자바 데이터 타입, 변수 그리고 배열

자바의 프리미티브 타입, 변수 그리고 배열을 사용하는 방법을 익힙니다.

프리미티브 타입 종류와 값의 범위 그리고 기본 값

- 우리가 주로 사용하는 값의 종류는 크게 문자와 숫자로 나눌 수 있으며 여기서 숫자는 다시 정수와 실수로 나뉜다.
- 기본형은 모두 8가지의 타입(자료형)이 있으며, 크게 논리형, 문자형, 정수형, 실수형으로 구분된다.

타입 종류

- 정수형은 가장 많이 사용되기에 타입이 4가지나 제공된다.
- 각 타입별로 범위가 다르기에 범위에 맞는 값을 사용하면 된다.

타입 범위

- boolean은 true와 false 두 값만 표현하면 되기에 1바이트면 충분하다.
  - 기본 값 : `false`
- char는 자바에서 유니코드(2 byte문자 체계)를 사용하기에 2byte
  - 기본 값 : `\u0000`
- byte는 크기가 1byte이므로 byte.
  - 기본 값 : `0`
- int(4 byte)를 기준으로 짧게는 (2 byte) 길게는 (8 byte)를 취사선택한다.
  - 기본 값 : `0`
- float은 실수값을 부동소수점(floating-point)방식으로 저장하기 때문에 float
  - 기본 값 : `0.0F`
- double은 float보다 두 배의 크기(8byte)를 갖기 때문에 double
  - 기본 값 : `0.0`

실수형의 정밀도

- 실수형은 정수형과 저장 방식이 다르기에 같은 크기라도 훨씬 큰 값을 표현할 수는 있지만, 오차가 발생할 수 있다. 그래서 정밀도(precision)가 중요한데, 정밀도가 높을수록 오차의 범위가 줄어든다.
- 위 표를 보면 `float`의 정밀도는 7자리로 10진수로 7자리의 수를 오차없이 저장할 수 있다는 의미다. 그렇기에 사용할 변수의 값의 범위가 7자리를 넘는다면 정밀도를 고려해 `double` 타입을 사용해야 한다.

프리미티브 타입과 레퍼런스 타입

자료형은 크게 '기본형(Primitive Type)' 과 참조형(Reference Type)으로 나눌 수 있다.

- 기본형(Primitive Type) : 논리형(boolean), 문자형(char), 정수형(byte, short, int, long), 실수형(float, double) 계산을 위한 실제 값을 저장한다.
- 참조형(reference type) : 객체의 주소를 저장한다. 기본적으로 `Java.lang.Object`를 상속받을 경우 참조형이 된다. 즉, 기본형을 제외하고는 참조형이라 생각해도 된다.

좀 더 얘기하자면 기본형은 메모리영역의 스택영역에 실제 값들이 저장된다면, 참조형은 실제 인스턴스는 힙영역에 생성되있고, 그 영역의 주소를 스택영역에서 저장하고 있다고 보면 된다.

리터럴

> 그 자체로 값을 의미하는 것

- 리터럴은 데이터 그 자체를 의미한다.
- 아래 그림에서 `2020`이 리터럴이다.

- 즉, 2020, 123, 3.14, “ABC” 와 같은 값들을 리터럴이라고 하는데 본래 이러한 값들은 상수라 불러야 하지만 프로그래밍에서는 상수를 **'값을 한 번 저장하면 변경할 수 없는 저장공간'**으로 정의했기 때문에 이와 구분하기 위해서 리터럴이라는 용어를 사용한다.
- 그러니 리터럴은 기존에 알고있던 상수의 다른 이름이라고 볼 수 있다.

> 인스턴스는 리터럴이 될 수 있을까?

- 인스턴스안의 값의 불변성`(Imutable)` 이 보장된다면 객체 리터럴이 될 수 있다.(불변 클래스`(imutable class)`)
- 하지만 이렇게 불변성을 보장하도록 설계된 클래스를 제외하고 보통의 인스턴스는 동적으로 사용되고 내용이 변할 수 있기 때문에 객체 리터럴이 될 수 없다.
- Ex: `Java.lang.String` 이나 `java.awt.Color` 같은 클래스는 내용이 변해야 하는 상황이면 새로운 객체를 만들어 내용의 불변성이 보장되기에 객체 리터럴이라 부른다.

변수 선언 및 초기화하는 방법

변수 선언

변수를 사용하기위해서는 우선 변수를 선언해야 하며 아래 그림과 같이 선언합니다.

- **변수 타입** : 변수에 저장될 값이 어떤 타입(type)인지 지정하는 것.
- **변수 이름** : 변수에 붙힌 이름. 변수가 *값을 저장할 수 있는 메모리 공간*을 의미하므로 변수 이름은 이 메모리 공간에 이름을 붙혀주는 것.

이렇게 변수를 선언하면, 메모리의 빈 공간에 '변수타입'에 알맞은 크기의 저장공간이 확보되고, 변수 이름을 붙혀서 이 이름을 통해 해당 저장공간을 사용할 수 있게 된다.

변수 초기화

> 변수를 사용하기 전 처음으로 값을 저장하는 것

변수를 선언하면 메모리에 변수의 저장공간이 확보되어있지만, 여러 프로그램에 의해 공유되기 때문에 이 공간안에 어떠한 값이 저장되어있을지는 알 수 없다.

그렇기에 초기화(`initialization`)를 해줘야 한다.

- 변수에 값을 저장할 때는 대입 연산자 =을 사용한다. 위 그림을 보면 `year`라는 `int` 변수타입과 `year`라는 변수 이름을 가진 변수에게 `2020`이라는 값을 대입한다.
- 즉, 대입연산자의 우측의 있는 값을 좌측에 있는 변수에 저장한다.
- 변수의 종류에 따라 변수의 초기화를 생략할 수 있는 경우도 있지만, 변수는 사용되기 전에 적절한 값으로 초기화 하는 것이 좋다.

지역변수는 사용하기 전 반드시 초기화 해야 한다.

그밖의 초기화의 종류

지역변수는 변수의 초기화로 충분하지만, 멤버변수의 초기화는 몇가지 방법이 더 있다.

1. 명시적 초기화(explicit initialization)

: 변수 선언과 동시에 초기화 하는 것을 명시적 초기화라 하는데, 위에서 소개한 변수의 초기화와 동일하며, 클래스 및 지역변수 어디서든 사용가능하며 여러 초기화 방법중 최우선적으로 고려한다.

2. 초기화 블럭(initialization block)

: 초기화 블럭은 클래스 초기화 블럭과 인스턴스 초기화 블럭으로 나뉜다.

```
class ExplicitInitialization {
  static {
  /*클래스 초기화 블럭 */
}
{
  /*인스턴스 초기화 블럭*/
}
}
```

- 클래스 초기화 블럭 : 클래스변수의 복잡한 초기화에 사용. 블럭내에서는 로직도 추가할 수 있기 때문에 명시적 초기화만으로 부족할 때 사용한다.
- 인스턴스 초기화 블럭 : 인스턴스 변수의 복잡한 초기화에 사용. 모든 생성자가 공통으로 수행해야 하는 로직이 있을 때 사용한다.

3. 생성자(constructor)

: 생성자는 말 그대로 인스턴스 생성시에 생성자 함수 안에서 명시적 초기화가 이뤄진다.

변수의 스코프와 라이프타임

스코프는 한글로 풀어보자면 범위이다. 즉, 변수의 스코프는 변수의 범위라는건데 이 범위는 키워드와 선언된 블럭위치에 따라서 달라진다.

선언위치에 따른 변수의 종류

```
class A {
  int instanceValue; //인스턴스 변수
  static int classValue;//클래스 변수(static, 공유 변수)
  void method(){
    int localValue = 0; //지역 변수
  }
}
```

클래스 내부에 선언되는 변수를 멤버변수라 한다. 여기서 static키워드가 붙은 변수를 클래스 변수, static 키워드가 없는 변수를 인스턴스 변수라 한다. 그리고 메서드 내부에 있는 localValue는 지역변수이다. 이 셋 모두의 범위와 생성시기는 다르다.

1. 변수의 종류와 특징

1) 인스턴스 변수(instance variable)

- 클래스 영역에 선언되며, 클래스의 인스턴스를 생성할 때 만들어진다. 그렇기에 인스턴스 변수 값을 읽어오거나 저장하기 위해서는 먼저 인스턴스를 생성해야 한다.
- 인스턴스 별로 별도의 저장공간을 확보하기에 인스턴스별 다른 값을 가질 수 있다.
- Ex: 아이스크림이라는 클래스의 돼지바 라는 인스턴스는 1000원이라는 가격과 쿠키라는 속성을 가진다.

2) 클래스 변수(class variable)

- 멤버변수에 `static` 키워드를 붙힐 경우 클래스 변수가 되며 한 클래스의 모든 인스턴스가 값을 공유한다.
- 클래스 변수는 인스턴스를 생성하지 않고 클래스가 메모리에 올라갔을때 선언되기 때문에 인스턴스에서는 언제든 바로 접근해서 사용할 수 있다.
- 그렇기에 어디서나 접근 할 수 있는 전역변수(global variable)의 성격을 가진다.

```
class LottoTicket {
  public static final LOTTO_PRICE = 1000;
  ...
}
public static void main(String[] args) {
  //LottoPrice: 1000
  System.out.println("LottoPrice: "+ LottoTicket.LOTTO_PRICE);
}
```

3) 지역 변수(local variable)

- 메소드 내에 선언되어 메소드 내에서만 사용 가능하며 메소드 종료와 함께 소멸된다.
- for문이나 while문같은 반복문도 동일하게 블럭내에서 선언된 지역변수는, 블럭을 벗어나면 소멸된다.

```
public static void main(String[] args) {
      for (int i = 0; i < 10; i++) {
        System.out.println("i = " + i);
    }
    System.out.println("i = " + i);//Checked Exception 발생
}
```

초기화 시기와 순서(라이프타임)

1. 초기화 시점

⇒ 프로그램 실행도중 클래스에 대한 정보가 요구될 때 클래스는 메모리에 로딩된다.
(만약 이미 메모리에 로딩되어 있다면 또다시 로딩하지는 않는다.)

2. 초기화 순서

3. 예제

```
class InitTest {
  static int classValue = 1;
  int instanceValue = 1;
  static {
      classValue = 2;
  }
  InitTest() {
      instanceValue = 3;
  }
}
```

- 클래스변수 초기화(1~3): 클래스가 처음 메모리에 로딩될 때 차례대로 수행된다.
- 인스턴스변수 초기화(4~7): 인스턴스를 생성할 때 차례대로 수행한다.
- **클래스 변수는 항상 인스턴스 변수보다 먼저 생성및 초기화된다.**

타입 변환, 캐스팅 그리고 타입 프로모션

타입 변환

> 변수 또는 상수의 타입을 다른 타입으로 변환하는 것

프로그램을 작성하다보면 서로 다른 타입간의 연산을 수행해야 하는 경우가 있다. 이럴 때 연산을 수행하기 전 서로의 타입을 일치시켜야하는데, 이렇게 변수나 리터럴의 타입을 다른 타입으로 변환하는 것을 형변환이라 한다.

형변환 방법

```
 (type)operand
```

- 변환할 변수나 리터럴 앞에 타입을 괄호와 함께 붙혀주기만 하면 된다. 이 때 형변환 연산자는 그저 피연산자의 값을 읽어서 지정된 타입으로 형변환하고 그 결과를 반환할 뿐이기에 기존의 변수

나 리터럴이 변화되지는 않는다.

```
double value = 123.456;
int score = (int)value;
System.out.println(value == 123.456); //true
```

- 기본형(primitive type) 변수는 boolean을 제외한 나머지 타입은 서로 형변환이 가능하다.
- 하지만, 타입간에는 각각이 가지는 범위(크기)가 다르기 때문에 형변환을 통해 크기의 차이만큼 값이 잘려나감으로써 값 손실(loss of data)이 발생할 수 있다.

자동 형변환

- 경우에 따라 형변환을 생략할 수 있다. 그래도 컴파일러가 생략된 형변환을 자동으로 추가한다.
- 하지만, 저장될 변수 타입의 범위가 더 작은 경우 에러가 발생하는데 이는 더 작은 값으로 할당되며 값 손실이 발생할 수 있기 때문이며 이를 이미 알고 명시적으로 형변환을 작성해주면 에러를 발생시키지 않는다.

```
byte b = 10000; //에러 발생. byte의 범위는 -128~127이다.
byte c = (byte)10000; //명시적 형변환으로 에러가 발생하지 않는다.
```

자동 형변환 규칙

> 기존의 값을 최대한 보존할 수 있는 타입으로 자동 형변환한다.

- 표현범위가 좁은 타입에서 넓은 타입으로 형변환 할 때 값 손실이 없기에 두 타입 중 표현범위가 더 넓은 쪽으로 형변환이 된다.

기본형의 자동 형변환의 방향

1차 및 2차 배열 선언하기

배열의 선언과 생성

1. 배열의 선언

- 배열의 선언은 타입 뒤 혹은 변수이름뒤에 대괄호([])를 붙히면 된다.

2. 배열의 생성

```
타입[] 변수이름 = new 타입[길이];
```

- new연산자를 사용해 배열의 타입과 길이를 지정하면 메모리에 해당 길이만큼 영역을 확보한다.
- 이때 배열의 길이는 최초 생성시 선언한 길이에서 정적이므로 길이의 변경을 원할 때는 새로운 배열을 생성해줘야 한다.

1차원 배열의 선언 & 2차원 배열의 선언

- 배열은 1차원 뿐 아니라 다차원 배열도 선언해서 사용할 수 있다.
- 다차원의 차원간 선언방법의 차이는 대괄호(`[]`)의 갯수 차이이다. 대괄호가 1개이면 1차원, 2개이면 2차원으로 메모리가 허용하는 한도까지 차원을 높힐 수 있다.

1. 1차원 배열의 선언

```
int[] oneDimensionalArray = new int[5]{1, 2, 3, 4, 5};
```

메모리에 올라간 1차원 배열

2. 2차원 배열의 선언

```
int[][] twoDimensionalArray = new int[2][2]{{1, 2}, {3, 4}};
```

메모리에 올라간 2차원 배열

- 2차원 배열은 각각의 대괄호를 행/열로 보면 좀 더 알기 쉽다 예를들어 `new int[행][열]`로 본다고 할 때 위에 코드인 `twoDimensionalArray` 는 2행 2열짜리 2차원 배열인 것이다.

타입 추론, var

타입 추론

- 자바 컴파일러에서 타입을 추론하는 것을 `Type Inference`이라 한다.
- 이 타입추론을 하기 위해 메소드 호출과 호출할 때 사용하는 인수(혹은 인수들)을 결정하기위한 선언부를 살펴본다.
- 추론 알고리즘(inference algorithm)을 통해 인수 타입을 결정하고 가능하다면 결과가 할당되는 타입(인수타입)이나 반환 타입도 추론한다.

```
static <T> T pick(T a1, T a2) { return a2; }
public static void main(String[] args) {
      Serializable d = pick("d", new ArrayList<String());
}
```

- 추론 알고리즘은 모든 인자와 어울리는 선에서 가장 구체적인 타입을 찾는데, 위 코드를 보면 pick 메소드의 매개변수는 T이고 메소드의 매개변수 a1, a2 둘 다 T 타입이다.
- 하지만 pick 메소드를 호출하면서 첫 번째 인자로 String을 주고 두 번째 인자로 ArrayList를 주었다. 이런 경우 모든 인자에 어울리는 선(공통 부모)란 Serializable으로 String과 ArrayList 둘 다

Serializable을 구현하고 있기 때문이다.

타입추론과 Generic Methods

타입추론덕분에 generic 메소드를 사용할 때 보통의 메소드처럼 특정 타입을 명시하지 않은 채로 호출할 수 있다. public class BoxDemo {

```
    public static <U> void addBox(U u, java.util.List<Box<U>> boxes) {
            Box<U> box = new Box<>();
            box.set(u);
            boxes.add(box);
    }
```

} public static <U> void outputBoxes(java.util.List<Box<U» boxes) {

```
  int counter = 0;
  for (Box<U> box: boxes) {
    U boxContents = box.get();
    System.out.println("Box #" + counter + " contains [" +
           boxContents.toString() + "]");
    counter++;
  }
}
```

```
public static void main(String[] args) {
  java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
    new java.util.ArrayList<>();
  BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes); //---(1)
  BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);//---(2)
  BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
  BoxDemo.outputBoxes(listOfIntegerBoxes);
}
```

} Java (1): addBox라는 generic 메소드를 호출할 때 <Integer> type witness와 함께 type parameter를 명시하여 사용할 수 있습니다. (2): Java 컴파일러가 메소드의 인자로부터 자동으로 Integer type임을 추론해주기 때문에 type witness를 생략할 수도 있습니다. 타입추론과 Generic 클래스의 인스턴스 Java 컴파일러가 컨텍스트로부터 타입추론이 가능하다면 Generic 클래스의 생성자를 호출하기 위해 필요한 type arguments를 비어있는 type witness(<>, the diamond)로 대체할 수 있다. List<String> myList1 = new ArrayList<String>(); List<String> myList2 = new ArrayList<>(); Java 타입추론과 Generic 생성자 클래스가 Generic/non-generic 인지 그 여부와 관계없이 생성자는 generic일 수 있다. class MyClass<X> {

```
<T> MyClass(T t) {
  // ...
}
```

}

public static void main(String[] args) {

```
MyClass<Integer> myInstance = new MyClass<Integer>("");
```

} Java ⇒ MyClass의 type 매개변수 X에는 Integer가 들어가지만 생성자의 type 매개변수 T에는 String이 들어간다. Java SE 7 이전에는 컴파일러에서 실제 type argument를 작성해 타입 추론을 할 수 있었지만, Java SE 7이후로 컴파일러는 the diamond(<>)를 사용하는 경우 다음과 같이 generic 클래스의 실제 type argument까지 추론이 가능하다. MyClass<Integer> myInstance = new MyClass<>(“”); Java Target Types Java 컴파일러는 generic method invocation의 type argument를 추론하기 위해 target typing의 이점을 이용한다. 표현식의 target type이란 표현식이 나타낸 위치에 의존하여(컨텍스트 의존) Java 컴파일러가 기대하는 데이터 타입이다 static <T> List<T> emptyList() { return new ArrayList<>(); } List<String> listOne = Collections.emptyList(); Java 위 코드는 Collection API의 emptyList 함수를 이용해 List<String> 객체를 반환한다. 이런 데이터 타입을 Target Type이라 하는데, emptyList 함수가 List<T> 타입을 리턴하기에 컴파일러에서 type argument T가 반드시 String일 것이라고 추론한다. 물론, type witness를 사용해 명시적 선언을 해줄 수도 있지만 위 코드에서는 불필요하다. List<String> listOne = Collections.<String>emptyList(); *불필요한 witness Java 하지만, Type Witness 가 필요한 경우도 있다. void processStringList(List<String> stringList) {* process stringList } public static void main(String[] args) {

```
    processStringList(Colections.emptyList());
```

} Java ⇒ 위 코드에서 메소드 호출은 정상적으로 동작할까? Java SE 7 컴파일러에서는 컴파일 되지 않고 에러가 발생하며 아래와 같은 에러 메세지가 출력된다. List<Object> cannot be converted to List<String> 이런 에러가 발생하는 이유는 컴파일러는 type argument T를 위한 value를 필요한데, 아무것도 주어지지 않았기에 Object를 value로 삼게된다. 그 결과 Collections.emptyList()는 List<Object> 객체를 리턴하며 이는 processStringList에서 호환하지않는 인수타입이기에 에러가 발생한다. 그렇기에 Java SE 7에서는 type witness를 명시해줘야 한다. processStringList(Colections.<String>emptyList()); Java 하지만, Java SE 8 부터는 위와 같은 경우에도 type witness를 명시해주지 않아도 Target type을 결정할 때 메소드의 argument도 살피도록 확장되었기 때문에 에러가 발생하지 않는다. 그렇기 때문에 Java SE 8이상에서는 위의 type witness가 없는 메소드 호출도 정상적으로 동작할 것이다. var Java 10부터 추가된 특징중 하나인 Local Variable Type Inference은 로컬 변수 선언을 var를 이용하여 기존의 엄격한 타입 선언방식에서 컴파일러에게 타입추론 책임을 위임할 수 있게 되었다. var list = new ArrayList<String>(); *infers ArrayList<String> var stream = list.stream();*infers Stream<String> Java Local Variable Type Inference 사용 조건 • 초기화된 로컬 변수 선언시 • 반복문에서 지역변수 선언시(enhanced for loop포함) var 활용 1. 지역변수 선언 var numbers = Arrays.asList(1, 2, 3, 4, 5);

for (var i = 0; i < numbers.size(); i++) {

```
  System.out.println("numbers = " + numbers.get(i));
```

} Java 2. forEach var numbers = Arrays.asList(1, 2, 3, 4, 5);

for (var number : numbers) {

```
  System.out.println(number);
```

} Java ⇒ 기존에는 Object타입으로 받아서 형변환을 하거나 IDE가 아닌 개발자가 직접 타입추론해 명시적 타입선언을 해줬는데 var를 사용하여 훨씬 편하게 타입선언이 가능해진다. 3. Lambda (Java 11) IntBinaryOperator plus10 = (@NonNull var one, @NonNull var two) → one + two + 10; Java ⇒ Java 11부터는 람다 인자에도 var사용이 가능해졌는데, 이를 통해 파라미터 어노테이션 사용까지 가능해졌다. 참고: 비어있는 type witness를 사용하면 Object로 추론한다.

Ref

https://catsbi.oopy.io/6541026f-1e19-4117-8fef-aea145e4fc1b
https://github.com/whiteship/live-study/issues/2

와프

From:
http://rwiki.repia.com/ - **문제를 잘 정의하는 것은 문제를 절반 해결한 것이다. - 2023.12**

Permanent link:
http://rwiki.repia.com/doku.php?id=wiki:java:java-lecture:2week&rev=1610926526

Last update: **2022/03/10 19:52**