " + ofDate);

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

실행 결과

```
오늘 날짜 = 2024-02-09
지정 날짜 = 2013-11-21
지정 날짜+10d = 2013-12-01
```

생성

- `now()` : 현재 시간을 기준으로 생성한다.
- `of(...)` : 특정 날짜를 기준으로 생성한다. 년, 월, 일을 입력할 수 있다.

계산

- `plusDays()` : 특정 일을 더한다. 다양한 `plusXxx()` 메서드가 존재한다.

주의! - 불변

모든 날짜 클래스는 불변이다. 따라서 변경이 발생하는 경우 새로운 객체를 생성해서 반환하므로 반환값을 꼭 받아야 한다.

LocalTime

```
package time;

import java.time.LocalTime;

public class LocalTimeMain {

    public static void main(String[] args) {
        LocalTime nowTime = LocalTime.now();
```

```
        LocalTime ofTime = LocalTime.of(9, 10, 30);

        System.out.println("현재 시간 = " + nowTime);
        System.out.println("지정 시간 = " + ofTime);

        //계산(불변)
        LocalTime ofTimePlus = ofTime.plusSeconds(30);
        System.out.println("지정 시간+30s = " + ofTimePlus);
    }
}
```

실행 결과

```
현재 시간 = 11:52:51.219602
지정 시간 = 09:10:30
지정 시간+30s = 09:11:00
```

생성

- `now()` : 현재 시간을 기준으로 생성한다.
- `of(...)` : 특정 시간을 기준으로 생성한다. 시, 분, 초, 나노초를 입력할 수 있다.

계산

- `plusSeconds()` : 특정 초를 더한다. 다양한 `plusXxx()` 메서드가 존재한다.

주의! - 불변

모든 날짜 클래스는 불변이다. 따라서 변경이 발생하는 경우 새로운 객체를 생성해서 반환하므로 반환값을 꼭 받아야 한다.

LocalDateTime

`LocalDateTime`은 `LocalDate`와 `LocalTime`을 내부에 가지고 날짜와 시간을 모두 표현한다.

LocalDateTime 클래스

```
public class LocalDateTime {
    private final LocalDate date;
    private final LocalTime time;
    ...
}
```

```java
package time;

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

public class LocalDateTimeMain {

    public static void main(String[] args) {
        LocalDateTime nowDt = LocalDateTime.now();
        LocalDateTime ofDt = LocalDateTime.of(2016, 8, 16, 8, 10, 1);
        System.out.println("현재 날짜시간 = " + nowDt);
        System.out.println("지정 날짜시간 = " + ofDt);

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

        //날짜와 시간 합체
        LocalDateTime localDateTime = LocalDateTime.of(localDate, localTime);
        System.out.println("localDateTime = " + localDateTime);

        //계산(불변)
        LocalDateTime ofDtPlus = ofDt.plusDays(1000);
        System.out.println("지정 날짜시간+1000d = " + ofDtPlus);
        LocalDateTime ofDtPlus1Year = ofDt.plusYears(1);
        System.out.println("지정 날짜시간+1년 = " + ofDtPlus1Year);

        //비교
        System.out.println("현재 날짜시간이 지정 날짜시간보다 이전인가? " +
nowDt.isBefore(ofDt));
        System.out.println("현재 날짜시간이 지정 날짜시간보다 이후인가? " +
nowDt.isAfter(ofDt));
        System.out.println("현재 날짜시간과 지정 날짜시간이 같은가? " +
nowDt.isEqual(ofDt));
    }
}
```

실행 결과

```
현재 날짜시간 = 2024-02-09T11:54:54.389163
지정 날짜시간 = 2016-08-16T08:10:01
```

```
localDate = 2016-08-16
localTime = 08:10:01
localDateTime = 2016-08-16T08:10:01
지정 날짜시간+1000d = 2019-05-13T08:10:01
지정 날짜시간+1년 = 2017-08-16T08:10:01
현재 날짜시간이 지정 날짜시간보다 이전인가? false
현재 날짜시간이 지정 날짜시간보다 이후인가? true
현재 날짜시간과 지정 날짜시간이 같은가? false
```

생성

- `now()` : 현재 날짜와 시간을 기준으로 생성한다.
- `of(...)` : 특정 날짜와 시간을 기준으로 생성한다.

분리

날짜(`LocalDate`)와 시간(`LocalTime`)을 `toXxx()` 메서드로 분리할 수 있다.

합체

`LocalDateTime.of(localDate, localTime)`
날짜와 시간을 사용해서 `LocalDateTime`을 만든다.

계산

- `plusXxx()` : 특정 날짜와 시간을 더한다. 다양한 `plusXxx()` 메서드가 존재한다.

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 일광 절약 시간제에 대한 정보와 UTC+9:00와 같은 UTC로 부터 시간 차이인 오프셋 정보를 모두 포함하고 있다.

타임존 목록 예시

- Europe/London
- GMT
- UTC
- US/Arizona -07:00
- America/New_York -05:00
- Asia/Seoul +09:00
- Asia/Dubai +04:00
- Asia/Istanbul +03:00
- Asia/Shanghai +08:00
- Europe/Paris +01:00
- Europe/Berlin +01:00

ZoneId

자바는 타임존을 `ZondId` 클래스로 제공한다.

```
package time;

import java.time.ZoneId;

public class ZoneIdMain {

    public static void main(String[] args) {
        for (String availableZoneId : ZoneId.getAvailableZoneIds()) {
            ZoneId zoneId = ZoneId.of(availableZoneId);
            System.out.println(zoneId + " | " + zoneId.getRules());
        }

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

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

```
    }
}
```

실행 결과

```
Europe/London | ZoneRules[currentStandardOffset=Z]
UTC | ZoneRules[currentStandardOffset=Z]
GMT | ZoneRules[currentStandardOffset=Z]
Asia/Seoul | ZoneRules[currentStandardOffset=+09:00]
Asia/Dubai | ZoneRules[currentStandardOffset=+04:00]
US/Arizona | ZoneRules[currentStandardOffset=-07:00]
Asia/Istanbul | ZoneRules[currentStandardOffset=+03:00]
Asia/Shanghai | ZoneRules[currentStandardOffset=+08:00]
...
Europe/Paris | ZoneRules[currentStandardOffset=+01:00]
ZoneId.systemDefault = Asia/Seoul
seoulZoneId = Asia/Seoul
```

생성

- `ZoneId.systemDefault()` : 시스템이 사용하는 기본 `ZoneId`를 반환한다.
  - 각 PC 환경 마다 다른 결과가 나올 수 있다.
- `ZoneId.of()` : 타임존을 직접 제공해서 `ZoneId`를 반환한다.

`ZoneId`는 내부에 일광 절약 시간 관련 정보, UTC와의 오프셋 정보를 포함하고 있다.

ZonedDateTime

`ZonedDateTime`은 `LocalDateTime`에 시간대 정보인 `ZoneId`가 합쳐진 것이다.

ZonedDateTime 클래스

```
public class ZonedDateTime {
    private final LocalDateTime dateTime;
    private final ZoneOffset offset;
    private final ZoneId zone;
}
```

ZonedDateTime: 시간대를 고려한 날짜와 시간을 표현할 때 사용한다. 여기에는 시간대를 표현하는 타임존이 포함된다.

- 예) `2013-11-21T08:20:30.213+9:00[Asia/Seoul]`

- `+9:00`은 UTC(협정 세계시)로 부터의 시간대 차이이다. 오프셋이라 한다. 한국은 UTC보다 +9:00 시간이다.
- `Asia/Seoul`은 타임존이라고 한다. 이 타임존을 알면 오프셋도 알 수 있다. +9:00 같은 오프셋 정보도 타임존에 포함된다.
- 추가로 `ZoneId`를 통해 타임존을 알면 일광 절약 시간제에 대한 정보도 알 수 있다. 따라서 일광 절약 시간제가 적용된다.

```
package time;

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

public class ZonedDateTimeMain {

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

        LocalDateTime ldt = LocalDateTime.of(2030, 1, 1, 13, 30, 50);
        ZonedDateTime zdt1 = ZonedDateTime.of(ldt, ZoneId.of("Asia/Seoul"));
        System.out.println("zdt1 = " + zdt1);

        ZonedDateTime zdt2 = ZonedDateTime.of(2030, 1, 1, 13, 30, 50, 0,
ZoneId.of("Asia/Seoul"));
        System.out.println("zdt2 = " + zdt2);

        ZonedDateTime utcZdt = zdt2.withZoneSameInstant(ZoneId.of("UTC"));
        System.out.println("utcZdt = " + utcZdt);
    }
}
```

실행 결과

```
nowZdt = 2024-02-09T12:02:13.457712+09:00[Asia/Seoul]
zdt1 = 2030-01-01T13:30:50+09:00[Asia/Seoul]
zdt2 = 2030-01-01T13:30:50+09:00[Asia/Seoul]
utcZdt = 2030-01-01T04:30:50Z[UTC]
```

생성

- `now()` : 현재 날짜와 시간을 기준으로 생성한다. 이때 `ZoneId`는 현재 시스템을 따른다. (`ZoneId.systemDefault()`)

- `of(...)` : 특정 날짜와 시간을 기준으로 생성한다. `ZoneId`를 추가해야 한다.
  - `LocalDateTime`에 `ZoneId`를 추가해서 생성할 수 있다.

타임존 변경

- `withZoneSameInstant(ZoneId)` : 타임존을 변경한다. 타임존에 맞추어 시간도 함께 변경된다. 이 메서드를 사용하면 지금 다른 나라는 몇 시 인지 확인일 수 있다. 예를 들어서 서울이 지금 9시라면, UTC 타임존으로 변경하면 0시를 확인할 수 있다.

OffsetDateTime

`OffsetDateTime`은 `LocalDateTime`에 UTC 오프셋 정보인 `ZoneOffset`이 합쳐진 것이다.

OffsetDateTime 클래스

```
public class OffsetDateTime {
    private final LocalDateTime dateTime;
    private final ZoneOffset offset;
}
```

OffsetDateTime: 시간대를 고려한 날짜와 시간을 표현할 때 사용한다. 여기에는 타임존은 없고, UTC로 부터의 시간대 차이인 고정된 오프셋만 포함한다.

- 예) `2013-11-21T08:20:30.213+9:00`
- `ZoneId`가 없으므로 일광 절약 시간제가 적용되지 않는다.

```
package time;

import java.time.*;

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, 30, 50);
        System.out.println("ldt = " + ldt);
        OffsetDateTime odt = OffsetDateTime.of(ldt, ZoneOffset.of("+01:00"));
        System.out.println("odt = " + odt);
    }
}
```

실행 결과

```
nowOdt = 2024-02-13T15:03:36.422230+09:00
ldt = 2030-01-01T13:30:50
odt = 2030-01-01T13:30:50+01:00
```

`ZoneOffset`은 `+01:00`처럼 UTC와의 시간 차이인 오프셋 정보만 보관한다.

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

`Instant`는 UTC(협정 세계시)를 기준으로 하는, 시간의 한 지점을 나타낸다. `Instant`는 날짜와 시간을 나노초 정밀도로 표현하며, 1970년 1월 1일 0시 0분 0초(UTC 기준)를 기준으로 경과한 시간으로 계산된다.

쉽게 이야기해서 Instant 내부에는 초 데이터만 들어있다. (나노초 포함)

따라서 날짜와 시간을 계산에 사용할 때는 적합하지 않다.

Instant 클래스

```
public class Instant {
    private final long seconds;
    private final int nanos;
    ...
}
```

- UTC 기준 1970년 1월 1일 0시 0분 0초라면 `seconds`에 0이 들어간다.

- UTC 기준 1970년 1월 1일 0시 0분 10초라면 `seconds`에 10이 들어간다.
- UTC 기준 1970년 1월 1일 0시 1분 0초라면 `seconds`에 60이 들어간다.

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.

Instant 특징

- **장점**:
  - **시간대 독립성**: `Instant`는 UTC를 기준으로 하므로, 시간대에 영향을 받지 않는다. 이는 전 세계 어디서나 동일한 시점을 가리키는데 유용하다.
  - **고정된 기준점**: 모든 `Instant`는 1970년 1월 1일 UTC를 기준으로 하기 때문에, 시간 계산 및 비교가 명확하고 일관된다.
- **단점**:
  - **사용자 친화적이지 않음**: `Instant`는 기계적인 시간 처리에는 적합하지만, 사람이 읽고 이해하기에는 직관적이지 않다. 예를 들어, 날짜와 시간을 계산하고 사용하는데 필요한 기능이 부족하다.
  - **시간대 정보 부재**: `Instant`에는 시간대 정보가 포함되어 있지 않아, 특정 지역의 날짜와 시간으로 변환하려면 추가적인 작업이 필요하다.

사용 예

- **전 세계적인 시간 기준 필요 시**: `Instant`는 UTC를 기준으로 하므로, 전 세계적으로 일관된 시점을 표현할 때 사용하기 좋다. 예를 들어, 로그 기록이나, 트랜잭션 타임스탬프, 서버 간의 시간 동기화 등이 이에 해당한다.
- **시간대 변환 없이 시간 계산 필요 시**: 시간대의 변화 없이 순수하게 시간의 흐름(예: 지속 시간 계산)만을 다루고 싶을 때 `Instant`가 적합하다. 이는 시간대 변환의 복잡성 없이 시간 계산을 할 수 있게 해준다.
- **데이터 저장 및 교환**: 데이터베이스에 날짜와 시간 정보를 저장하거나, 다른 시스템과 날짜와 시간 정보를 교환할 때 `Instant`를 사용하면, 모든 시스템에서 동일한 기준점(UTC)을 사용하게 되므로 데이터의 일관성을 유지하기 쉽다.

일반적으로 날짜와 시간을 사용할 때는 `LocalDateTime`, `ZonedDateTime`등을 사용하면 된다. `Instant`는 날짜를 계산하기 어렵기 때문에 앞서 사용 예와 같은 특별한 경우에 한정해서 사용하면 된다.

```
package time;

import java.time.Instant;
```

```java
import java.time.ZonedDateTime;

public class InstantMain {

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

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

        Instant epochStart = Instant.ofEpochSecond(0);
        System.out.println("epochStart = " + epochStart);

        //계산
        Instant later = epochStart.plusSeconds(3600);
        System.out.println("later = " + later);

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

실행 결과

```
now = 2024-02-13T06:46:07.101393Z
from = 2024-02-13T06:46:07.117732Z
epochStart = 1970-01-01T00:00:00Z
later = 1970-01-01T01:00:00Z
laterEpochSecond = 3600
```

생성

- `now()` : UTC를 기준 현재 시간의 `Instant` 를 생성한다.
- `from()` : 다른 타입의 날짜와 시간을 기준으로 `Instant` 를 생성한다. 참고로 `Instant` 는 UTC를 기준으로 하기 때문에 시간대 정보가 필요하다. 따라서 `LocalDateTime` 은 사용할 수 없다.
- `ofEpochSecond()` : 에포크 시간을 기준으로 `Instant` 를 생성한다. 0초를 선택하면 에포크 시간인 1970년 1월 1일 0시 0분 0초로 생성된다.

계산

- `plusSeconds()` : 초를 더한다. 초, 밀리초, 나노초 정도만 더하는 간단한 메서드가 제공된다.

조회

- `getEpochSecond()` : 에포크 시간인 UTC 1970년 1월 1일 0시 0분 0초를 기준으로 흐른 초를 반환한다.
- 여기서는 앞서 에포크 시간에 3600초를 더했기 때문에 3600이 반환된다.

Instant 정리

- 조금 특별한 시간, 기계 중심, UTC 기준
- 에포크 시간으로부터 흐른 시간을 초 단위로 저장
- 전세계 모든 서버 시간을 똑같이 맞출 수 있음, 항상 UTC 기준이므로 한국에 있는 `Instant`, 미국에 있는 `Instant`의 시간이 똑같음
- 서버 로그, epoch 시간 기반 계산이 필요할 때, 간단히 두 시간의 차이를 구할 때
- 단점: 초 단위의 간단한 연산 가능, 복잡한 연산 못함
- 대안: 날짜 계산 필요하면 `LocalDateTime` 또는, `ZonedDateTime` 사용

기간, 시간의 간격 - Duration, Period

시간의 개념은 크게 2가지로 표현할 수 있다.

- 특정 시점의 시간(시각)
  - 이 프로젝트는 2013년 8월 16일 까지 완료해야해
  - 다음 회의는 11시 30분에 진행한다.
  - 내 생일은 8월 16일이야.
- 시간의 간격(기간)
  - 앞으로 4년은 더 공부해야 해
  - 이 프로젝트는 3개월 남았어
  - 라면은 3분 동안 끓어야 해

`Period`, `Duration`은 시간의 간격(기간)을 표현하는데 사용된다.
시간의 간격은 영어로 amount of time(시간의 양)으로 불린다.

Period

두 날짜 사이의 간격을 년, 월, 일 단위로 나타낸다.

- 이 프로젝트는 3개월 정도 걸릴 것 같아

- 기념일이 183일 남았어
- 프로젝트 시작일과 종료일 사이의 간격: 프로젝트 기간

Duration

두 시간 사이의 간격을 시, 분, 초(나노초) 단위로 나타낸다.

- 라면을 끓이는 시간은 3분이야
- 영화 상영 시간은 2시간 30분이야
- 서울에서 부산까지는 4시간이 걸려

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

Period

두 날짜 사이의 간격을 년, 월, 일 단위로 나타낸다.

```
public class Period {
    private final int years;
    private final int months;
    private final int days;
}
```

```
package time;

import java.time.LocalDate;
import java.time.Period;

public class PeriodMain {

    public static void main(String[] args) {
        //생성
        Period period = Period.ofDays(10);
        System.out.println("period = " + period);
```

```
        //계산에 사용
        LocalDate currentDate = LocalDate.of(2030, 1, 1);
        LocalDate plusDate = currentDate.plus(period);
        System.out.println("현재 날짜: " + currentDate);
        System.out.println("더한 날짜: " + plusDate);

        //기간 차이
        LocalDate startDate = LocalDate.of(2023, 1, 1);
        LocalDate endDate = LocalDate.of(2023, 4, 2);
        Period between = Period.between(startDate, endDate);
        System.out.println("기간: " + between.getMonths() + "개월 " +
between.getDays() + "일");
    }
}
```

실행 결과

```
period = P10D
현재 날짜: 2030-01-01
더한 날짜: 2030-01-11
기간: 3개월 1일
```

생성

- `of()` : 특정 기간을 지정해서 `Period`를 생성한다.
  - `of(년, 월, 일)`
  - `ofDays()`
  - `ofMonths()`
  - `ofYears()`

계산에 사용

- 2030년 1월 1일에 10일을 더하면 2030년 1월 11일이 된다. 라고 표현할 때 특정 날짜에 10일이라는 기간을 더할 수 있다.

기간 차이

- 2023년 1월 1일과 2023년 4월 2일간의 차이는 3개월 1일이다. 라고 표현할 때 특정 날짜의 차이를 구하면 기간이 된다.
- `Period.between(startDate, endDate)`와 같이 특정 날짜의 차이를 구하면 `Period`가 반환된다.

Duration

두 시간 사이의 간격을 시, 분, 초(나노초) 단위로 나타낸다.

```
public class Duration {
    private final long seconds;
    private final int nanos;
}
```

내부에서 초를 기반으로 시, 분, 초를 계산해서 사용한다.

- 1분 = 60초
- 1시간 = 3600초

```
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("기준 시간 = " + lt);

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

        //시간 차이
        LocalTime start = LocalTime.of(9, 0);
        LocalTime end = LocalTime.of(10, 0);
        Duration between = Duration.between(start, end);
        System.out.println("차이: " + between.getSeconds() + "초");
        System.out.println("근무 시간: " + between.toHours() + "시간 " +
between.toMinutesPart() + "분");
    }
}
```

실행 결과

```
duration = PT30M
```

```
기준 시간 = 01:00
더한 시간 = 01:30
차이: 3600초
근무 시간: 1시간 0분
```

생성

- `of()` : 특정 시간을 지정해서 `Duration`를 생성한다.
  - `of(지정)`
  - `ofSeconds()`
  - `ofMinutes()`
  - `ofHours()`

계산에 사용

- 1:00에 30분을 더하면 1:30이 된다. 라고 표현할 때 특정 시간에 30분이라는 시간(시간의 간격)을 더할 수 있다.

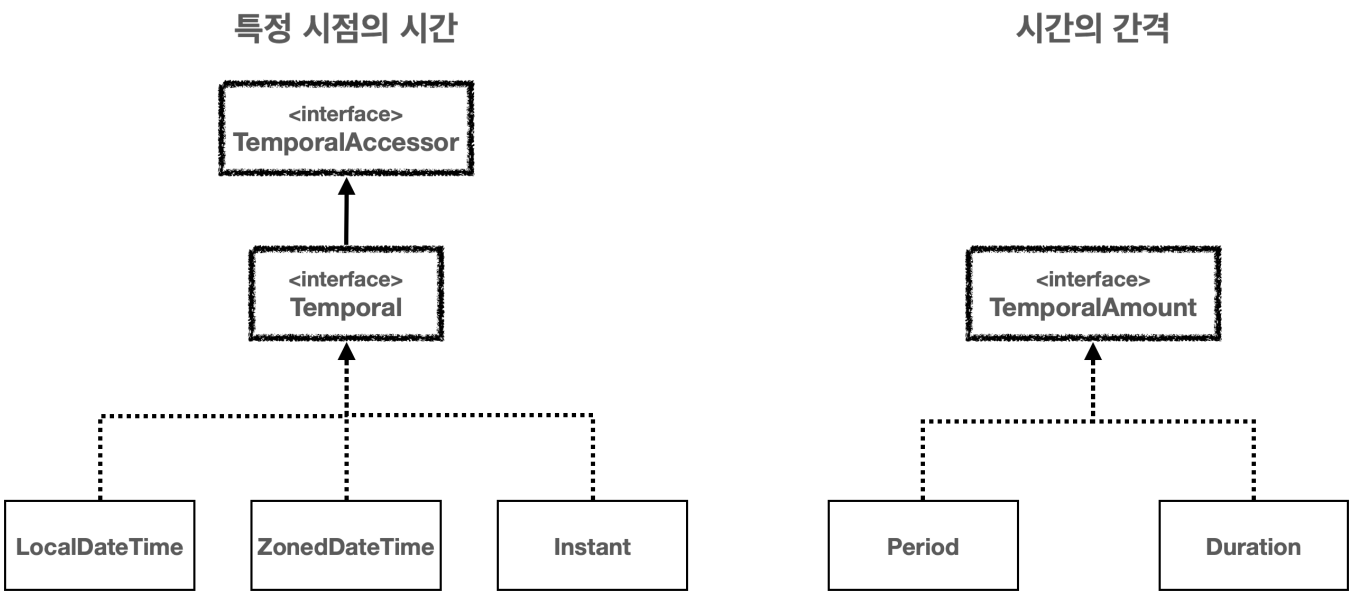
시간 차이

- 9시와 10시의 차이는 1시간이라고 표현할 때 시간의 차이를 구할 수 있다.
- `Duration.between(start, end)`와 같이 특정 시간의 차이를 구하면 `Duration`이 반환된다.

날짜와 시간의 핵심 인터페이스

날짜와 시간은 특정 시점의 시간(시각)과 시간의 간격(기간)으로 나눌 수 있다.

- 특정 시점의 시간(시각)
  - 이 프로젝트는 2013년 8월 16일 까지 완료해야해
  - 다음 회의는 11시 30분에 진행한다.
  - 내 생일은 8월 16일이야.
- 시간의 간격(기간, 시간의 양)
  - 앞으로 4년은 더 공부해야 해
  - 이 프로젝트는 3개월 남았어
  - 라면은 3분 동안 끓어야 해



- 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

- 날짜와 시간을 읽기 위한 기본 인터페이스
- 이 인터페이스는 특정 시점의 날짜와 시간 정보를 읽을 수 있는 최소한의 기능을 제공한다.

Temporal 인터페이스

- `TemporalAccessor`의 하위 인터페이스로, 날짜와 시간을 조작(추가, 빼기 등)하기 위한 기능을 제공한다. 이를 통해 날짜와 시간을 변경하거나 조정할 수 있다.

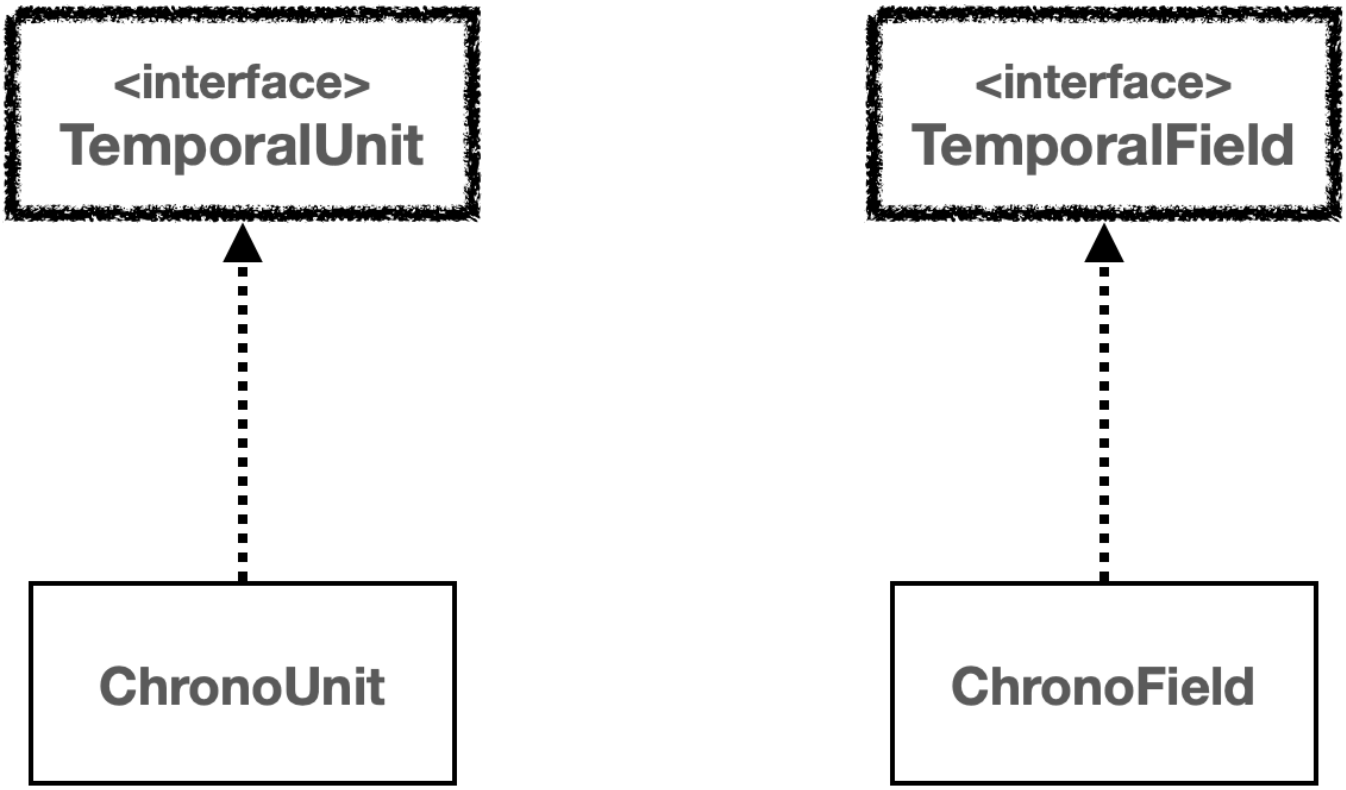
간단히 말하면, `TemporalAccessor`는 읽기 전용 접근을, `Temporal`은 읽기와 쓰기(조작) 모두를 지원한다.

TemporalAmount 인터페이스

시간의 간격(시간의 양, 기간)을 나타내며, 날짜와 시간 객체에 적용하여 그 객체를 조정할 수 있다. 예를 들어, 특정 날짜에 일정 기간을 더하거나 빼는 데 사용된다.

시간의 단위와 시간 필드

다음으로 설명할 날짜와 시간의 핵심 인터페이스는 시간의 단위를 뜻하는 `TemporalUnit`(`ChronoUnit`)과 시간의 각 필드를 뜻하는 `TemporalField`(`ChronoField`)이다.



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 단위

ChronoUnit	설명
NANOS	나노초 단위
MICROS	마이크로초 단위
MILLIS	밀리초 단위
SECONDS	초 단위
MINUTES	분 단위
HOURS	시간 단위

날짜 단위

ChronoUnit	설명
DAYS	일 단위
WEEKS	주 단위
MONTHS	월 단위
YEARS	년 단위
DECADES	10년 단위
CENTURIES	세기 단위
MILLENNIA	천년 단위

기타 단위

ChronoUnit	설명
ERAS	시대 단위
FOREVER	무한대의 시간 단위

ChronoUnit의 주요 메서드

메서드 이름	설명
between(Temporal, Temporal)	두 Temporal 객체 사이의 시간을 현재 ChronoUnit 단위로 측정하여 반환다.
isDateBased()	현재 ChronoUnit이 날짜 기반 단위인지 (예: 일, 주, 월, 년) 여부를 반환한
isTimeBased()	현재 ChronoUnit이 시간 기반 단위인지 (예: 시, 분, 초) 여부를 반환한다.
isSupportedBy(Temporal)	주어진 Temporal 객체가 현재 ChronoUnit 단위를 지원하는지 여부를 반한다.
getDuration()	현재 ChronoUnit의 기간을 Duration 객체로 반환한다.

```
package time;

import java.time.LocalTime;
```

```java
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("HOURS = " + ChronoUnit.HOURS);
        System.out.println("HOURS.duration = " +
ChronoUnit.HOURS.getDuration().getSeconds());
        System.out.println("DAYS = " + ChronoUnit.DAYS);
        System.out.println("DAYS.duration = " +
ChronoUnit.DAYS.getDuration().getSeconds());

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

        long secondsBetween = ChronoUnit.SECONDS.between(lt1, lt2);
        System.out.println("secondsBetween = " + secondsBetween);

        long minutesBetween = ChronoUnit.MINUTES.between(lt1, lt2);
        System.out.println("minutesBetween = " + minutesBetween);
    }
}
```

실행 결과

```
value = Nanos
value = Micros
value = Millis
value = Seconds
value = Minutes
value = Hours
value = HalfDays
value = Days
value = Weeks
value = Months
value = Years
value = Decades
value = Centuries
value = Millennia
```

```
value = Eras
value = Forever
HOURS = Hours
HOURS.duration = 3600
DAYS = Days
DAYS.duration = 86400
secondsBetween = 600
minutesBetween = 10
```

`ChronoUnit`을 사용하면 두 날짜 또는 시간 사이의 차이를 해당 단위로 쉽게 계산할 수 있다.
예제 코드에서는 두 `LocalTime` 객체 사이의 차이를 초, 분 단위로 구한다.

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.

연도 관련 필드

필드 이름	설명
ERA	연대, 예를 들어 서기(AD) 또는 기원전(BC)
YEAR_OF_ERA	연대 내의 연도
YEAR	연도

EPOCH_DAY	1970-01-01부터의 일 수

월 관련 필드

필드 이름	설명
MONTH_OF_YEAR	월 (1월 = 1)
PROLEPTIC_MONTH	연도를 월로 확장한 값

주 및 일 관련 필드

필드 이름	설명
DAY_OF_WEEK	요일 (월요일 = 1)
ALIGNED_DAY_OF_WEEK_IN_MONTH	월의 첫 번째 요일을 기준으로 정렬된 요일
ALIGNED_DAY_OF_WEEK_IN_YEAR	연의 첫 번째 요일을 기준으로 정렬된 요일
DAY_OF_MONTH	월의 일 (1일 = 1)
DAY_OF_YEAR	연의 일 (1월 1일 = 1)
EPOCH_DAY	유닉스 에폭(1970-01-01)부터의 일 수

시간 관련 필드

필드 이름	설명
HOUR_OF_DAY	시간 (0-23)
CLOCK_HOUR_OF_DAY	시계 시간 (1-24)
HOUR_OF_AMPM	오전/오후 시간 (0-11)
CLOCK_HOUR_OF_AMPM	오전/오후 시계 시간 (1-12)
MINUTE_OF_HOUR	분 (0-59)
SECOND_OF_MINUTE	초 (0-59)
NANO_OF_SECOND	초의 나노초 (0-999,999,999)
MICRO_OF_SECOND	초의 마이크로초 (0-999,999)

MILLI_OF_SECOND	초의 밀리초 (0-999)

기타 필드

필드 이름	설명
AMPM_OF_DAY	하루의 AM/PM 부분
INSTANT_SECONDS	초를 기준으로 한 시간
OFFSET_SECONDS	UTC/GMT에서의 시간 오프셋 초

주요 메서드

메서드	반환 타입	설명
getBaseUnit()	TemporalUnit	필드의 기본 단위를 반환한다. 예를 들어, 분 필드의 기본 단위는 ChronoUnit.MINUTES이다.
getRangeUnit()	TemporalUnit	필드의 범위 단위를 반환한다. 예를 들어, MONTH_OF_YEAR의 범위 단위는 ChronoUnit.YEARS이다.
isDateBased()	boolean	필드가 주로 날짜를 기반으로 하는지 여부를 나타냅니다. YEAR와 같은 날짜 기반 필드는 true를 반환한다.
isTimeBased()	boolean	필드가 주로 시간을 기반으로 하는지 여부를 나타낸다. HOUR_OF_DA와 같은 시간 기반 필드는 true를 반환한다.
range()	ValueRange	필드가 가질 수 있는 값의 유효 범위를 ValueRange 객체로 반환한다. 객체는 최소값과 최대값을 제공한다.

```
package time;

import java.time.temporal.ChronoField;

public class ChronoFieldMain {

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

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

실행 결과

```
NanoOfSecond, range = 0 - 999999999
NanoOfDay, range = 0 - 86399999999999
MicroOfSecond, range = 0 - 999999
MicroOfDay, range = 0 - 86399999999
MilliOfSecond, range = 0 - 999
MilliOfDay, range = 0 - 86399999
SecondOfMinute, range = 0 - 59
SecondOfDay, range = 0 - 86399
MinuteOfHour, range = 0 - 59
MinuteOfDay, range = 0 - 1439
HourOfAmPm, range = 0 - 11
ClockHourOfAmPm, range = 1 - 12
HourOfDay, range = 0 - 23
ClockHourOfDay, range = 1 - 24
AmPmOfDay, range = 0 - 1
DayOfWeek, range = 1 - 7
AlignedDayOfWeekInMonth, range = 1 - 7
AlignedDayOfWeekInYear, range = 1 - 7
DayOfMonth, range = 1 - 28/31
DayOfYear, range = 1 - 365/366
EpochDay, range = -365243219162 - 365241780471
AlignedWeekOfMonth, range = 1 - 4/5
AlignedWeekOfYear, range = 1 - 53
MonthOfYear, range = 1 - 12
ProlepticMonth, range = -11999999988 - 11999999999
YearOfEra, range = 1 - 999999999/1000000000
Year, range = -999999999 - 999999999
Era, range = 0 - 1
InstantSeconds, range = -9223372036854775808 - 9223372036854775807
OffsetSeconds, range = -64800 - 64800
MONTH_OF_YEAR.range() = 1 - 12
DAY_OF_MONTH.range() = 1 - 28/31
```

정리

`TemporalUnit(ChronoUnit)`, `TemporalField(ChronoField)`는 단독으로 사용하기 보다는 주로 날짜와 시간을 조회하거나 조작할 때 사용한다. 다음 시간을 통해서 날짜와 시간을 조회하고 조작하는 방법을 알아보자.

날짜와 시간 조회하고 조작하기1

날짜와 시간 조회하기

날짜와 시간을 조회하려면 날짜와 시간 항목중에 어떤 필드를 조회할 지 선택해야 한다. 이때 날짜와 시간의 필드를 뜻하는 `ChronoField`가 사용된다.

```
package time;

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

public class GetTimeMain {

    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2030, 1, 1, 13, 30, 59);
        System.out.println("YEAR = " + dt.get(ChronoField.YEAR));
        System.out.println("MONTH_OF_YEAR = " +
dt.get(ChronoField.MONTH_OF_YEAR));
        System.out.println("DAY_OF_MONTH = " +
dt.get(ChronoField.DAY_OF_MONTH));
        System.out.println("HOUR_OF_DAY = " + dt.get(ChronoField.HOUR_OF_DAY));
        System.out.println("MINUTE_OF_HOUR = " +
dt.get(ChronoField.MINUTE_OF_HOUR));
        System.out.println("SECOND_OF_MINUTE = " +
dt.get(ChronoField.SECOND_OF_MINUTE));

        System.out.println("편의 메서드 사용");
        System.out.println("YEAR = " + dt.getYear());
        System.out.println("MONTH_OF_YEAR = " + dt.getMonthValue());
        System.out.println("DAY_OF_MONTH = " + dt.getDayOfMonth());
        System.out.println("HOUR_OF_DAY = " + dt.getHour());
        System.out.println("MINUTE_OF_HOUR = " + dt.getMinute());
        System.out.println("SECOND_OF_MINUTE = " + dt.getSecond());
```

```
        System.out.println("편의 메서드에 없음");
        System.out.println("MINUTE_OF_DAY = " +
dt.get(ChronoField.MINUTE_OF_DAY));
        System.out.println("SECOND_OF_DAY = " +
dt.get(ChronoField.SECOND_OF_DAY));
    }
}
```

실행 결과

```
YEAR = 2030
MONTH_OF_YEAR = 1
DAY_OF_MONTH = 1
HOUR_OF_DAY = 13
MINUTE_OF_HOUR = 30
SECOND_OF_MINUTE = 59

편의 메서드 사용
YEAR = 2030
MONTH_OF_YEAR = 1
DAY_OF_MONTH = 1
HOUR_OF_DAY = 13
MINUTE_OF_HOUR = 30
SECOND_OF_MINUTE = 59

편의 메서드에 없음
MINUTE_OF_DAY = 810
SECOND_OF_DAY = 48659
```

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.

편의 메서드 사용

- `get(TemporalField field)`을 사용하면 코드가 길어지고 번거롭기 때문에 자주 사용하는 조회 필드는 간단한 편의 메서드를 제공한다.
- `dt.get(ChronoField.DAY_OF_MONTH))` ➜ `dt.getDayOfMonth()`

편의 메서드에 없음

- 자주 사용하지 않는 특별한 기능은 편의 메서드를 제공하지 않는다.
- 편의 메서드를 사용하는 것이 가독성이 좋기 때문에 일반적으로는 편의 메서드를 사용하고, 편의 메서드가 없는 경우 `get(TemporalField field)` 을 사용하면 된다.

날짜와 시간 조작하기

날짜와 시간을 조작하려면 어떤 시간 단위(Unit)를 변경할 지 선택해야 한다. 이때 날짜와 시간의 단위를 뜻하는 `ChronoUnit` 이 사용된다.

```
package time;

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

public class ChangeTimePlusMain {

    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2018, 1, 1, 13, 30, 59);
        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);
    }
}
```

```
dt = 2018-01-01T13:30:59
plusDt1 = 2028-01-01T13:30:59
plusDt2 = 2028-01-01T13:30:59
plusDt3 = 2028-01-01T13:30:59
```

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.

편의 메서드 사용

- 자주 사용하는 메서드는 편의 메서드가 제공된다.
- `dt.plus(10, ChronoUnit.YEARS)` ➜ `dt.plusYears(10)`

Period를 사용한 조작

`Period`나 `Duration`은 기간(시간의 간격)을 뜻한다. 특정 시점의 시간에 기간을 더할 수 있다.

정리

시간을 조회하고 조작하는 부분을 보면 `TemporalAccessor.get()`, `Temporal.plus()`와 같은 인터페이스를 통해 특정 구현 클래스와 무관하게 아주 일관성 있는 시간 조회, 조작 기능을 제공하는 것을 확인할 수 있다.
덕분에 `LocalDateTime`, `LocalDate`, `LocalTime`, `ZonedDateTime`, `Instant`와 같은 수 많은 구현에 관계없이 일관성 있는 방법으로 시간을 조회하고 조작할 수 있다.

하지만 모든 시간 필드를 다 조회할 수 있는 것은 아니다.

```
package time;

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

public class IsSupportedMain1 {

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

실행 결과

```
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
Unsupported field: SecondOfMinute
```

`LocalDate`는 날짜 정보만 가지고 있고, 분에 대한 정보는 없다. 따라서 분에 대한 정보를 조회하려고 하면 예외가 발생한다.
이런 문제를 예방하기 위해 `TemporalAccessor`와 `Temporal` 인터페이스는 현재 타입에서 특정 시간 단위나 필드를 사용할 수 있는지 확인할 수 있는 메서드를 제공한다.

TemporalAccessor

```
boolean isSupported(TemporalField field);
```

Temporal

```
boolean isSupported(TemporalUnit unit);
```

```
package time;

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

public class IsSupportedMain2 {

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

실행 결과

```
supported = false
```

`LocalDate`는 분의 초 필드를 지원하지 않으므로 `ChronoField.SECOND_OF_MINUTE`를 조회하면 `false`를 반환한다.

날짜와 시간 조회하고 조작하기2

날짜와 시간을 조작하는 `with()` 메서드에 대해 알아보자.

```
package time;

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

public class ChangeTimeWithMain {

    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2018, 1, 1, 13, 30, 59);
        System.out.println("dt = " + dt);

        LocalDateTime changedDt1 = dt.with(ChronoField.YEAR, 2020);
        System.out.println("changedDt1 = " + changedDt1);

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

        //TemporalAdjuster 사용
        //다음주 금요일
        LocalDateTime with1 = dt.with(TemporalAdjusters.next(DayOfWeek.FRIDAY));
        System.out.println("기준 날짜: "  + dt);
        System.out.println("다음 금요일: " + with1);

        //이번 달의 마지막 일요일
        LocalDateTime with2 =
dt.with(TemporalAdjusters.lastInMonth(DayOfWeek.SUNDAY));
        System.out.println("같은 달의 마지막 일요일 = " + with2);
    }
}
```

실행 결과

```
dt = 2018-01-01T13:30:59
changedDt1 = 2020-01-01T13:30:59
changedDt2 = 2020-01-01T13:30:59
기준 날짜: 2018-01-01T13:30:59
```

```
다음 금요일: 2018-01-05T13:30:59
같은 달의 마지막 일요일 = 2018-01-28T13:30:59
```

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

DayOfWeek

월, 화, 수, 목, 금, 토, 일을 나타내는 열거형이다.

TemporalAdjusters 클래스가 제공하는 주요 기능

메서드	설명
dayOfWeekInMonth	주어진 요일이 몇 번째인지에 따라 날짜를 조정한다.
firstDayOfMonth	해당 월의 첫째 날로 조정한다.
firstDayOfNextMonth	다음 달의 첫째 날로 조정한다.

firstDayOfNextYear	다음 해의 첫째 날로 조정한다.
firstDayOfYear	해당 해의 첫째 날로 조정한다.
firstInMonth	주어진 요일 중 해당 월의 첫 번째 요일로 조정한다.
lastDayOfMonth	해당 월의 마지막 날로 조정한다.
lastDayOfNextMonth	다음 달의 마지막 날로 조정한다.
lastDayOfNextYear	다음 해의 마지막 날로 조정한다.
lastDayOfYear	해당 해의 마지막 날로 조정한다.
lastInMonth	주어진 요일 중 해당 월의 마지막 요일로 조정한다.
next	주어진 요일 이후의 가장 가까운 요일로 조정한다.
nextOrSame	주어진 요일 이후의 가장 가까운 요일로 조정하되, 현재 날짜가 주어진 요일인 경우 현 날짜를 반환한다.
previous	주어진 요일 이전의 가장 가까운 요일로 조정한다.
previousOrSame	주어진 요일 이전의 가장 가까운 요일로 조정하되, 현재 날짜가 주어진 요일인 경우 현 날짜를 반환한다.

날짜와 시간 문자열 파싱과 포맷팅

- **포맷팅**: 날짜와 시간 데이터를 원하는 포맷의 문자열로 변경하는 것, `Date` ➜ `String`
- **파싱**: 문자열을 날짜와 시간 데이터로 변경하는 것, `String` ➜ `Date`

```
package time;

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

public class FormattingMain1 {
    public static void main(String[] args) {
        // 포맷팅: 날짜를 문자로
```

```
        LocalDate date = LocalDate.of(2024, 12, 31);
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy년 MM월 dd
일");
        String formattedDate = date.format(formatter);
        System.out.println("날짜와 시간 포맷팅: " + formattedDate);

        // 파싱: 문자를 날짜로
        String input = "2030년 01월 01일";
        LocalDate parsedDate = LocalDate.parse(input, formatter);
        System.out.println("문자열 파싱 날짜와 시간: " + parsedDate);
    }
}
```

실행 결과

```
현재 날짜와 시간 포맷팅: 2024년 12월 31일
문자열 파싱 날짜와 시간: 2030-01-01
```

`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.

DateTimeFormatter 패턴(공식 사이트)

기호	의미	표현 방식	예시
G	연호	텍스트	AD; Anno Domini; A
u	연도	연도	2004; 04
y	연대의 연도	연도	2004; 04
D	연중 일수	숫자	189
M/L	연중 월	숫자/텍스트	7; 07; Jul; July; J
d	월의 일수	숫자	10
Q/q	분기	숫자/텍스트	3; 03; Q3; 3rd quarter
Y	주 기준 연도	연도	1996; 96
w	주 기준 연중 주	숫자	27
W	월의 주	숫자	4

E	요일	텍스트	Tue; Tuesday; T
e/c	지역화된 요일	숫자/텍스트	2; 02; Tue; Tuesday; T
F	월의 주	숫자	3
a	오전/오후	텍스트	PM
h	오전/오후 시(1-12)	숫자	12
K	오전/오후 시(0-11)	숫자	0
k	24시간제 시(1-24)	숫자	0
H	24시간제 시(0-23)	숫자	0
m	분	숫자	30
s	초	숫자	55
S	초의 일부	분수	978
A	일의 밀리초	숫자	1234
n	초의 나노초	숫자	987654321
N	일의 나노초	숫자	1234000000
V	시간대 ID	시간대 ID	America/Los_Angeles; Z; -08:30
z	시간대 이름	시간대 이름	Pacific Standard Time; PST
O	지역화된 시간대 오프셋	오프셋-O	GMT+8; GMT+08:00; UTC-08:00;
X	'Z'로 표시된 시간대 오프셋	오프셋-X	Z; -08; -0830; -08:30; -083015; -08:30:15;
x	시간대 오프셋	오프셋-x	+0000; -08; -0830; -08:30; -083015; -08:30:1
Z	시간대 오프셋	오프셋-Z	+0000; -0800; -08:00;
p	다음 패딩	패드 수정자	1
'	텍스트를 위한 이스케이프	구분자	
''	단일 인용부호	리터럴	'

DateTimeFormatter 패턴 공식 사이트

https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns

문자열을 날짜와 시간으로 파싱

문자열을 읽어서 날짜와 시간 객체로 만드는 것을 파싱이라 한다.

이때 문자열의 어떤 부분이 년이고, 월이고 일인지 각각의 위치를 정해서 읽어야 한다.

`2030년 01월 01일`

```
LocalDate.parse(input, formatter)
```

이때도 앞서 설명한 `DateTimeFormatter`를 사용한다.

`2030년 01월 01일`의 경우 `yyyy년 MM월 dd일` 포맷으로 읽어들이면 된다.

이번에는 날짜에 시간까지 포함해서 문자열을 포맷팅하고 파싱해보자.

```
package time;

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

public class FormattingMain2 {
    public static void main(String[] args) {
        // 포맷팅: 날짜와 시간을 문자로
        LocalDateTime now = LocalDateTime.of(2024, 12, 31, 13, 30, 59);
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
        String formattedDateTime = now.format(formatter);
        System.out.println("날짜와 시간 포맷팅: " + formattedDateTime);

        // 파싱: 문자를 날짜와 시간으로
        String dateTimeString = "2030-01-01 11:30:00";
        LocalDateTime parsedDateTime = LocalDateTime.parse(dateTimeString,
formatter);
        System.out.println("문자열 파싱 날짜와 시간: " + parsedDateTime);
    }
}
```

실행 결과

```
현재 날짜와 시간 포맷팅: 2024-12-31 13:30:59
문자열 파싱 날짜와 시간: 2030-01-01T11:30
```

`LocalDateTime`과 같은 날짜와 시간 객체를 원하는 형태의 문자로 변경하려면 `DateTimeFormatter`를 사용하면 된다. 여기에 `ofPattern()`으로 원하는 포맷을 지정하면 된다. 여기서는 `yyyy-MM-dd HH:mm:ss` 포맷을 지정했다.

문자열을 날짜와 시간으로 파싱

문자열을 읽어서 날짜와 시간으로 파싱할 때는 년, 월, 일, 시, 분, 초의 위치를 정해서 읽어야 한다.
`2030-01-01 11:30:00`

```
LocalDateTime.parse(dateTimeString, formatter)
```

이때도 앞서 설명한 `DateTimeFormatter`를 사용한다.
`2030-01-01 11:30:00`의 경우 `yyyy-MM-dd HH:mm:ss` 포맷으로 읽어들이면 된다.

문제와 풀이1

`time.test` 패키지를 사용해라.

문제1 - 날짜 더하기

문제 설명

- 2024년 1월 1일 0시 0분 0초에 1년 2개월 3일 4시간 후의 시각을 찾아라.
- `TestPlus` 클래스에 문제를 풀어라.

실행 결과

```
기준 시각: 2024-01-01T00:00
1년 2개월 3일 4시간 후의 시각: 2025-03-04T04:00
```

정답

```
package time.test;

import java.time.LocalDateTime;
```

```
public class TestPlus {
    public static void main(String[] args) {
        LocalDateTime dateTime = LocalDateTime.of(2024, 1, 1, 0, 0, 0);
        LocalDateTime futureDateTime =
dateTime.plusYears(1).plusMonths(2).plusDays(3).plusHours(4);
        System.out.println("기준 시각: " + dateTime);
        System.out.println("1년 2개월 3일 4시간 후의 시각: " + futureDateTime);
    }
}
```

문제2 - 날짜 간격 반복 출력하기

문제 설명

- 2024년 1월 1일 부터 2주 간격으로 5번 반복하여 날짜를 출력하는 코드를 작성하세요.
- `TestLoopPlus` 클래스에 문제를 풀어라

실행 결과

```
날짜 1: 2024-01-01
날짜 2: 2024-01-15
날짜 3: 2024-01-29
날짜 4: 2024-02-12
날짜 5: 2024-02-26
```

정답

```
package time.test;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class TestLoopPlus {
    public static void main(String[] args) {
        LocalDate startDate = LocalDate.of(2024, 1, 1);
        for (int i = 0; i < 5; i++) {
            LocalDate nextDate = startDate.plus(2 * i, ChronoUnit.WEEKS);
            System.out.println("날짜 " + (i + 1) + ": " + nextDate);
        }
    }
}
```

문제3 - 디데이 구하기

문제 설명

- 시작 날짜와 목표 날짜를 입력해서 남은 기간과 디데이를 구해라. 실행 결과를 참고하자.
- 남은 기간: x년 x개월 x일 형식으로 출력한다.
- 디데이: x일 남은 형식으로 출력한다.

```
package time.test;

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

public class TestBetween {
    public static void main(String[] args) {
        LocalDate startDate = LocalDate.of(2024, 1, 1);
        LocalDate endDate = LocalDate.of(2024, 11, 21);

        // 코드 작성
    }
}
```

실행 결과

```
시작 날짜: 2024-01-01
목표 날짜: 2024-11-21
남은 기간: 0년 10개월 20일
디데이: 325일 남음
```

정답

```
package time.test;

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

public class TestBetween {
    public static void main(String[] args) {
        LocalDate startDate = LocalDate.of(2024, 1, 1);
        LocalDate endDate = LocalDate.of(2024, 11, 21);
```

```
        Period period = Period.between(startDate, endDate);
        long daysBetween = ChronoUnit.DAYS.between(startDate, endDate);

        System.out.println("시작 날짜: " + startDate);
        System.out.println("목표 날짜: " + endDate);
        System.out.println("남은 기간: " + period.getYears() + "년 " +
period.getMonths() + "개월 " + period.getDays() + "일 ");
        System.out.println("디데이: " + daysBetween + "일 남음");
    }
}
```

문제4 - 시작 요일, 마지막 요일 구하기

- 입력 받은 월의 첫날 요일과 마지막날 요일을 구해라.

```
package time.test;

import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;

public class TestAdjusters {

    public static void main(String[] args) {
        int year = 2024;
        int month = 1;

        // 코드 작성
    }
}
```

실행 결과

```
firstDayOfWeek = MONDAY
lastDayOfWeek = WEDNESDAY
```

정답

```
package time.test;

import java.time.DayOfWeek;
```

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

public class TestAdjusters {

    public static void main(String[] args) {
        //입력 받은 월의 첫날 요일과 마지막날 요일을 구해라.
        int year = 2024;
        int month = 1;

        LocalDate date = LocalDate.of(year, month, 1);
        DayOfWeek firstDayOfWeek = date.getDayOfWeek();
        DayOfWeek lastDayOfWeek =
date.with(TemporalAdjusters.lastDayOfMonth()).getDayOfWeek();
        System.out.println("firstDayOfWeek = " + firstDayOfWeek);
        System.out.println("lastDayOfWeek = " + lastDayOfWeek);
    }
}
```

문제5 - 국제 회의 시간

- 서울의 회의 시간은 2024년 1월 1일 오전 9시다. 이때 런던, 뉴욕의 회의 시간을 구해라.
- 실행 결과를 참고하자.
- `TestZone` 클래스에 문제를 풀어라.

실행 결과

```
서울의 회의 시간: 2024-01-01T09:00+09:00[Asia/Seoul]
런던의 회의 시간: 2024-01-01T00:00Z[Europe/London]
뉴욕의 회의 시간: 2023-12-31T19:00-05:00[America/New_York]
```

정답

```
package time.test;

import java.time.LocalDate;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class TestZone {
```

```
    public static void main(String[] args) {
        ZonedDateTime seoulTime = ZonedDateTime.of(LocalDate.of(2024, 1, 1),
LocalTime.of(9, 0), ZoneId.of("Asia/Seoul"));
        ZonedDateTime londonTime =
seoulTime.withZoneSameInstant(ZoneId.of("Europe/London"));
        ZonedDateTime nyTime = seoulTime.withZoneSameInstant(ZoneId.of("America/
New_York"));

        System.out.println("서울의 회의 시간: " + seoulTime);
        System.out.println("런던의 회의 시간: " + londonTime);
        System.out.println("뉴욕의 회의 시간: " + nyTime);
    }
}
```

문제와 풀이2

문제6 - 달력 출력하기

- 실행 결과를 참고해서 달력을 출력해라.
- 입력 조건: 년도, 월
- 실행시 날짜의 간격에는 신경을 쓰지 않아도 된다. 간격을 맞추는 부분은 정답을 참고하자.

실행 결과

```
년도를 입력하세요: 2024
월을 입력하세요: 1
Su Mo Tu We Th Fr Sa
    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
```

```
년도를 입력하세요: 2025
월을 입력하세요: 1
Su Mo Tu We Th Fr Sa
          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
```

정답

```java
package time.test;

import java.time.DayOfWeek;
import java.time.LocalDate;
import java.util.Scanner;

public class TestCalendarPrinter {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("년도를 입력하세요: ");
        int year = scanner.nextInt();

        System.out.print("월을 입력하세요: ");
        int month = scanner.nextInt();

        printCalendar(year, month);
    }

    public static void printCalendar(int year, int month) {
        LocalDate firstDayOfMonth = LocalDate.of(year, month, 1);
        LocalDate firstDayOfNextMonth = firstDayOfMonth.plusMonths(1);

        //월요일=1(1%7=1), ... 일요일=7(7%7=0)
        int offsetWeekDays = firstDayOfMonth.getDayOfWeek().getValue() % 7;

        System.out.println("Su Mo Tu We Th Fr Sa ");

        for (int i = 0; i < offsetWeekDays; i++) {
            System.out.print("   ");
        }

        LocalDate dayIterator = firstDayOfMonth;
        while (dayIterator.isBefore(firstDayOfNextMonth)) {
            System.out.printf("%2d ", dayIterator.getDayOfMonth());
            if (dayIterator.getDayOfWeek() == DayOfWeek.SATURDAY) {
                System.out.println();
```

```
                }
                dayIterator = dayIterator.plusDays(1);
            }

        }
    }
```

정리

날짜와 시간의 주요 메서드 정리

날짜와 시간의 주요 메서드는 다음과 같다. 대략만 훑어보고 필요할 때 찾아보자.

- `LocalDateTime`의 주요 메서드
- `ZonedDateTime`의 주요 메서드
- `Instant`의 주요 메서드

LocalDateTime의 주요 메서드

생성 (Creation)

메서드	설명
`now()`	현재 시간대의 날짜와 시간을 가지는 `LocalDateTime` 인스턴스를 반환한다.
`of(int year, int month, int dayOfMonth, int hour, int minute)`	주어진 날짜와 시간으로 `LocalDateTime` 인스턴스 생성한다.
`of(int year, int month, int dayOfMonth, int hour, int minute, int second)`	초를 포함하여 주어진 날짜와 시간으로 `LocalDateTime` 인스턴스를 생성한다.
`of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond)`	나노초까지 포함하여 주어진 날짜와 시간으로 `LocalDateTime` 인스턴스를 생성한다.
`of(LocalDate date, LocalTime time)`	`LocalDate`와 `LocalTime`을 결합하여 `LocalDateTime` 인스턴스를 생성한다.

`from(TemporalAccessor temporal)`	다른 시간 객체로부터 `ZonedDateTime` 객체를 생성한다.

변환 (Conversion)

메서드	설명
`toLocalDate()`	`LocalDateTime` 인스턴스의 날짜 부분을 `LocalDate`로 변환한다.
`toLocalTime()`	`LocalDateTime` 인스턴스의 시간 부분을 `LocalTime`로 변환한다.
`toEpochSecond(ZoneOffset offset)`	주어진 시간대 오프셋을 사용하여 에포크 초로 변환한다.

시간대 관련 (Time-Zone)

메서드	설명
`atZone(ZoneId zone)`	지정된 시간대를 사용하여 `ZonedDateTime` 인스턴스를 생성한다.

조회 (Query)

메서드	설명
get(TemporalField field)	지정된 필드를 반환한다. 주로 `ChronoField`를 사용한다.
`getYear()`	연도를 반환한다.
`getMonth()`	월을 반환한다.
`getDayOfMonth()`	일(월의 몇 번째 날)을 반환한다.
`getDayOfWeek()`	요일을 반환한다.
`getHour()`	시간을 반환한다.
`getMinute()`	분을 반환한다.
`getSecond()`	초를 반환한다.
`getNano()`	나노초를 반환한다.

비교 (Comparison)

메서드	설명
`isBefore(LocalDateTime other)`	다른 `LocalDateTime` 보다 이전인지 비교한다.
`isAfter(LocalDateTime other)`	다른 `LocalDateTime` 보다 이후인지 비교한다.
`isEqual(LocalDateTime other)`	다른 `LocalDateTime` 과 같은지 비교한다. 시간을 기준으로 비교한다.

수정 (Adjustment)

메서드	설명
`with(TemporalField field, long newValue)`	지정된 필드를 새 값으로 변경한다.
`with(TemporalAdjuster adjuster)`	제공된 조정기를 사용하여 날짜를 조정한다.
`withYear(int year)`	연도를 수정한다.
`withMonth(int month)`	월을 수정한다.
`withDayOfMonth(int dayOfMonth)`	일(월의 몇 번째 날)을 수정한다.
`withHour(int hour)`	시간을 수정한다.
`withMinute(int minute)`	분을 수정한다.
`withSecond(int second)`	초를 수정한다.
`withNano(int nanoOfSecond)`	나노초를 수정한다.

추가 (Addition)

메서드	설명
`plus(long amountToAdd, TemporalUnit unit)`	지정된 시간 단위로 시간을 더한다.
`plus(TemporalAmount amountToAdd)`	주어진 시간만큼 더한다.
`plusYears(long years)`	연을 더한다.
`plusMonths(long months)`	월을 더한다.
`plusWeeks(long weeks)`	주를 더한다.

`plusDays(long days)`	일수를 더한다.
`plusHours(long hours)`	시간을 더한다.
`plusMinutes(long minutes)`	분을 더한다.
`plusSeconds(long seconds)`	초를 더한다.
`plusNanos(long nanos)`	나노초를 더한다.

- 동일하게 `minusXxx()`가 존재한다.

포맷팅 (Formatting)

메서드	설명
`format(DateTimeFormatter formatter)`	주어진 포매터를 사용하여 `LocalDateTime`을 문자열로 환한다.

ZonedDateTime의 주요 메서드

`LocalDateTime`과 중복되는 내용은 제외한다.

생성 및 변환

메서드	설명
`now()`	현재 시간대의 현재 날짜와 시간을 가진 `ZonedDateTime` 객체를 생성한다.
`now(ZoneId zone)`	지정된 시간대의 현재 날짜와 시간을 가진 `ZonedDateTime` 객체를 생성한다.
`of(LocalDate date, LocalTime time, ZoneId zone)`	주어진 날짜, 시간, 시간대로 `ZonedDateTime` 객체 생성한다.
`of(LocalDateTime dateTime, ZoneId zone)`	주어진 `LocalDateTime`과 시간대로 `ZonedDateTime` 객체를 생성한다.

`of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond, ZoneId zone)`	주어진 날짜, 시간, 나노초, 시간대로 `ZonedDateTi` 객체를 생성한다.

타임존 관리

메서드	설명
`withZoneSameInstant(ZoneId zone)`	다른 시간대로 시간을 변환하되, 절대 시간(UTC)을 유지한다.
`withZoneSameLocal(ZoneId zone)`	시간대를 변경하되, 로컬 날짜와 시간은 그대로 유지한다.
`withEarlierOffsetAtOverlap()`	겹치는 오프셋(여름 시간제 등)이 있을 때, 이전 오프셋을 사용한다.
`withLaterOffsetAtOverlap()`	겹치는 오프셋이 있을 때, 나중 오프셋을 사용한다.

조회

메서드	설명
`getOffset()`	현재 `ZonedDateTime`의 오프셋을 반환한다.
`getZone()`	현재 `ZonedDateTime`의 시간대를 반환한다.

기타 유용한 메서드

메서드	설명
`toLocalDateTime()`	`ZonedDateTime`을 `LocalDateTime`으로 변환한다.
`toInstant()`	`ZonedDateTime`을 `Instant`으로 변환한다.
`toEpochSecond()`	1970-01-01T00:00:00Z부터 현재 `ZonedDateTime`까지의 초 수를 반환한다.

Instant의 주요 메서드

`LocalDateTime`과 중복되는 내용은 제외한다.

생성(Creation)

메서드	설명
`now()`	현재 UTC 기준의 `Instant`를 반환한다.
`ofEpochMilli(long epochMilli)`	주어진 에폭시 시간(밀리초)을 기준으로 `Instant`를 생성한다.
`ofEpochSecond(long epochSecond)`	주어진 에폭시 시간(초)을 기준으로 `Instant`를 생성한다.
`ofEpochSecond(long epochSecond, long nanoAdjustment)`	주어진 에폭시 시간(초)과 나노초 조정값을 기준으로 `Instant`를 생성한다.

변환(Conversion)

메서드	설명
atOffset(ZoneOffset offset)	Instant를 OffsetDateTime으로 변환한다.
atZone(ZoneId zone)	Instant를 특정 시간대의 ZonedDateTime으로 변환한다.
toEpochMilli()	Instant의 에폭시 시간을 밀리초 단위로 반환한다.

조회

메서드	설명
getEpochSecond()	에폭시 시간을 초 단위로 반환한다.
getNano()	현재 초의 나노초 부분을 반환한다.