Local-Variable Type Inference

sanghyuck.na@lge.com

1

Local-Variable Type Inference

- 지역변수의 초기화 선언 시 타입추론확장 개선안10-JEP 286, 11-JEP323
  - 개발자들은 보통 제네릭 문법적인 이유 때문에 Type선언이 길어지는 점
  - 좋은 변수 이름으로 충분히 코드에 의미부여를 할 수 있는 경우
  - C++은 지역변수타입추론 auto와 동일
- 지역변수의 LVT 대상
  - 초기화를 하는 지역변수
  - For-loop인덱스변수
  - Loop의 변수

Owner	Dan Smith
Type	Feature
Scope	SE
Status	Closed/Delivered
Release	10

2

사용법

- Reserved type name "var"
  - 기존 var명칭을 사용하는 오래된 코드는 호환성에 영향 받지 않습니다

```
var name = initializer;
```

```
var x = 10;
var lst = List.of(1, 2, 3);

for (var e : lst) {
    System.out.print(e + " ");
}

for (var i = 0; i < lst.size(); ++i) {
    var e = lst.get(i);
}
```

3

사용법

- 할 수 없는 것
  - 초기화가 없는 지역변수
  - 여러 변수를 한 선언에 동시에 하는 지역변수
  - 배열초기화 값으로 만드는 지역변수
  - ~~초기화되는 변수를 참조하는 지역변수~~

```
var x;
var g = null;
var x = 10, y = 20, z = 30;
var k = { 1, 2 };

var k = new int[] { 1, 2 };
var ok = k;
```

4

사용법 – in Lambda, Stream

- Lambda expression, Method reference에 사용불가
- Local-Variable Syntax[11-JEP323]
  - Lambda expression의 parameter type으로 사용가능

```
var f = () -> {
};
var m = this::test1;


var x = IntStream.of(1, 2);
x.forEach((var i) -> System.out.println(i));
Consumer<Integer> c = (var i) -> System.out.println(i);
```

5

Style Guidelines

- 기본 원칙(Principals)
  - P1. 가독성이 개발보다 중요하다
  - P2. 코드는 지역 범위 추론에서 명확 해야 한다
  - P3. 가독성은 IDE에 의존하면 안된다
  - P4. 현재 Concrete Type은 trade-off
- 지침(Guidelines)
  - G1. 유용한 정보를 주는 변수명 선택
  - G2. 지역 변수의 범위를 최소화
  - G3. 초기화 값이 Reader에게 충분한 정보를 제공한다면 var 사용
  - G4. 지역 변수의 중첩 or 연속된(Chained) 표현식을 분리하는데 사용
  - G5. 지역 변수의 "Interface 프로그래밍"에 너무 걱정하지 마라
  - G6. Generic 또는 Diamond 메소드를 가진 var를 사용할 경우 주의하라
  - G7. Literals에 var를 사용하는 경우 주의하라

https://goo.gl/DvFXtk Style Guidelines for Local Variable Type Inference in Java

6

G1 – 유용한 정보를 주는 변수 명 사용

- 의미있는 이름짓기
  - 타입은 완전히 가려지기 때문에 변수이름으로 의미를 이해할 수 있어야한다. 변수 이름짓기(naming)이 더 중요해지면서 코드에 자주 등장하는 단일문자 변수명은 되도록 지양한다

```
List<Integer> x = List.<Integer>of();

var lst = List.<Integer>of();
```

7

G2 – 지역변수의 범위 최소화

- 논리적 버그
  - LVT는 타입을 가린다는 점에 간결한 코드를 작성할 수 있지만, 코드가 길어질 수록 버그의 원인을 파악하기 어렵게 만든다. 아래는 논리오류가 있는 코드로 눈으로만 확인하는 경우 기존코드보다 버그 찾기가 어렵다

```
Integer MUST_BE_ADDED_LAST = Integer.valueOf(1);
var items = new HashSet<Integer>();

… code가 엄청나게 많았다 …

items.add(MUST_BE_ADDED_LAST);
for (var i : items) {
        // ??? 마지막에 추가되지 않았다..
}
```

8

G3 – Initializer는 충분한 정보를 제공해야 한다

- 선언 타입 일원화
  - 지역변수는 자주 생성자로 초기화 되지만, 생성되는 클래스이름과 변수이름은 지역변수 선언구문의 왼쪽에서 자주 반복되어 출현한다. 만약 타입이름이 긴 경우 var 사용으로 인한 변수이름은 의미 있는 정보를 제공한다

```
ByteArrayOutputStream outputStream = new ByteArrayOutputStream(…);
var outputStream = new ByteArrayOutputStream(…);


BufferedReader reader = Files.newBufferedReader(…);
var reader = Files.newBufferedReader(…);

List<String> stringList = List.of("a", "b", "c");
var stringList = List.of("a", "b", "c");
```

9

G4. 연결되거나 중첩된 표현식 분리

- 체인 명령문을 분리하여 가독성 높이기
  - 다음 코드는 자주 출현하는 단어를 찾는 코드다. 이 코드는 정확히 동작하지만, 3개 스트림 Pipeline으로 구성된 표현식임에도 단일 스트림과 같이 보여진다. 첫번째는 map으로 그룹핑하고, 다음 map를 reduce하고, key를 추출하여 화면에 출력한다
- var를 사용하여 그 의미단위로 분리하기
  - 성능도 고려할 부분이겠지만, 상황에 맞춰 분리하여 가독성을 높인다

```
List<String> newStringList() {return Arrays.asList(("It felt in the beginning like things " + "could get messier: hundreds of angry " + "students had gathered outside " + "the gate of the venue of the graduation " + "ceremony in south Delhi and were demanding a " + "meeting with the vice-chancellor who they alleged " + "had been avoiding them.").split(" "));}
----
newStringList().stream().collect(Collectors.groupingBy(s -> s,
Collectors.counting())).entrySet().stream().max(Map.Entry.comparing
ByValue()).map(Map.Entry::getKey).ifPresent(System.out::println);
```

```
var freqMap = newStringList().stream().collect(
    Collectors.groupingBy(s -> s, Collectors.counting()));
var maxEntryOpt = freqMap.entrySet().stream()
    .max(Map.Entry.comparingByValue());
maxEntryOpt.map(Map.Entry::getKey).ifPresent(System.out::println);
```

10

G5. 지역 변수의 "Interface 프로그래밍"에 대한 염려

- Interface 기반 프로그래밍
  - 일반 JAVA프로그래밍은 Concrete type의 인스턴스로 생성해서 Interface type 변수에 할당합니다. 이것은 code를 구현(implementation) 대신에 추상화(abstraction)에 묶습니다. 이점은 앞으로 코드의 유지보수에 좀더 유연하게 합니다.
- 지역적
  - 그 범위는 method, attribute, parameter 타입추론까지 영향을 미치지 않습니다. var에서도 interface programming은 여전히 중요합니다. 미래에 초기화가 바뀌어 추론타입도 바뀌어 컴파일 에러나 동적오류가 발생한다 할지라도, 그 파급되는 범위는 오로지 지역변수에 한합니다.
  - Static Factory method

```
List<Integer> list = new ArrayList<Integer>();

var arrayList = new ArrayList<Integer>();
arrayList.trimToSize();
arrayList.ensureCapacity(5);
```

11

G6. Diamond를 갖는 var 사용주의

- var와 "Diamond"사용은 분명한 타입정보 생략
  - 이미 오른쪽 부분에 존재하는 Generic type 정보에서 유도될 수 있습니다. 하지만 그 Generic type정보가 만약 생략되어 존재하지 않다면 의도와는 다르게 <Object>로 추론되어 사용을 조심해야 합니다
- var선언 문의 Right side의 Generic type은 충분히 타입정보를 모두 작성

```
PriorityQueue<String> que = new PriorityQueue<String>();
PriorityQueue< String > que = new PriorityQueue<>();
var que = new PriorityQueue<String>();
var que = new PriorityQueue<>();
Comparator<String> cmp = Integer::compare;
var que = new PriorityQueue<>(cmp);

var intList = List.of();
var intList = List.<Integer>of();
var bigList = List.of(BigInteger.ZERO);
```

12

G7. Literal 사용주의

- Primitive literals[JSR-334]
  - boolean, character, long, string literals로 type은 명확히 추론됨
- 초기화 정수숫자는 모두 int로 추론
  - Primitive literals이 없어도 기존은 왼쪽타입으로 묵시적으로 타입캐스팅 됐으나, var 사용으로 명확한 왼쪽타입이 생략되어 명시적캐스팅이 필요합니다

```
boolean ready = true;
char c = '\ufffd';
String s = "wombat";
```

```
var ready = true;
var c = '\ufffd';
var s = "wombat";
```

```
byte flags = 0;
short mask = 0x7fff;
long base = 17;
```

```
var flags = (byte)0;
var mask = (short)0x7fff;
var base = 17L;
```

```
float f = 2.0f;
double d = 2.0d;
```

```
var f = 2.0f;
var d = 2.0d;
```

13

도전 하세요!

- 다음 메소드에 적절한 부분을 LVT를 사용해보세요

```
void removeMatches(Map<? extends String
                  , ? extends Number> map, int max) {

    Iterator<? extends Map.Entry
        <? extends String, ? extends Number>>
        iterator = map.entrySet().iterator();

    while(iterator.hasNext())
    {
        Map.Entry<? extends String, ? extends Number>
           entry = iterator.next();
    }
}
```

14

도전 하세요!

- 다음 메소드에 적절한 부분을 LVT를 사용해보세요

```
String readLine(Socket socket, String cn) {

    try (InputStream is = socket.getInputStream();
        InputStreamReader isr = new InputStreamReader(is, cn);
        BufferedReader buf = new BufferedReader(isr)) {
        return buf.readLine();
    } catch (IOException e) {
    }
    return "";
}
```

15

정리

기본 원칙(Principals)
P1. 코드 읽기는 쓰기보다 중요하다
P2. 코드는 지역 범위 추론에서 더욱 명확히 해야 한다
P3. 코드 가독성은 IDE에 의존하면 안된다
P4. Concrete Type은 tradeoff 다.

지침(Guidelines)
G1. 되도록 유용한 정보를 주는 변수 이름 선택하기
G2. 되도록 지역 변수의 범위를 최소화하기
G3. 초기화 값이 Reader에게 충분한 정보를 제공한다면 var 사용하라
G4. 지역 변수의 중첩 or 연속된(Chained) 표현식을 분리하는데 사용하라
G5. 지역 변수의 "Interface 화 programming"에 너무 걱정하지 마라
G6. Generic 또는 Diamond 메소드를 가진 var를 사용할 경우 주의하라
G7. Literals에 var를 사용하는 경우 주의하라

16

Appendix: Underscore[7]

- 숫자 자릿수 구분표현
  - Numeric literals상에 숫자 간 그룹을 분리
- 사용불가 케이스
  - 숫자 맨 앞, 뒤에 인접위치
  - 실수 소수점에 인접위치
  - Literal F와 L 접미사의 바로 앞, 숫자

```
var creditCardNumber = 1234_5678_9012_3456L;
var socialSecurityNumber = 999_99_9999L;
var pi = 3.14_15F;
var hexBytes = 0xFF_EC_DE_5E;
var hexWords = 0xCAFE_BABE;
var maxLong = 0x7fff_ffff_ffff_ffffL;
var nybbles = 0b0010_0101;
var bytes = 0b11010010_01101001_10010100_10010010;
```

17