[읽을거리] 6. 선형탐색

BinarySearch 직접 구현하기

이진탐색 (Binary Search)를 직접 구현해 봅시다.

함수 시그니처 설계

함수명은 `binarySearch` 라고 해봅시다.

입력받을 인자로는 목록과 찾을 데이터를 받겠습니다.

그러면 이렇게 설계 되겠군요.

```
void binarySearch(List data, ? target)
```

데이터 타입이 필요하겠군요. 제너릭을 사용해서 어떤 데이터 타입이라도 처리가 가능하도록 해보겠습니다.

리턴값은 찾아낸 데이터를 반환하는 것으로 하겠습니다.

그러면 이렇게 변경되겠네요

```
<T> T binarySearch(List<T> data, T target)
```

그럼 이제 하나씩 구현해 봅시다.

내용구현

가운데에서 부터 범위를 좁혀가면서 데이터를 찾아낼 것이니까, 가운데 값을 찾아봅시다.

```
int min = 0;
int max = data.size();
int mid = (end - start) / 2;
```

mid 는 가운데 위치하는 값의 index 가 됩니다.

이 mid 의 위치에 해당하는 값을 꺼내오겠습니다.

```
T m = data.get(mid);
```

이것과 우리가 찾으려는 target 과 비교를 해봐야 겠죠. 비교는 대소관계를 비교해야 하니까 compareTo 를 사용할 겁니다. 그런데, T 타입에는 compareTo를 사용할 수 있는 정보가 없네요?

그래서 시그니처는 이렇게 변경될겁니다.

```
<T extends Comparable> T binarySearch(List<T> data, T target)
```

T 는 Comparable 을 상속/구현 하고 있는 것으로 제한시키겠습니다.

이제 우리는 m 과 target을 서로 (대소)비교할 수 있게 되었습니다.

```
int c = m.compareTo(target)
```

c 가 0 이라는 것은 우리가 찾으려는 바로 그 값인 거죠. 그러면 바로 return을 시켜도 좋겠습니다.

```
if(c == 0) return m;
```

target이 m 보다 오른쪽에 있다면, 즉 m < target 이라면 min ~ max 의 범위를 **mid** ~ max 로 변경하면 될겁니다.

m < target 이라는 것은 작은 값(m) 에서 큰 값(target) 을 빼는 경우가 되니까 음수(<0) 가 됩니다.

```
if(c < 0) min = mid + 1;
```

mid 위치에 있는 값은 이미 비교 했으니까 mid + 1 위치로 옮기겠습니다.

m > target 이라면 max를 mid 로 쪽으로 옮겨야 겠죠.

```
else max = mid;
```

반복하기

이러한 과정을 계속 반복해서 target 과 같은 값을 만나면 return 될 테니까 바로 종료될 것입니다.

근데 target값이 없다면 언제까지 반복해야 할까요? 우리는 min, max 를 가지고 계속 중간 값을 찾아나가고 있으니까 min ≥ max 가 되는 경우라면 더이상 반복할 필요도 없을 것입니다.

```
while(min < max) {
  ...
}
```

코드 구성 및 테스트

최종 코드는 이렇게 됩니다.

```
static <T extends Comparable> T binarySearch(List<T> data, T target) {
    int min = 0;
    int max = data.size();
    while(min < max) {
        int mid = (max - min) / 2;
        T m = data.get(mid);

        int c = m.compareTo(target);
        if(c == 0) return m;
        if(c < 0) min = mid + 1;
        else max = mid;
    }
    return null;
}
```

이 코드를 테스트 해봅시다.

```
void test() {
    List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    System.out.println(binarySearch(list, 4));
}
```

정상적으로 4를 찾아낸다면 화면에 4가 출력될 것입니다.

하지만, 실행해보신 분들은 프로그램이 종료되지 않고 값도 찾아내지 못한다는 것을 알 수 있을 것입니다.

버그 찾기

버그를 찾기 위해서는 IDE에서 제공되는 디버깅 기능을 사용해서 step 별로 값의 변화를 보면서 원하는대로 동작하는지를 확인해보면 알 수 있습니다.

```
▶ 초기상태
min : 0
max : 10

▶ 첫번째 루프
mid : 5
m   : 6
c   : 1
min : 0
max : 5

▶ 두번째 루프
mid : 2
m   : 3
c   : -1
min : 3
max : 5

▶ 세번째 루프
mid : 1   ⬅ 버그발견
m   : 2
...
```

세번째 루프에서 버그가 발견되었습니다. 두번째 루프에서 min ~ max 는 3 ~ 5 이니까 세번째 루프에서 mid 값은 4가 나와야 하지만, 1인 나왔습니다.

`(max - min) / 2` 만큼을 min 에 더해주어야 했던 것입니다.

버그수정

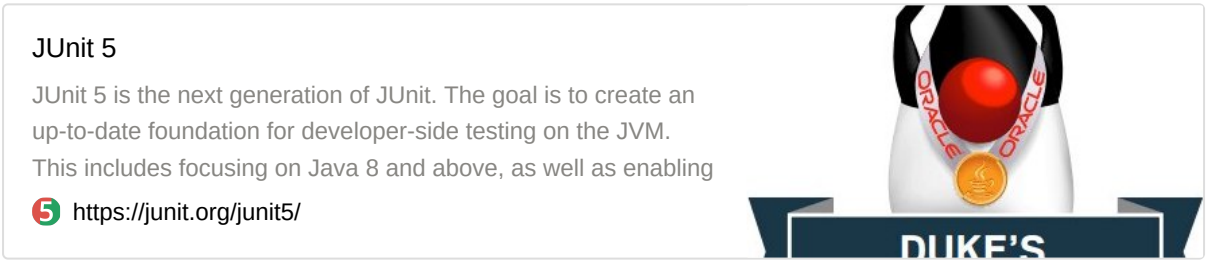
```
static <T extends Comparable> T binarySearch(List<T> data, T target) {
    int min = 0;
    int max = data.size();
    while(min < max) {
        int mid = (max - min) / 2 + min;  // 수정된 부분
        T m = data.get(mid);

        int c = m.compareTo(target);
        if(c == 0) return m;
        if(c < 0) min = mid + 1;
        else max = mid;
    }
    return null;
}
```

이제 테스트를 수행했을 때 원하는 값인 4를 찾아내는 결과를 얻을 수 있습니다.

유닛테스트

프로그램을 만들었다면, 정상동작하는지 항상 확인을 해야 합니다. 실제 프로그램을 동작시켜서 그 결과를 볼 수도 있지만, 로직의 일부 혹은 함수 하나만을 확인하기 위해서 테스트코드를 작성하여 그 동작을 검증하게 됩니다. 이것을 유닛테스트 라고 합니다.

유닛테스트를 할 수 있는 많은 라이브러리들이 있지만, 가장 많이 사용되는 JUnit을 사용해서 테스트 해보겠습니다.



JUnit 사이트에서 jar 라이브러리를 다운받아 프로젝트의 classpath에 넣고 수행해도 되고, gradle이나 maven 같은 패키지 매니저를 사용한다면 쉽게 의존성을 관리할 수 있습니다.

Gradle

```
implementation 'org.junit.jupiter:junit-jupiter:5.8.2'
```

Maven

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.8.2</version>
</dependency>
```

테스트 코드 작성

테스트 코드를 작성할 때는, 정상동작, 비정상동작, 예외상황 등의 여러가지 상황을 모두 확인할 수 있는 테스트 코드를 작성하는 것이 좋습니다.

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNull;

class SolutionTest {
```

```
    @Test
    @DisplayName("정상테스트")
    void binarySearch_ok() {
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        Integer found = new Solution().binarySearch(list, 4);
        assertEquals(4, found);  // 예상 결과 검증
    }

    @Test
    @DisplayName("비정상테스트")
    void binarySearch_nok() {
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        Integer found = new Solution().binarySearch(list, 40);
        assertNull(found);  // 예상 결과 검증
    }
}
```

항상 프로그램을 작성하고 나면 동작을 검증할 수 있는 테스트 코드를 작성하는 습관을 들이는 것이 필요합니다.

JUnit은 널리 사용되고 있고, 매우 많은 기능들이 제공되고 있기 때문에 따로 시간을 들여 학습해보실 것을 권장합니다.