2주차 과제: 자바 데이터 타입, 변수 그리고 배열

https://github.com/whiteship/live-study/issues/2

- 프리미티브 타입과 레퍼런스 타입

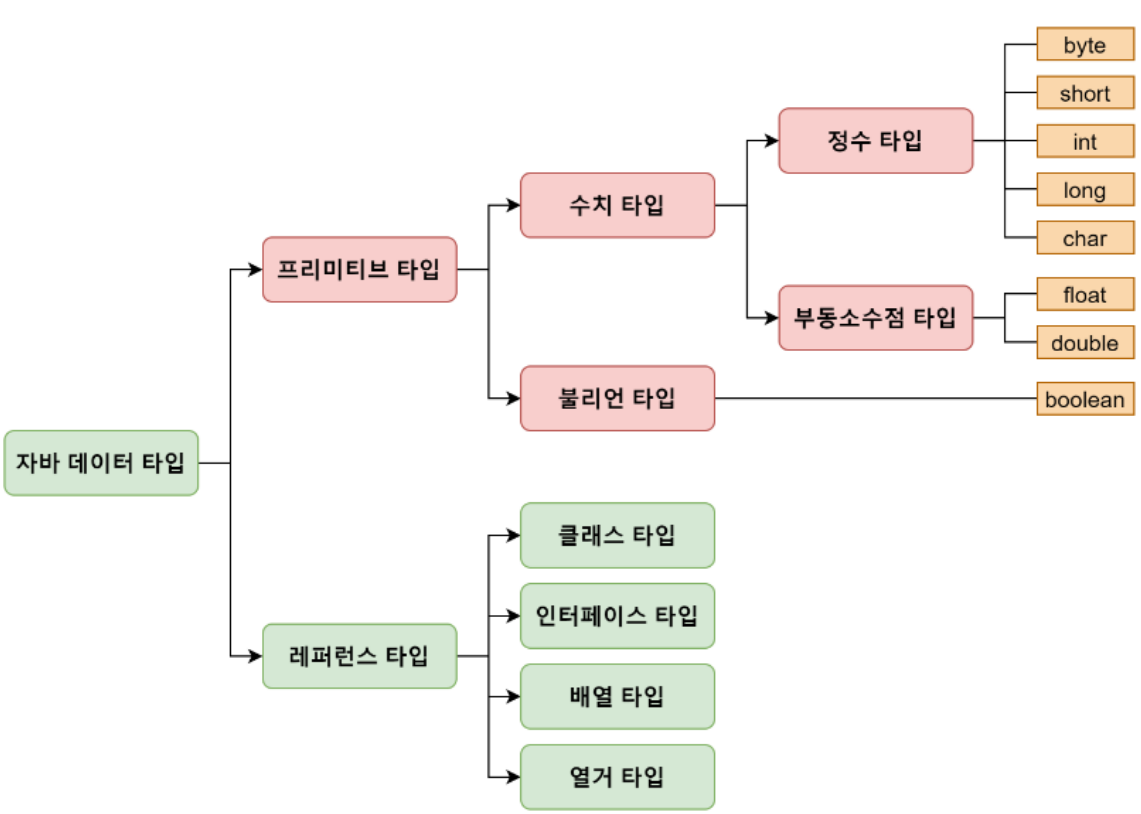
Java의 데이터 타입은 Primitive type, Reference 타입이 있다.

프리미티브 타입 = 정수, 실수, 문자, 논리 리터럴 등의 실제 데이터 값을 저장하는 타입

레퍼런스 타입 = 객체의 번지를 참조(주소를 저장)하는 타입으로 메모리 번지 값을 통해 객체를 참조하는 타입.

음 잘 모르겠다

예시를 보자.



이미지 출처: https://antstudy.tistory.com/187

그러니까 데이터 타입이 두 개로 나뉘는데 그래도 예시를 보니 대충은 감이 온다.

우선 타입이란 프로그램 안에서 취급하는 데이터 형식을 정의한 것이다.

```
String s = "JAVA STUDY";
Int i = 1234;
String[] arr = {"1", "2", "3"}
```

라고 했을 때, s라는 데이터의 형식을 String으로 정의하고, b라는 데이터의 형식을 Int로, arr라는 데이터의 형식을 Stringp[]로 정의하였다고 볼 수 있다. 이 때, String 과 Int, String[] 는 타입이 된다고 한다.

이런 타입은 Java에서 두 가지로 분류할 수 있는데, 프리미티브 타입과 레퍼런스 타입으로 나뉜다.

1. 프리미티브 타입

- 프리미티브 타입은 기본형 타입, 원시 타입이라고도 불린다.
- 자바 언어에 내장되어 있으며, 미리 정의하여 제공한다.
- 기본 값이 있기 때문에 Null이 존재하지 않는다.
- Null 을 넣고 싶다면 래퍼 클래스 (Wrapper Class)를 활용해야함.

'기본값이 있기 때문에 null 이 존재하지 않는다' 를 '비객체 타입이기 때문에 null 값을 가질 수 없다'고도 말하던데 결국 같은 말이다. 이 부분은 레퍼런스 타입의 기본값과 관련있는데 뒤에서 한 번 더 다뤄보자.

아무튼 프리미티브 타입은 기본값이 지정되어있기 때문에 null 을 값으로 넣으면 컴파일 에러가 난다.

아래 표는 프리미티브 타입의 기본값 과 표현 범위이다.

	타입	할당되는 메모리 크기	기본값	표현 범위
문자형	char	2byte	"＼u000"	0 ~ 65,535
정수형	byte	1byte	0	-128 ~ 127
	short	2byte	0	-32,768 ~ 32,767
	int	4byte	0	-2,147,483,648 ~ 2,147,483,647
	long	8byte	0L	-9,223,372,036,854,775,808 ~

				9,223,372,036,854,775,807
실수형	float	4byte	0.0F	(3.4 X 10-38) ~ (3.4 X 1038) 의 근사값
	double	8byte	0.0	(1.7 X 10-308) ~ (1.7 X 10308) 의 근사값
논리형	boolean	1byte	false	true, false

출처: https://yeastriver.tistory.com/7

1.1. char

- java 에서 유일하게 제공되는 unsigned 형태.

  ex) 다른 타입들은 맨 앞의 비트를 음수, 양수를 표현하기 위해 사용하지만 char은 그렇지 않다. 그래서 같은 2 byte 라도 char 의 범위는 0~65525 인 반면 short 는 그 절반인 -32,768 ~ 32,767 를 범위로 갖는다.

- char 끼리 대소 비교가 가능한 이유는 Unicode 정수 형태로 저장이 되기 때문이다.
- java 의 경우 Unicode 를 사용한다. 동양 글자의 경우 2 byte 가 필요하기 때문에 2byte 를 사용한다.

1.2. int

- JVM 의 피연산자 스택이 피연산자를 4 byte 단위로 저장하기 때문에 int 보다 작은 자료형의 값을 계산 시, int 형으로 형변환되어 연산이 수행된다.
- 정수형 데이터를 사용하게 되면 JVM에서 기본적으로 int 형 데이터 타입의 데이터로 인식해주게 된다.
- int형 데이터 타입의 범위를 넘어서는 long 타입의 정수를 사용하고자 하는 경우에는, 정수 데이터 맨 뒤쪽에 접미사 'l' or 'L' 을 붙여줘야 한다.

1.3. double

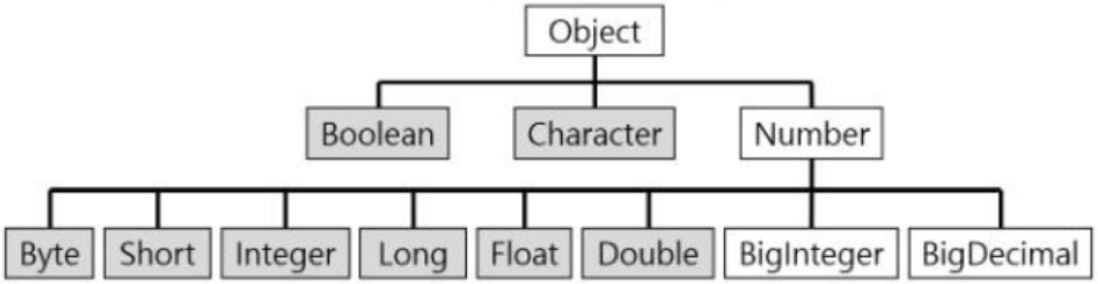
- 실수형 데이터 타입에서 기본 데이터 타입.
- double 형 데이터 타입의 범위를 넘어서는 float 형 데이터 타입의 실수형 데이터를 사용하고자 하는 경우에도 실수 데이터 맨 뒤쪽에 접미사 'f' 나 'F' 를 붙여줘야 한다.

1.4. boolean

- true, false 는 1bit 만 있어도 되는 게 아닌가? 하지만 cpu 가 1byte 보다 작은 데이터를 처리할 수 없기 때문이다. 즉, 주소를 매길 수 있는 최소한의 단위가 byte 라는 것이다.
  c 언어에서의 bool 타입도 똑같이 1byte 를 할당받는다.

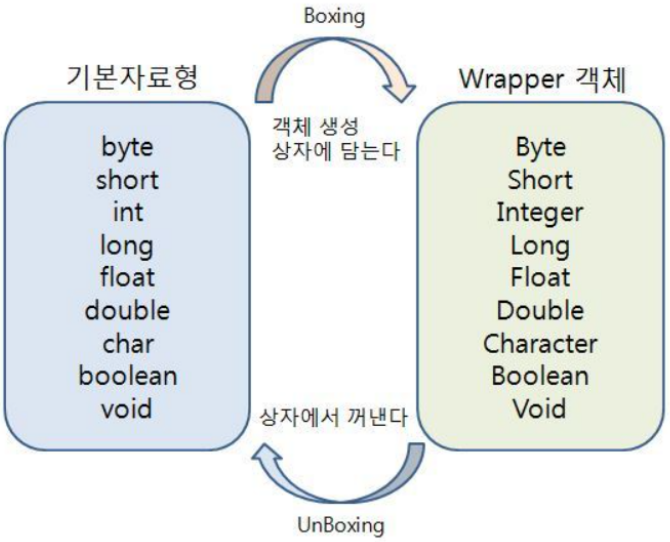
이렇게 기본값이 지정되어 있는 프리미티브 타입에 null 을 넣기 위해서, 즉 객체로 다루기 위해서

Wrapper Class 를 사용해야한다고 하는데, Wrapper Class 란 각 primitive type 을 클래스로 만든 것이다.



이미지 출처: https://develop-im.tistory.com/m/81

위 계층구조를 보면 알 수 있듯이 모든 래퍼 클래스의 부모는 Object 이고, 최종 클래스로 정의된다.



이미지 출처: https://yunanp.tistory.com/18

래퍼 클래스를 사용하는 방법은 래퍼 클래스의 객체가 자신과 대응하는 기본 자료형의 데이터를 파라미터로 전달받는 형식이다. 프리미티브 타입을 래퍼 클래스로 변환하는 과정을 박싱, 래퍼 클래스를 프리미티브 타입으로 변환하는 과정을 언박싱이라고 한다.

```
int num = 1;
Integer wrapper = new Integer(num); // Boxing
int n = wrapper.intValue(); // Unboxing
```

이렇게 직접 박싱과 언박싱을 하지 않아도 자동으로 해줄 수도 있다.

```
Integer num = 10; // 자동 박싱
int n = num; // 자동 언박싱
```

래퍼 클래스를 사용하는 이유는 아래와 같다.

1. 기본 데이터 타입을 Object로 변환할 수 있다.
2. java.util 패키지의 클래스는 객체만 처리하므로 Wrapper class는 이 경우에 도움이 된다.
3. ArrayList 등과 같은 Collection Framework의 데이터 구조는 기본 타입이 아닌 객체만 저장하게 되고, Wrapper class를 사용하여 자동박싱/언박싱이 일어난다.
4. 멀티스레딩에서 동기화를 지원하려면 객체가 필요하다.

---

2. 레퍼런스 타입

- 기본적으로 프리미티브 타입을 제외한 모든 타입들이 레퍼런스 타입.

  ex) 객체(Object), 배열(Array) 등

- java.lang.Object 를 상속받으면 레퍼런스 타입이 된다.
- 값이 저장되어 있는 곳의 주소값을 저장하는 공간으로 heap 메모리에 저장된다.

레퍼런스 타입은 기본값으로 null 을 가지는데, 레퍼런스 타입은 모두 `java.lang.Object` 를 상속하기 때문에 객체 타입이 된다.

레퍼런스 타입은 메모리 공간을 할당 받고 그 주소값을 반환해주기 때문에 메모리 공간을 할당받지 않은 상태라면 null 값을 가질 수 있다. null 값이 갖는 의미는 다음과 같다.

```
int num; // 초기값인 0을 갖는다.
Student person; // 처음 선언될 때 아무것도 참조하고 있지 않기 때문에 null 값을 갖는다.
person = new Student(); // new로 초기화를 하면 Student 의 실제 객체를 담고 있는 주소 x 값이 person 에 담겨있다.
```

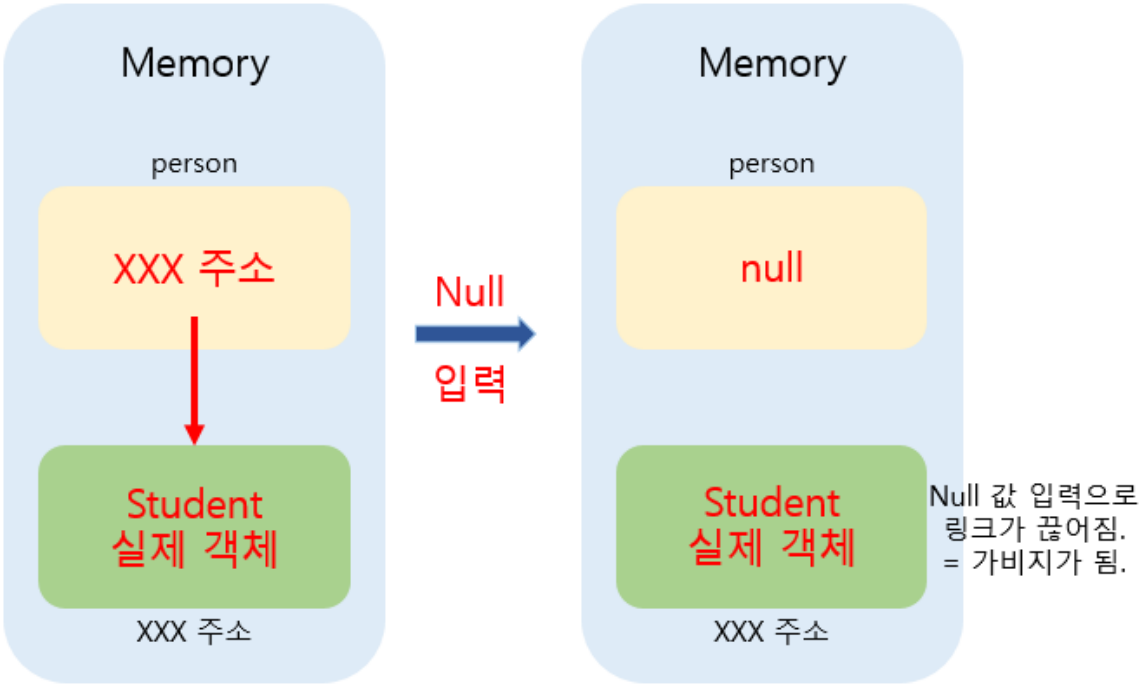
혹은 null 값을 직접 할당해줄 수도 있다.

```
Student person = null;
```

처음 기본값으로 null 을 갖는 것은 이해가 되는데 왜 null 값을 넣어주는걸까?

= 메모리 릴리즈 때문이다!

JVM 을 공부하면서 가비지 컬렉터를 배웠는데 이 가비지 컬렉터가 레퍼런스 타입에서 다 쓰고 난 객체들을 수집해서 해지(Release) 해준다.



이미지 참고: [Java 강의25] 자바 데이터 타입 - 2(레퍼런스 타입) : 네이버 블로그

링크가 하나도 없는 객체는 재사용이 불가능하기 때문에 가비지라고 한다.

이런 가비지들은 자동으로 가비지 컬렉터가 수집해가기 때문에 메모리 낭비를 막을 수 있다.

+ 할당된 stack이 해제되는 시점

> heap 영역은 사용 완료 후, GC(가비지 컬렉터)가 해제한다.
> 하지만 stack 은 GC의 영역이 아니다. 그럼 **stack**은 언제 해제될까??
>
> 기본적으로 새로운 스레드가 생기면 stack 영역에 **frame** 을 만든다.

이 frame 은 스레드의 메서드가 호출될 때의 수행 정보(메서드 호출 주소, 매개 변수, 지역 변수, 연산 스택)를 담고 있는 단위이다.
각각의 frame은 stack 영역을 갖는다.

즉, frame 안에 레퍼런스 타입의 실제 값, stack 값들이 있다. 그래서 stack은 "thread-safe"하다고 표현되는데, 각각의 스레드가 가지는 frame 영역은 다른 스레드와 공유하지 않기 때문이다.

그리고 하나의 frame은 스레드가 종료됨에 따라 pop 되어, 해제된다.

Stack 과 Heap 에 각각 데이터가 저장/소멸되는 과정
: [JAVA] Stack & Heap? 자바에서 스택과 힙이란?

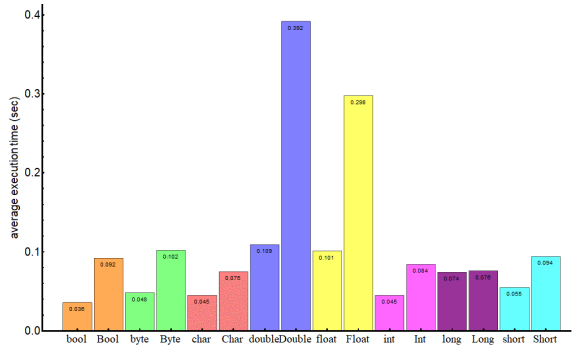
이렇게 보면

"그럼 그냥 편하게 레퍼런스 타입만 쓰면 안되는 건가...?"

하는 생각이 들지만, 프리미티브 타입은 성능 상의 이점이 있다.

우선, 프리미티브 타입은 직접 값을 저장하기 때문에 레퍼런스 타입보다 접근 속도가 빠르다. 레퍼런스 타입은 값을 필요로 할 때마다 언박싱 과정을 거쳐야하기 때문이다.

또한, 레퍼런스 타입은 주소값을 저장하는 공간, 실제 값이 저장되어 있는 공간 두 가지가 모두 필요하기 때문에 프리미티브 타입보다 레퍼런스 타입이 사용하는 메모리의 양이 압도적으로 높다.



원시타입이 사용하는 메모리	참조타입이 사용하는 메모리
boolean - 1bit	Boolean – 128 bits
byte - 8bits	Byte - 128bits
short, cagr - 16bits	Short, Charater - 128bits
int, float - 32bits	Integer, Float - 128bits
long, double - 64bits	Long, Double - 196bits

이미지 출처: 원시타입, 참조타입(Primitive Type, Reference Type)

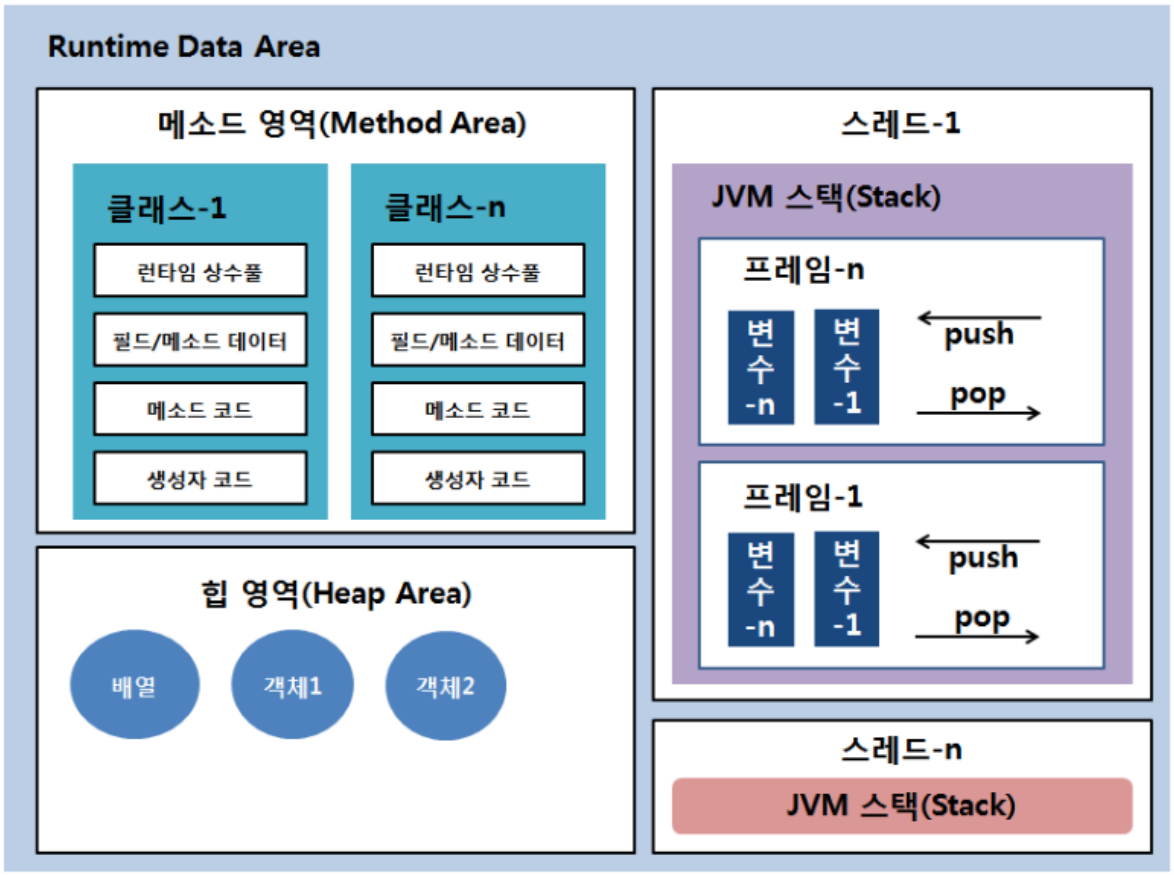
Stack, Heap 메모리 공간

앞에서 살펴봤듯이 프리미티브 타입은 Stack 영역에 저장이 되는 반면, 레퍼런스 타입은 Heap 영역에 메모리가 저장된다. 구조와 함께 한 번 더 짚고 넘어가보자면!

Stack 영역은 컴파일 시점에 크기가 결정되는 영역이다. 매개변수, 지역 변수 등이 저장된다.

Heap 영역은 런타임에 크기가 결정되는 영역이다. Heap 영역에는 주로 긴 생명주기를 가지는 데이터들이 저장되는데 모든 Object 타입은 heap 영역에 생성된다. 몇 개의 스레드가 존재하든지 상관없이 단 하나의 heap 영역만 존재한다.

Runtime Data Area 은 최종적으로 이런 구조를 띄게 된다.



이미지 출처: https://koreanfoodie.me/635

데이터 공간 구조를 보다가 문득 프리미티브/레퍼런스 타입과 전역/지역 변수 사이의 연관성이 있을까? 라는 의문이 들었다. 그래서 **JVM** 메모리 구조 및 변수 적재 위치를 함께 찾아보았다.

- 변수의 스코프와 라이프타임

기본적으로 Java 에서는 크게 전역변수, 지역변수로 먼저 구분되고 전역변수는 클래스 변수, 인스턴스 변수(객체 변수)로 세분화 된다.

변수의 종류를 결정짓는 요소는 변수가 어느 위치에서 선언되었느냐에 따라 변하게 된다.

종류		선언 위치	생성 시기	특징
전역 변수	클래스 변수 **(=static 변수)**	클래스 영역	클래스가 메모리에 올라갈 때	- static 메모리에 생성 - 프로그램 실행 시 생성, 종료 시 소멸 - 변수의 초기화 지원(초기화 필요 없음)
	인스턴스 변수 **(=객체 변수)**		인스턴스가 생성되었을 때	- heap 메모리에 생성 - GC에 의해 메모리 소멸
지역변수		클래스 영역 이외의 영역 (메서드, 생성자, 초기화 블럭 내부)	메소드 블럭 안에서 변수 선언문이 수행되었을 때	- 메서드 수행 시 메모리에 생성 - stack 메모리에 생성 - 초기화가 안되므로 초기화 후 사용 가능 - 메소드 종료 시 메모리에서 소멸

출처: https://k9e4h.tistory.com/389

지난 시간 JVM의 동작 과정을 생각해보면 쉽다.

1) 전역 변수(= 멤버 변수, 필드)

먼저 컴파일러가 .java 파일을 .class 파일로 컴파일 할 때, 클래스가 통째로 메모리에 올라가면서 static 이 붙은 클래스 변수, 즉 static 변수들은 모든 인스턴스가 공통된 저장공간을 공유하게 되므로 딱 한 번, **Method Area**(=Static Area) 에 생성되게 된다.

반대로 인스턴스 변수는 서로 다른 값을 갖기 위해 독립된 저장공간을 가져야한다. 인스턴스가 생성될 때마다 생성이 되며 Method Area 가 아닌 **heap** 메모리에서 생성된다. 소멸은 위에서 확인했듯이 링크가 하나도 없는 객체는 가비지로 취급되어 가비지 콜렉터에 의해 주기적으로 소멸된다.

+ 문득 왜 객체는 Heap 영역에 저장되는 걸까 궁금해서 찾아본 추가 자료

  : 객체를 힙 영역에 저장하는 이유

2) 지역 변수

메소드 블럭 안에 선언된 변수들로, 블럭이 끝나는 순간(**=}**) 소멸된다.

같은 원리로 for 문, while 문의 블럭 내에서 선언된 지역 변수는 해당 지역변수가 선언된 블럭 안에서만 사용될 수 있다.

또한, 여기서 살펴볼 수 있는 점이 전역 변수, 그 중에서도 클래스 변수의 경우 초기화를 하지 않더라도 컴파일러가 기본값을 넣어주는 반면 지역 변수는 초기화를 해주지 않기 때문에 프로그래머가 직접 초기화를 해야한다. 왜 그럴까? 싶어서 한 번 찾아보았다!

```
public class Test {
	static int a; // static 전역변수
	public static void main(String[] args) {
		System.out.println(a); // 기본값 0 출력
	}
}

public class Test {
	public static void main(String[] args) {
		int a; // 지역변수
		System.out.println(a); // Error
	}
}
```

위의 결과와 같이 static 전역 변수는 컴파일러에 의한 초기값이 할당되나, 지역변수는 컴파일러가 초기값을 할당해주지 않는다는 결과를 확인할 수 있다.

찾아봐도 언어 스펙에서 그렇게 정의했다라고만 나오고 정확한 이유는 오라클 문서에서도 확인할 수 없는데 이에 대해 내가 본 블로그에서는 이렇게 기술하였다.

> 지역변수의 경우 메서드 안에서 사용되며, 에러가 발생했을 때 사용되는 메서드만 보면 됩니다. 즉 해당 메서드의 어떤 변수가 문제가 있는지만 보면 된다고 생각합니다.
>
> 하지만 전역 변수의 경우 해당 변수는 여러 메서드에 걸쳐서 사용될 수 있습니다. 즉 컴파일 시점에 어디서 에러가 발생하는지 파악하기 어렵기 때문에 컴파일러가 알아서 초기값을 넣어주는 게 아닌가 생각합니다.
>
> 여기서 말하는 “어디서 에러가 발생하는지”의 의미는 “컴파일러가 해당 변수를 읽는 시점에 해당 변수가 어디서 사용될지 알 수 없다”의 의미입니다. 그렇기 때문에 런타임 시점에 에러가 발생하는 것보다는 그냥 해당 시점에 기본값을 넣고, 에러를 발생시키지 않는 것이 좋아 보이기 때문인 것으로 혼자 추측하고 있습니다.

출처: https://yeon-kr.tistory.com/180

다른 포스팅에서도 이와 비슷한 의견을 내었다.

: Weekly Java: 인스턴스 변수는 기본값으로 초기화되지만, 왜 지역 변수는 초기화되지 않나요? | by Sigrid Jin

한 번 스터디에서 다 같이 생각해보면 좋겠다...!

- ## 타입 변환, 캐스팅 그리고 타입 프로모션

연산을 수행하기 위해서는 대상이 서로 같은 타입이어야 한다. 그래서 만약 타입이 다르다면 수행 전 같은 타입으로 만들어주어야하는데 이렇게 타입 변환을 하는 것을 형변환이라고 한다.

형변환은 작은 데이터 타입에서 큰 데이터 타입으로 변환하는 프로모션(자동/묵시적 형변환)과 큰 데이터 타입에서 작은 데이터 타입으로 변환하는 캐스팅(명시적 형변환)이 있다.

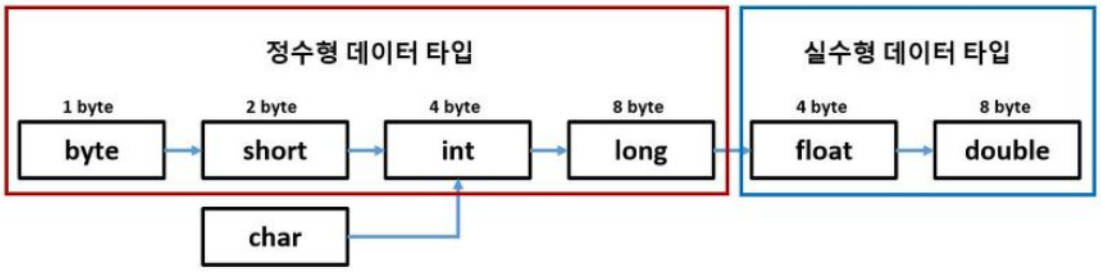
- 형변환의 특징

  - 더 큰 범위의 타입으로 변환하는 경우, 데이터는 손실되지 않는다.
  - 하지만 더 작은 범위의 타입으로 변환하는 경우 데이터는 손실될 수 있고, 변환하려는 타입을 입력해주어야 한다.

    ex) float의 소수는 int 로 표현할 수 없기 때문에 변환 중 손실된다.

  - 프리미티브 타입에서 boolean 을 제외한 나머지 타입들은 서로 형변환이 가능하다.
  - 기본형과 참조형간의 형변환은 불가능하다.

자동 형변환이 되는 순서는 아래와 같다. 반대 방향으로의 변환은 반드시 강제 형변환을 해주어야한다!



이미지 출처: https://inor.tistory.com/40

- 타입 추론, var

기본적으로 Java는 정적 언어기 때문에 정의할 때 타입을 명시해줘야한다.

+ 정적(강타입) 언어 / 동적(약타입) 언어

- 정적 언어: 컴파일러 검사 과정에서 형변환을 금지하고 컴파일을 중단하면 강타입
- 동적 언어: 형이 서로 다르면 경고 표시를 하는데, 자동으로 형변환을 시킴.

즉, 컴파일 시 잘못된 타입으로 정의하면 컴파일 에러가 발생한다. 하지만..! 놀랍게도 명시하지 않아도 알아서 추론해주는 게 있다! 원래부터 있었던 건 아니고, 자바 10부터 변수에 대해 타입 추론을 지원한다.

타입추론은 말그대로 개발자가 변수의 타입을 명시적으로 적어주지 않고도, 컴파일러가 알아서 이 변수의 타입을 대입된 리터럴로 추론하는 것이다.

+ 리터럴이란?

: 데이터 그 자체. 프로그램에서 직접 표현한 값

ex) 아래 코드에서 리터럴은 A가 된다.

```
car c = 'A';
```

타입 추론은 **var** 라는 Local Variable Type-Inference 를 사용하여 실행한다. 단, var 는 초기화가 가능한 지역 변수에 한해서 사용 가능하다.

```
var name = "name"; // java 10 이상에서 지원되는 타입추론 var
String name = "name"; // java 9 이하
```

- 1차 및 2차 배열 선언하기

Java에서 배열 선언은 창피한 말이지만 매번 다른 언어들의 배열 선언과 헷갈려서 검색해보고 선언한다... 이번 기회에 기왕이면 외워놓자

배열이란?

: 동일한 자료형(Data Type)의 데이터를 연속된 공간에 저장하기 위한 자료구조.
-> 연관된 데이터를 그룹화하여 묶어준 것!

1. 배열의 선언

: 배열을 선언하는 건 단순히 생성된 배열을 다루기 위한 참조변수를 위한 공간만 만든다.

```
// 1차원 배열
int[] nums; // 타입[] 변수명;
String[] names;

int score[]; // 타입 변수명[];
String names[];

// 2차원 배열
int[][] numArr;
String[][] stringArr;
```

2. 배열의 생성

```
: 배열을 생성해야 값을 저장할 수 있는 공간이 만들어진다.
//  길이가 5인 1차원 배열 생성
nums = new Int[5];
// 3의 크기의 배열을 4개 가질 수 있는 2차원 배열 생성
numArr = new int[4][3];
```

3. 배열의 선언과 생성 동시에 하는 법

```
int[] nums = new int[5]; // 1차원 배열 선언과 생성 작업을 동시에 진행
int[][] numArr = new int[4][3]; // 2차원 배열
int[] odds = {1, 3, 5, 7, 9}; // 값을 직접 할당해줄수도 있다
```

3주차 과제: 연산자

https://github.com/whiteship/live-study/issues/3

연산자는 산출 방법에 따라 산술, 부호, 문자열, 대입, 증감, 비교 연산자로 구분하고, 피연산자 수에 따라 단항, 이항, 삼항 연산자로 구분할 수 있다.
먼저 피연산자 수에 따라 한 번 나누고, 산출 방법에 따라 한 번 더 구분해보았다.

1. 단항 연산자

피연산자가 단 하나뿐인 연산자이다. 부호, 증감, 논리 부정 연산자가 단항 연산자에 속한다.

1) 부호 연산자(+, -)
: 양수 및 음수를 표시하는 연산자이다. boolean 타입과 char 타입을 제외한 나머지 기본 타입(=프리미티브 타입)에 사용할 수 있다.

2) 증감 연산자(++, –)
: 변수의 값을 1 증가(++) 시키거나, 1 감소(--)시키는 연산자.
boolean 타입을 제외한 모든 기본 타입의 피연산자에 적용 가능.

3) 논리 부정 연산자(!)
: true 를 false로, false를 true로 변경하기 때문에 boolean 타입에만 사용 가능.

피연산자는 편의상 oper으로 표기한다.

종류	연산식		연산식 결과
부호	+	oper	oper 부호 유지
	-	oper	oper 부호 변경
증감	++	oper	다른 연산 수행하기 전에 oper 값을 1 증가시킴
	--	oper	다른 연산 수행하기 전에 oper 값을 1 증가시킴
	oper	++	다른 연산 수행한 후에 oper 값을 1 증가시킴
	oper	--	다른 연산 수행한 후에 oper 값을 1 증가시킴
논리 부정	!	oper	oper 가 true 라면 false를, false 라면 true 를 산출.

---

2. 이항 연산자

피연산자가 2개인 연산자. 산술, 관계/비교, 논리, 비트, instanceof, 대입, 문자열 결합 연산자가 여기에 속한다.

1) 산술 연산자(+, -, *, /, %)

: 사칙연산과 나머지 구하기를 수행하는 연산자. boolean 타입을 제외한 모든 기본 타입에 사용 가능.

2) 관계/비교 연산자(<, <=, >=, >, ==, !=)

: 피연산자의 대소 또는 동등을 비교해서 true/false 산출.

대소 연산자는 boolean을 제외한 기본 타입에 사용 가능. 동등 연산자는 모든 타입에 사용 가능.

비교 연산자는 if, for, while 문에서 주로 이용되어 실행 흐름을 제어할 때 사용된다!

3) 논리 연산자(&&, ||, ^, !)

: 논리곱, 논리합, 배타적 논리합, 논리 부정 연산을 수행. boolean 타입만 사용 가능.

4) 비트 연산자(<<, >>, >>>, ~, &, |, ^)

: 비트 연산자에는 비트 이동을 하는 비트 이동 연산자와 논리 연산을 하는 비트 논리 연산자가 있다. 비트 논리 연산자는 논리 연산자와 동일하게 논리곱, 논리합, 배타적 논리합, 논리 부정 연산을 수행하지만, 피연산자의 비트에 수행하며 연산 기호에도 차이가 있다. 정수 타입에만 사용 가능하다.

5) 대입(assignment) 연산자(=)

: 값을 변수에 할당할 때 쓰는 연산자.

종류	연산자	연산식 결과
산술	+	덧셈 연산
	-	뺄셈 연산
	*	곱셈 연산
	/	왼쪽 피연산자를 오른쪽 피연산자로 나눗셈 연산
	%	왼쪽 피연산자를 오른쪽 피연산자로 나눈 나머지를 구하는 연산
관계/비교	==, !=	동등 비교. 두 피연산자의 값이 같은지/다른지 검사.
	<, <=, >=, >	크기 비교. 두 피연산자의 값의 대소 비교.

논리	&&	논리곱. 피연산자 모두가 true일 경우에만 true 산출.
	||	논리합. 피연산자 중 하나만 true이면 true 산출.
	^	배타적 논리합. 피연산자가 서로 다를 경우에만 true 산출.
	!	논리 부정. 피연산자의 논리값을 바꿈.
비트	<<, >>>	왼쪽 피연산자의 각 비트를 오른쪽 피연산자의 값만큼 좌우로 이동시킨다. 빈자리는 0으로 채워진다.
	>>	왼쪽 피연산자의 각 비트를 오른쪽 피연산자의 값만큼 오른쪽으로 이동시킨다. 빈자리는 왼쪽 피연산자의 최상위 부호비트와 같은 값으로 채워진다.
	&, |, *, ~	논리 연산자에 적용된 바를 각 비트에 적용한 값이 산출된다.
대입	연산식이 산술 연산자, 비트 연산자와 = 기호가 합쳐진 형태로, 산술/비트 연산자를 연산한 후 그 결과값을 왼쪽 피연산자에 대입한다.	

6) 문자열 결합 연산자(+)

: 문자열을 서로 결합하는 연산자. 문자열과 숫자가 혼합된 연산식은 나머지 피연산자도 문자열로 자동 형변환이 진행되고, 왼쪽부터 오른쪽으로 연산 진행.

7) instanceof 연산자(객체타입 확인)

: 객체 타입을 확인하는 데 사용된다. 형변환 가능 여부에 따라 true, false 값이 산출된다. 주로 상속 관계에서 부모 객체인지 자식 객체인지 확인하는데 사용한다.

```
class Parent{}
class Child extends Parent{}

public class InstanceofTest {
    public static void main(String[] args){
        Parent parent = new Parent();
        Child child = new Child();

        System.out.println( parent instanceof Parent );  // true
        System.out.println( child instanceof Parent );   // true
        System.out.println( parent instanceof Child );   // false
        System.out.println( child instanceof Child );    // true
    }
```

```
}
```

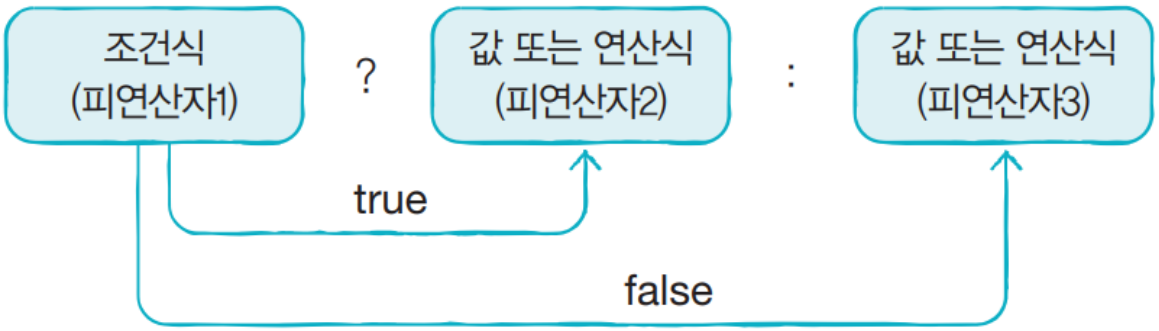
출처: https://computer-science-student.tistory.com/336

8) 화살표(->) 연산자

: 화살표 연산자는 Java 8 부터 추가된 것으로, 람다 표현식과 함께 사용된다.

3. 삼항 연산자(?:)

: 삼항 연산자는 조건식 하나와 값 또는 연산식 두 개로 이루어져 조건식의 true / false 값에 따라 콜론 앞 뒤의 피연산자가 선택되기 때문에 조건 연산식이라고도 불린다.



이미지 출처: 자바 기초: 연산자 & 연산의 방향과 우선순위

4. 연산의 방향과 우선순위

위에서 다루었던 모든 연산자들의 우선순위 및 연산 방향은 아래와 같다!

이미지 출처: [자바 기초: 연산자 & 연산의 방향과 우선순위]

4주차 과제: 제어문

https://github.com/whiteship/live-study/issues/4

제어문이란 코드의 실행 흐름(순서)를 제어하는 구문!

- 선택문
- 반복문

https://kils-log-of-develop.tistory.com/349

https://www.notion.so/Live-Study-4-ca77be1de7674a73b473bf92abc4226a

[Java] 선택문과 제어문

출처

2주차

- https://league-cat.tistory.com/407
- https://yunanp.tistory.com/18
- https://yeastriver.tistory.com/7
- https://yeon-kr.tistory.com/180
- https://jithub.tistory.com/40
- https://shanepark.tistory.com/2
- https://bytheprogramer-fortheprogramer.tistory.com/5
- https://keep-cool.tistory.com/14
- https://i-am-seongni.tistory.com/30
- [JAVA/자바] 배열(Array) 선언 및 사용 방법 : 네이버 블로그

3주차

- 자바 기초: 연산자 & 연산의 방향과 우선순위
- https://catch-me-java.tistory.com/24
- https://park-youjin.tistory.com/16
- https://kils-log-of-develop.tistory.com/336
- https://park-youjin.tistory.com/17
- https://recoderr.tistory.com/24
- https://computer-science-student.tistory.com/336
- http://www.tcpschool.com/java/java_operator_assignment
- https://coding-factory.tistory.com/521

4주차

-