자바 스프링

과제 01 : JVM은 무엇이며 자바 코드는 어떻게 실행하는 것인가.

JVM이란 무엇인가

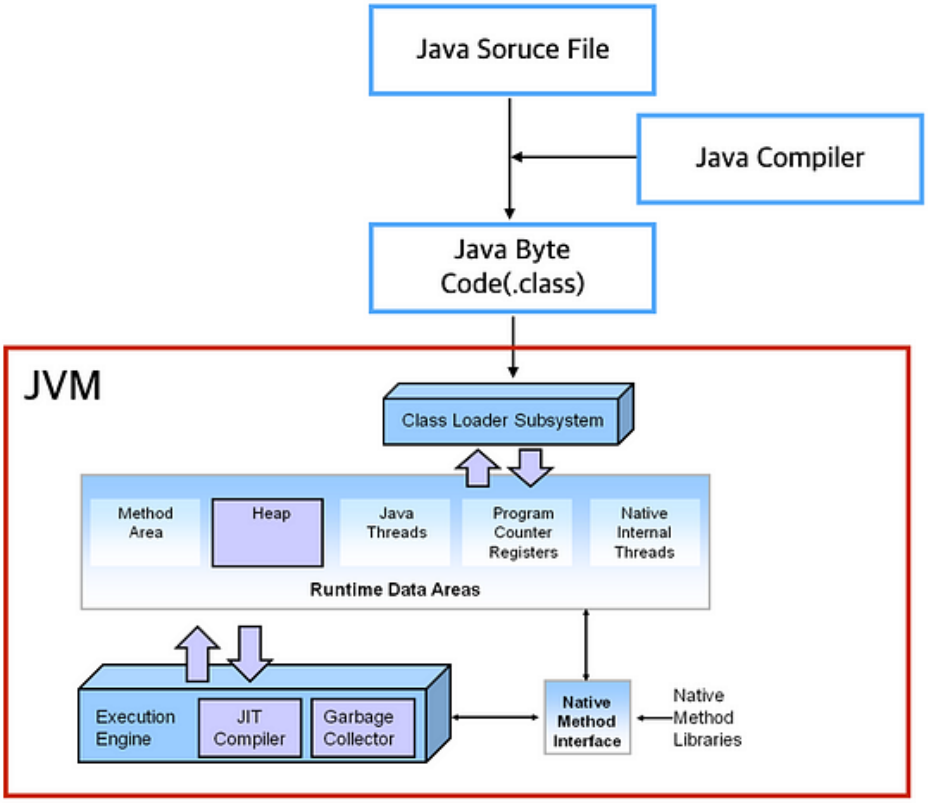
Java virtual machine

자바를 실행하기 위한 가상 컴퓨터

자바 애플리케이션은 jvm이랑만 상호작용하기 때문에, os와 하드웨어에 독립적이다.

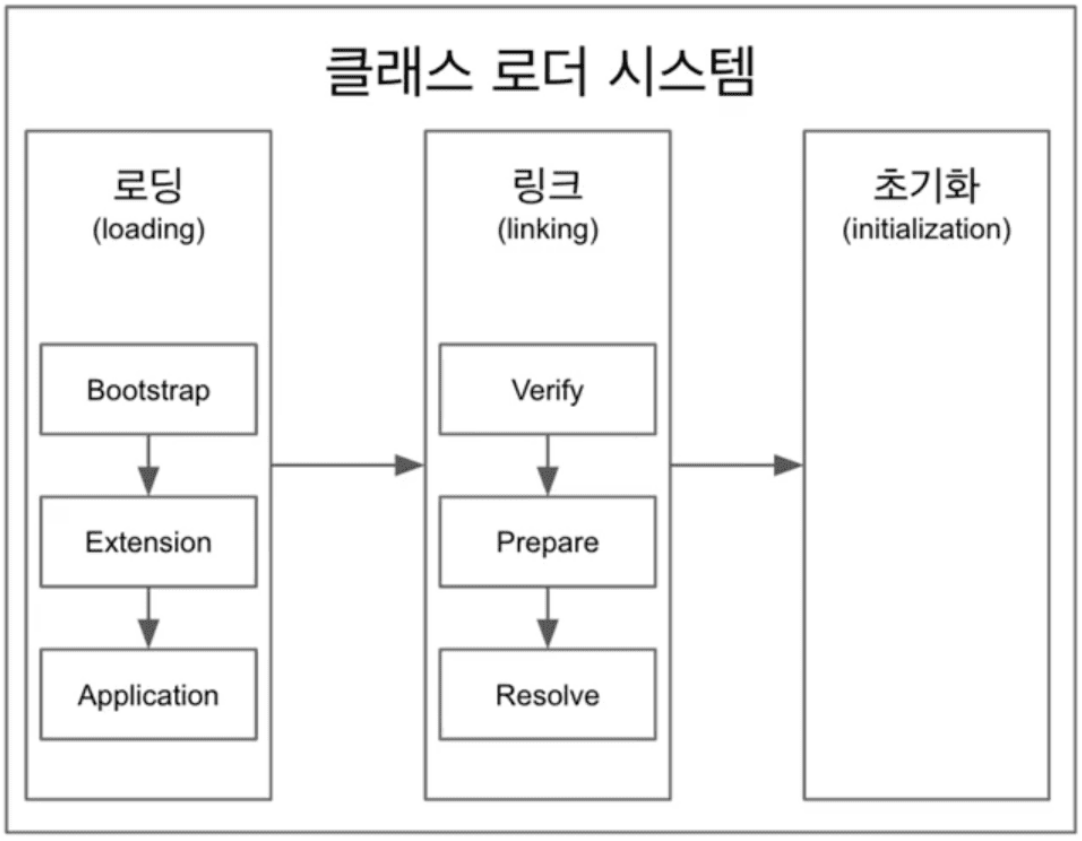
반대로 jvm은 os에 종속적이기 때문에, 각 os에 맞는 jvm을 설치해야 한다.

JVM 구성 요소



▼ Class Loader

Runtime 시점에 .class에서 바이트코드를 읽고 메모리에 저장



▼ 로딩

클래스 로더가 .class 파일을 읽고 적절한 바이너리 데이터를 만들어서 메소드 영역에 저장

Fully Qualified Class Name란? (java.lang.String s = **new** java.lang.String();)

로딩이 끝나면 해당 클래스 타입의 class객체를 생성하여 heap에 저장

- 위임 원칙
  클래스 로더는 클래스 또는 리소스를 찾기 위해 요청을 받았을 때, 상위 클래스 로더에게 책임을 위임하는 위임 모델을 따른다.
- 가시 범위 원칙
  하위 클래스 로더는 상위 클래스 로더가 로드한 클래스를 볼 수 있지만, 반대로 상위 클래스 로더는 하위 클래스 로더가 로드한 클래스를 알 수 없다.
- 유일성의 원칙
  - 하위 클래스 로더가 상위 클래스 로더에게 로드한 클래스를 다시 로드하지 않아야 한다는 원칙이다.
  - 위임 원칙에 의해서 위쪽으로 책임을 위임하기 때문에 고유한 클래스를 보장할 수 있다.

▼ 링크

Verify, Prepare, Resolve(Optional) 세 단계로 나눠져 있다.

verify : .class 파일 형식이 유효한지 체크한다.

Preparation : 클래스 변수(static 변수)의 기본값에 따라 필요한 메모리

Resolve : 심볼릭 메모리 레퍼런스를 메서드 영역에 있는 실제 레퍼런스로 교체한다.

심볼릭 메모리 레퍼런스란? 객체 이름만 할당해둔거

▼ 초기화

Static 변수의 값을 할당한다.(static 블럭이 있다면 이때 실행된다.)

▼ Runtime Data Areas

Heap과 Method는 모든 쓰레드가 공유 나머지는 쓰레드 마다 생성

JVM이 프로그램을 수행하기 위해 OS한테 할당받은 메모리 공간

PC Register : CPU가 Instruction을 수행하는 동안 필요한 정보를 저장

▼ Instruction

instruction은 Opcode(연산자)와 Oprand(피연산자)로 구성

Opcode는 mov, add, jump와 같은 연산 기호로 데이터 처리, 조건문, 메모리에 있는 데이터를 불러오거나 저장, I/O와 통신하는 작업으로 분류

Oprand는 연산할 데이터를 저장하고 있는 메모리 주소

opcode와 oprand가 하나의 instruction에서 차지하는 크기는 ISA마다 다름

JVM Stack: Thread가 시작될 때 생성되며 Method와 Method 정보 저장

Method란? 클래스에 포함되어있는 함수

Native Method Stack: Java 이외의 언어로 작성된 native 코드를 위한 Stack(JNI)

Method Area: 모든 쓰레드가 공유하는 메모리 영역(클래스, 인터페이스, 메소드, 필드, Static 변수등의 바이트 코드 등을 보관)

Heap: 런타임시 동적으로 할당하여 사용하는 영역 class를 통해 instance를 생성하면 Heap에 저장됨

instance란? class라는 붕어빵 틀로 찍어낸 붕어빵(객체)

Heap의 경우 명시적으로 만든 class와 암묵적인 static 클래스(.class 파일의 class)가 담긴다.

또한 암묵적인 static 클래스의 경우 클래스 로딩 시 class 타입의 인스턴스를 만들어 힙에 저장한다. 이는 Reflection에 등장한다.

▼ Execution Engine

Load된 Class의 ByteCode를 실행하는 Runtime Module

Class Loader를 통해 JVM 내의 Runtime Data Areas에 배치된 바이트 코드는 Execution Engine에 의해 실행(바이트 코드를 명령어 단위로 읽어서 실행)

컴파일 하는 방법

1. Java Compiler(javac 명령어로 실행)가 Java Source(.java 확장자)로부터 Byte Code(.class 확장자)가 생성된다.
2. JVM에 있는 Class Loader에 의해 Byte Code는 JVM내로 로드되고, 실행엔진에 의해 기계어로 해석되어 메모리 상(Runtime Data Area)에 배치된다.
3. 실행엔진에는 interpreter(코드를 한 줄씩 읽는 프로그램)와 JIT(just-in-time) Compiler가 있는데, interpreter에 의해 byte code를 한 줄씩 읽어 실행하다가 적절한 시점에 byte code 전체를 컴파일하고 더이상 인터프리팅하지 않고 해당 코드를 직접 실행한다.

JIT 컴파일러는 캐시에 보관하기 때문에, 한번 컴파일후에는 빠르게 실행된다.

인터프리터는 한 줄씩 실행하기 때문에 느리지만, 한번만 실행해도 되는 코드에 대해서는 인터프리터가 유리함

컴파일 장점

- 생성되는 코드의 안전성
    - 자바가 수행 중에 만들어내는 기계어 코드가 안전한 공간안에서 돌아가기 때문에 외부 해킹에 안전
- 동작하는 메모리 공간의 안전성
    - 모든 자바 객체들은 heap에서만 수행
    - 다른 프로세스들과 다른 메모리를 쓰기 때문에 Stack overflow에 강하다.
- 최적화 재사용에 유일한 관련 클래스간 상속구조
    - 메모리 위치상 가깝게 관련된 객체와 메소드들을 위치시킨다.
    - method inlining 같은 성능을 높이기 위한 기술들이 자바에서 효율적으로 작동
- 동적 최적화와 취소 및 재 최적화 가능
    - static 언어와 다르게 dynamic class loading으로 어떤 방식으로든 변경 가능하며, 컴파일러를 통해 수시로 최적화

실행하는 방법

뭘 실행한다는거야

바이트 코드란 무엇인가

바이트 코드(Bytecode, portable code, p-code)는 특정 하드웨어가 아닌 가상 컴퓨터에서 돌아가는 실행 프로그램을 위한 이진 표현법이다. 하드웨어가 아닌 소프트웨어에 의해 처리되기 때문에 보통 기계어보다 더 추상적이다.

JVM이 이해할 수 있는 언어로 변환된 자바 소스코드를 의미

자바 컴파일러에 의해 변환되는 코드의 명령어의 크기가 1byte라서 자바 바이트 코드라고 불림

JIT 컴파일러란 무엇이며 어떻게 동작하는지

JIT 컴파일 (just-in-time-compilation) 또는 동적 번역(dynamic translation)은 프로그램을 실제 실행하는 시점에 기계어로 번역하는 컴파일 기법이다.

JIT 컴파일러는 실행 시점에서 인터프리트 방식으로 기계어 코드를 생성하면서 그 코드를 캐싱하여, 같은 함수가 여러 번 불릴 때 매번 기계어 코드를 생성하는 것을 방지한다.

자바 컴파일러가 자바 프로그램 코드를 바이트 코드로 변환한 다음, 실제 바이트 코드를 실행하는 시점에서 자바 가상 머신이 바이트 코드를 JIT 컴파일을 통해 기계어로 번역한다.

JDK와 JRE의 차이

jre(Java Runtime Environment) 자바 프로그램을 실행하는데 필요하다.

jdk(Java Development Kit) 자바를 개발하는데 필요한 기능들 (jre 포함)

과제 01 질문

클래스 로더의 세가지 원칙은 무엇인가요?

과제 02: 자바 데이터 타입, 변수 그리고 배열

프리미티브 타입 종류와 값의 범위 그리고 기본 값

	타입	기본값	값의 범위	값의 크기
정수형	byte	0	-128 ~ 127	1byte
	short	0	-32,768 ~ 32,767	2byte
	int	0	-2,147,483,648 ~ 2,147,483,647	4byte
	long	0L	9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807	8byte
실수형	float	0.0f	(3.4 X 10-38) ~ (3.4 X 1038) 의 근사값	4byte
	double	0.0	(1.7 X 10-308) ~ (1.7 X 10308) 의 근사값	8byte
문자형	char	null	0 ~ 65,535	2byte
논리형	boolean	FALSE	false, ture	1byte

프리미티브 타입과 레퍼런스 타입

프리미티브 타입(Primitive Type)

실제 데이터값을 저장

기본값이 있기 때문에 null이 존재하지 않음(캐릭터형의 경우 ascll 0(null)('\u0000')인 것)

값이 할당될때 JVM의 Runtime Data Area 영역 중 Stack영역에 값이 저장됨

값의 범위를 벗어나면 컴파일 에러가 발생

레퍼런스 타입(Reference Type)

대표적으로 class, interface, enum, array, string type이 있음

원시타입(프리미티브 타입)을 제외한 모든 타입은 참조형(주소값)

빈 객체를 의미하는 NULL이 있음

주소값을 저장하기때문에 데이터 영역은 Heap

리터럴

데이터 그 자체. 변수에 넣는 변하지 않는 데이터.

정수, 실수, 문자, 논리, 문자열 리터럴(final String PROTOCOL_NAME = “https”;)

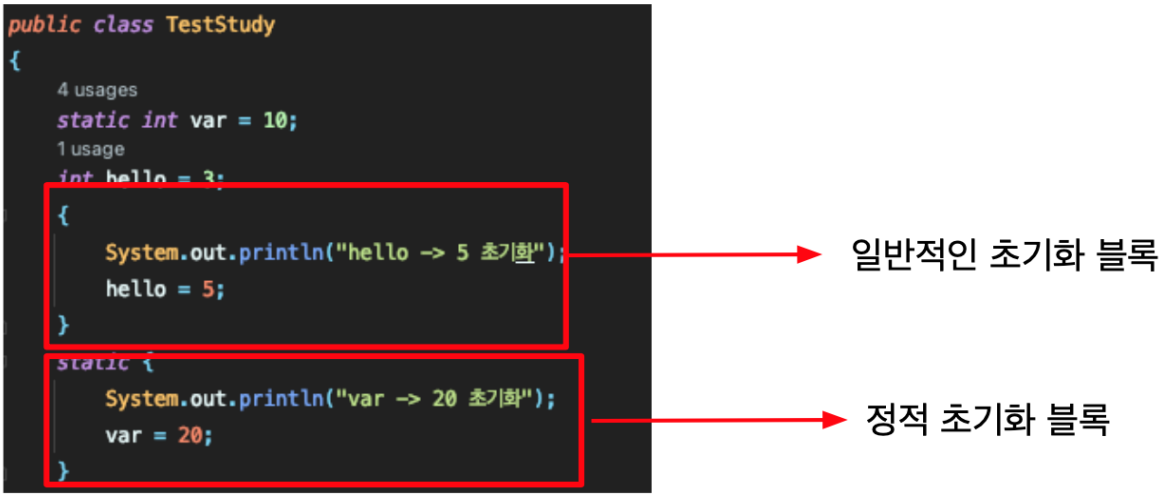
변수 선언 및 초기화하는 방법

C와 같다.

지역변수와 멤버변수가 나뉘는데 클래스안에서 선언하는 멤버변수가 전역변수같은 역할을 함.

멤버변수는 선언하자마자 초기화 되어있음.

지역변수는 알다시피 함수 안에서만 쓰고 사라지는 애들



일반적인 초기화 블록 : Class가 new 를통해 인스턴스를 생성하는 순간 초기화가 진행이 된다.

정적 초기화 블록 : JVM이 클래스로더로 로딩 시점에 초기화 진행

변수의 스코프와 라이프타임

변수의 라이프타임(lifetime) 변수가 메모리에서 살아있는 시간

변수의 스코프(scope) 어디서부터 어디까지인지 (예제 보셈)

▼ **Instance Variables**

```
class LeaguecatSample
{
    int x, y; //인스턴스 변수
    static int result;

    void add(int a, int b) // a 와 b는 로컬 변수
    {
        x = a;
        y = b;
        int sum = x + y; //Sum
        System.out.println("Sum = "+sum);
    }

    public static void main(String[] args)
    {
        Sample obj = new Sample();
        obj.add(10,20);
    }
}
```

정의 : 클래스 내부와 모든 메소드 및 블록 외부에서 선언된 변수

scope : 정적 메서드를 제외한 클래스 전체

라이프타임 : 객체가 메모리에 남아있을 때까지.

예시 : x와 y의 scope

▼ **Class Variables**

```java
class LeaguecatSample
{
    int x, y; //인스턴스 변수
    static int result;   // Classvariable

    void add(int a, int b) // a 와 b는 로컬 변수
    {
        x = a;
        y = b;
        int sum = x + y; //Sum
        System.out.println("Sum = "+sum);
    }

    public static void main(String[] args)
    {
        Sample obj = new Sample();
        obj.add(10,20);
    }
}
```

정의 : 클래스 내부, 모든 블록 외부에서 선언되고 static으로 표시된 변수

scope : 클래스 전체

라이프타임 : 프로그램이 끝날때까지 또는 클래스가 메모리에 로드 되는 동안

예시 : result(class variable)의 scope

▼ **Local Variables**

```java
class LeaguecatSample
{
    int x, y; //인스턴스 변수
    static int result;   // Classvariable

    void add(int a, int b) // a 와 b는 로컬 변수
    {
        x = a;
        y = b;
        int sum = x + y; //Sum
        System.out.println("Sum = "+sum);
    }

    public static void main(String[] args)
    {
        Sample obj = new Sample();
        obj.add(10,20);
    }
}
```

정의 : 인스턴스 및 클래스 변수가 아닌 모든 변수

scope : 선언된 블록 내에 있음

라이프타임 : 컨트롤이 선언 된 블록을 떠날때까지

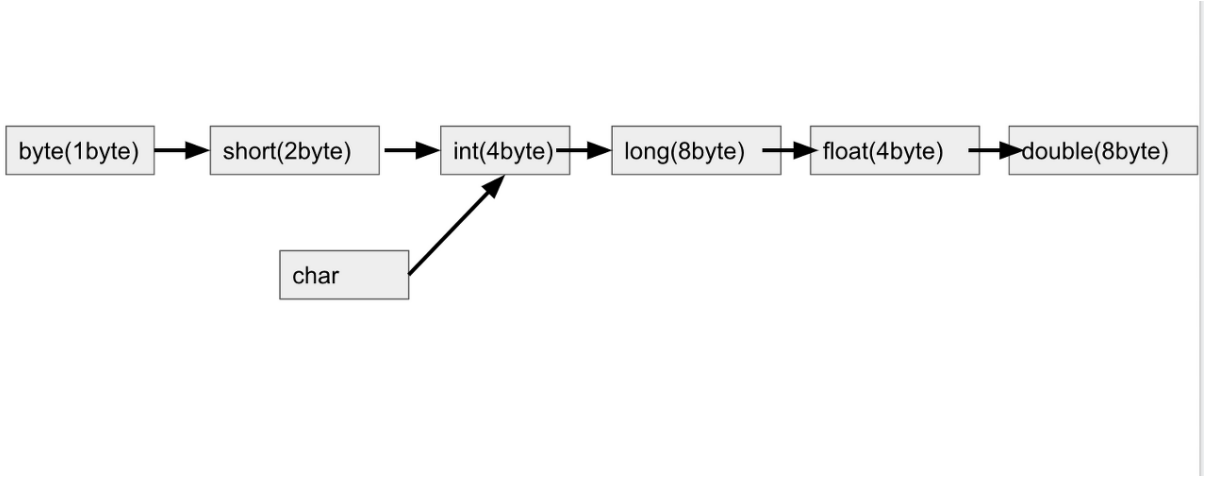
예시 : a, b (local variable)의 scope

타입 변환, 캐스팅 그리고 타입 프로모션

연산하려면 같은 타입끼리 해야함, 그래서 바꾸는게 타입 변환

프로모션 (자동 / 묵시적 형 변환) 작은 데이터 타입에서 큰 데이터 타입으로 형

캐스팅(명시적 형 변환) 큰 데이터 타입에서 작은 데이터 타입으로 형 변환




```
short shortNum = 100;
int intNum = shortNum;                  // 확장성에 의한 자동 변환 (short
-> int)

int intNum = 1;
short shortNum = (short)intNum;         // 컴파일 에러 해결
```

1차 및 2차 배열 선언하기

타입[] 변수이름

변수이름 = new 타입[길이];

타입[] 변수이름 = new 타입[길이];

```
int[] score;
score = new int[5];

int[] score = new int[5];
```

```
int[][] array;
array = new int[1][5];

int[][] array = new int[1][5];
```

타입 추론, var

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

대표적인 타입추론 언어는 자바스크립트, 코틀린, 스위프트 등이 있다.

자바 10부터 var키워드가 생김

var는 멤버변수, 또는 메소드의 파라미터, 리턴 타입으로 사용이 불가

선언할때 초기화 값이 있어야함

null 못 넣음

로컬 변수에만 선언 가능(멤버 변수가 불가능)

람다식에서는 명시적인 타입을 지정해주어야함

배열을 선언할때는 못 씀

```
for (var people : peopleList)
이런식으로 c++에서 auto쓰듯이 쓰면 되는데, 지 맘대로 쓰면 다른사람이 코드 읽을때 힘
```

자바 11부터 람다식에서도 var쓸 수 있는데, 코드 보기 힘드니까 익명 클래스에서만 쓰자

과제 02 질문

문자형 기본값은 '\u0000'인데, 이게 결국 아스키로는 널인데 왜 널이 없다고 하는걸까?

레퍼런스 타입은 주소 값을 저장하기 때문에 데이터 영역이 힙에 잡힌다.

주소값을 저장하는 그 주소값이 담긴 곳은 스택에 잡히나? (sizeof(char *))가 힙인지 스택인지

과제 03: 연산자

산술 연산자

+	왼쪽의 피연산자에 오른쪽의 피연산자를 더함
-	왼쪽의 피연산자에 오른쪽의 피연산자를 뺌
*	왼쪽의 피연산자에 오른쪽의 피연산자를 곱함
/	왼쪽의 피연산자에 오른쪽의 피연산자를 나눔
%	왼쪽의 피연산자에 오른쪽의 피연산자를 나누고 나머지를 반환함

대입 연산자 (assignment(=) operator)

=	왼쪽의 피연산자에 오른쪽의 피연산자를 대입함
+=	왼쪽의 피연산자에 오른쪽의 피연산자를 더한 후, 그 결괏값을 왼쪽의 피연산자에 대입함
-=	왼쪽의 피연산자에서 오른쪽의 피연산자를 뺀 후, 그 결괏값을 왼쪽의 피연산자에 대입함
*=	왼쪽의 피연산자에 오른쪽의 피연산자를 곱한 후, 그 결괏값을 왼쪽의 피연산자에 대입함
/=	왼쪽의 피연산자를 오른쪽의 피연산자로 나눈 후, 그 결괏값을 왼쪽의 피연산자에 대입함
%=	왼쪽의 피연산자를 오른쪽의 피연산자로 나눈 후, 그 나머지를 왼쪽의 피연산자에 대입함

증감 연산자

++x	먼저 피연산자의 값을 1 증가시킨 후에 해당 연산을 진행함
x++	먼저 해당 연산을 수행하고 나서, 피연산자의 값을 1 증가시킴
--x	먼저 피연산자의 값을 1 감소시킨 후에 해당 연산을 진행함
x--	먼저 해당 연산을 수행하고 나서, 피연산자의 값을 1 감소시킴

비교 연산자 (관계 연산자)

==	왼쪽의 피연산자와 오른쪽의 피연산자가 같으면 1을 반환함
!=	왼쪽의 피연산자와 오른쪽의 피연산자가 같지 않으면 1을 반환함
>	왼쪽의 피연산자가 오른쪽의 피연산자보다 크면 1을 반환함
>=	왼쪽의 피연산자가 오른쪽의 피연산자보다 크거나 같으면 1을 반환함
<	왼쪽의 피연산자가 오른쪽의 피연산자보다 작으면 1을 반환함
<=	왼쪽의 피연산자가 오른쪽의 피연산자보다 작거나 같으면 1을 반환함

논리 연산자

&&	논리식이 모두 참이면 1을 반환함. (논리 AND 연산)	논리곱
||	논리식 중에서 하나라도 참이면 1을 반환함. (논리 OR 연산)	논리합
!	논리식의 결과가 참이면 0을, 거짓이면 1을 반환함. (논리 NOT 연산)	배타적 논리합
^	논리식의 결과가 하나만 참일때만 1을 반환함. (논리 XOR 연산)	논리부정

기타 연산자

쉼표 연산자 : 두 연산식을 하나로 나타내고자 할때(int a, b;) | 둘 이상의 인수를 함수로 전달하고자 할 때

삼항 연산자 : 조건식 ? 반환값1 : 반환값2 ex) (num < 2) ? true : false;

sizeof 연산자 : sizeof(char) 해당 변수나 상숫값에 대항하는 타입의 크기를 반환 (크기는 컴퓨터 환경마다 다름)

형변환 연산자 : (type) 형변환 할 때 사용

포인터 연산자 : *주소값 | 주소값 안에 있는 데이터를 지칭하기 위해 사용

참조 연산자 : &변수명 | 변수명의 주소값을 불러오기 위해 사용

다이아 연산자 : <> | pair<int, int> = pair<>이런식으로 자동완성으로 사용

비트 연산자

&	대응되는 비트가 모두 1이면 1을 반환함. (비트 AND 연산)
|	대응되는 비트 중에서 하나라도 1이면 1을 반환함. (비트 OR 연산)
^	대응되는 비트가 서로 다르면 1을 반환함. (비트 XOR 연산)
~	비트를 1이면 0으로, 0이면 1로 반전시킴. (비트 NOT 연산)
<<	지정한 수만큼 비트들을 전부 왼쪽으로 이동시킴. (left shift 연산)

>>	부호를 유지하면서 지정한 수만큼 비트를 전부 오른쪽으로 이동시킴. (right shift 연산)
>>>	비트 값을 주어진 숫자 만큼 오른쪽으로 이동 시킨 후 빈공간을 모두 0으로 채운다.
a <<= b	a << b의 결과를 a에 대입
a >>= b	a >> b의 결과를 a에 대입
a &= b	a & b의 결과를 a에 대입
a ^= b	a ^ b의 결과를 a에 대입
a |= b	a | b의 결과를 a에 대입

비트를 왼쪽으로 한칸 옮기면 정수의 값은 두배가 되는데, 이를 통해 곱셈과 나눗셈 연산을 비트연산으로 대체할 수 있으며, 이는 성능 향상에 도움이 된다.

> why? bit는 최소정보단위로 다른 추상화된 정보들과 달리 즉시 해석되어 전달해서 연산 속도가 매우 빠르다.

비트를 옮길때 비는 칸은 전부 0으로 채워진다.

하지만 맨 왼쪽 음수표시를 위해 남겨둔 칸은 CPU에 따라 다르다.

instanceof

객체이름 instanceof 클래스이름

```
class Rabbit {}
let rabbit = new Rabbit();

rabbit instanceof Rabbit;
// return true

배열같은 내장클래스도 가능함
let arr = [1, 2, 3];
alert( arr instanceof Array ); // true
alert( arr instanceof Object ); // true
```

객체가 해당하는 클래스이거나 해당하는 클래스를 상속받는 클래스라면 TRUE를 리턴

아니면 FALSE (NULL 넣으면 FALSE 나옴)

도트 연산자, 화살표(->) 연산자

포인터이름->멤버변수이름 = 값

(*포인터이름).멤버변수이름 = 값

```
const materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium'];

console.log(materials.map((material) => material.length));
```

연산자 우선 순위

기본적으로 사칙연산을 따라감.

1	++	후위 증가 연산자
	--	후위 감소 연산자
	()	함수 호출
	[]	첨자 연산자
	.	참조에 의한 선택
	->	포인터를 통한 선택
2	!	논리 NOT 연산자
	~	비트 NOT 연산자
	+	양의 부호 (단항 연산자)
	-	음의 부호 (단항 연산자)
	++	전위 증가 연산자
	--	전위 감소 연산자
	(타입)	타입 캐스트 연산자
	*	참조 연산자 (단항 연산자)
	&	주소 연산자 (단항 연산자)
	sizeof	크기
3	*	곱셈 연산자
	/	나눗셈 연산자
	%	나머지 연산자
4	+	덧셈 연산자 (이항 연산자)
	-	뺄셈 연산자 (이항 연산자)
5	<<	비트 왼쪽 시프트 연산자
	>>	부호 비트를 확장하면서 비트 오른쪽 시프트
6	<	관계 연산자(보다 작은)
	<=	관계 연산자(보다 작거나 같은)
	>	관계 연산자(보다 큰)
	>=	관계 연산자(보다 크거나 같은)
7	==	관계 연산자(와 같은)
	!=	관계 연산자(와 같지 않은)
8	&	비트 AND 연산자
9	^	비트 XOR 연산자
10	|	비트 OR 연산자

11	&&	논리 AND 연산자
12	||	논리 OR 연산자
13	? :	삼항 조건 연산자
14	=	대입 연산자 및 복합 대입 연산자 (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=)
15	,	쉼표 연산자

사칙연산이 비트연산보다 빠르다.

. → 로 주소값을 전위 연산자로 올리는 케이스

++str→arr[1]; 같은 경우는 전위연산자가 더 느림

웬만하면 괄호를 치자

(optional) Java 13. switch 연산자

```
switch (day)
{
        case MON:
            System.out.println(Day.MON + " Day");
            break;
        case TUE:
            System.out.println(Day.TUE + " Day");
            break;
        case SUN:
            System.out.println(Day.SUN + " Day");
            break;
}
```

기존 switch case문에서

브레이크 없어도 되고

yield 명령어로 리턴값을 리턴해도 됨

단 모든 경우를 만들어야함

```
string s = "day";
int return_value = switch (s)
{
        case "mon"
                yield 1;
        case "tue"
                system.out.println("뭐야 월요일 아닌데?"); //yield로 리턴값
        default :
                system.out.println("월요일이랑 화요일밖에 모르는데?");
```

```
                    yield 0;
}
```

과제 04: 제어문 (실습)

선택문

switch case 문

```
int month = 1;
switch (month)
{
    case 1: String day = "Monday";
                    break;
    default : String day = "Not found";
                    break;
}
브레이크를 하지 않으면 다음 케이스도 실행됨.
default는 else와 같음
```

반복문

for 문

```
for (int i = 0; i < 10; i++)
{
        여기서 인트 i는 이 for문이 끝날때 사라짐
}
```

while 문

```
int i = 0;
while(i < 10)
{
    i++;
}
```

do while 문

```
int i = 0;
do
{
    먼저 한번 실행하고
```

```
}
while (i < 10)
{
    조건이 맞으면 몇번 더 실행함
};

break는 반복문을 끝내고
continue는 이번 반복을 끝낸다.
```

for each 문

```
for (자료형 변수이름:iterate 객체)
{
}

String[] names = {"Kim", "Lee", "Park"};
for(String name : names)
{
    System.out.println(name)
}
객체의 모든 원소들에 하나씩 접근하면서 name변수에 대입해줌.
```

실습 00. JUnit 5 학습하세요.

인텔리J, 이클립스, VSCode에서 JUnit5로 테스트 코드 작성하는 방법에 익숙해 질 것.

https://letsbegin.tistory.com/33 | 노트북으로 해야함

더 자바, 테스트 강의도 있으니 참고

실습 01. live-study 대시 보드를 만드는 코드를 작성하세요.

▼ checklist

깃헙 이슈 1번부터 18번까지 댓글을 순회하며 댓글을 남긴 사용자를 체크 할 것.

참여율을 계산하세요. 총 18회에 중에 몇 %를 참여했는지 소숫점 두자리까지 보여줄 것.

live-study 깃허브 링크 : https://github.com/whiteship/live-study

Github 자바 라이브러리를 사용하면 편리합니다.

깃헙 API를 익명으로 호출하는데 제한이 있기 때문에 본인의 깃헙 프로젝트에 이슈를 만들고 테스트를 하시면 더 자주 테스트할 수 있습니다.

토큰을 받아서 쓴다. (깃허브 홈페이지 설정 - 개발자 설정 - 토큰 어쩌고)

```
import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import org.kohsuke.github.GHIssue;
import org.kohsuke.github.GHIssueState;
import org.kohsuke.github.GHRepository;
import org.kohsuke.github.GitHub;
import org.kohsuke.github.GitHubBuilder;
import org.kohsuke.github.GHIssueComment;

// 왜 클래스 안에 있는가 (함수의 개념이 없기 때문에, 메인 메소드가 있는 클래스를 메인 클래스로 봄)
public class dashboard {

    private static final String TOKEN = "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";
    //public(접근 제어자) private -> protected -> public이 있으며, public은 어느 곳에서든 해당 객체를 참조할 수 있다는 뜻
    // String[] args를 왜 쓰는가?  || C와 C++과 달리 자바는 매개변수를 미리 정의하고 나중에 안쓰는것이 매개변수를 못 받는 main을 정의하는 것을 허용하는 것 보다 덜 헷갈린다고 생각한 자바개발자 때문임 마찬가지로 int 이딴거 넣어도 안됨
    // final은 실행할때 결정됨, const는 컴파일시에 결정됨(datetime) 함수를 쓸때는 const는 컴파일 시에 시간을 담을 수 없으므로 사용불가능함. 둘다 한번 값을 대입 하면 바꿀 수 없음

    public static void main(String[] args)
    {
        try
        {
            GitHub github = new GitHubBuilder().withOAuthToken(TOKEN).build(); // 토큰으로 로그인
            GHRepository ghRepository = github.getRepository("spring-for-hitchhiker/java-study"); // 어느 리포지토리를 쓸건지
            List<GHIssue> issues = ghRepository.getIssues(GHIssueState.ALL); // 리스트로 이슈탭을 전부 받아옴

            HashMap<String, Integer> dash_board = new HashMap<>(); // 이름 - 이슈번호 순으로 정리하기 위해 해쉬 맵 사용
            // 왜 String에서 S는 대문자임? String은 클래스라서 클래스는 대문자로 시작해야하는 법을 따름 Integer 처럼 int를 자료형으로 만든것도 있음
            // <>()가 뭐임? 제네릭을 나타내는 부분 앞에서 쓴대로 알아서 자동완
```

성 해주는 부분 '다이아 몬드 연산자' ()는 클래스 생성자를 호출하고 객체를 초기화 하는거임

```
            for (int i = 0; i < issues.size(); i++) // 이슈가 끝날 때까지
            {
                GHIssue issue = issues.get(i); // 현재 인덱스의 이슈
                List<GHIssueComment> comment_list = issue.getComments(); // 그 이슈의 댓글 리스트로 만들기
                HashSet<String> people_id = new HashSet<>(); // set으로 중복되는 아이디를 제거하기 위함 및 초기화
                for (GHIssueComment comment : comment_list) // 댓글 전부 확인
                    people_id.add(comment.getUser().getLogin()); // 전부 set에 추가
                for (String s : people_id) // set 끝날때 까지 맵에다 정리
                {
                    if (!dash_board.containsKey(s)) { // 만약 맵에 유저 이름이 없다면
                        dash_board.put(s, 1); // 유저 이름을 키로, 참여 이슈 수를 1로 저장
                    } else {
                        // 이미 유저 이름이 있을 경우, 참여 이슈 수를 증가
                        dash_board.put(s, dash_board.get(s) + 1);
                    }
                }
            }
            for (Map.Entry<String, Integer> entry : dash_board.entrySet()) // entry를 사용하면 key value 쌍을 갖고 있는 하나의 객체로 가져옴
            {
                double check = (double) entry.getValue() / issues.size() * 100; // 출석률 계산
                String formattedCheck = String.format("%.2f", check); // 소수점 둘째자리까지만 표시
                System.out.println("깃허브 아이디 : " + entry.getKey() + ", 출석률 : " + formattedCheck + "%");
            }
        }
        catch (IOException e) // 입출력 쓰면 이거 안하면 컴파일 오류 생김. 예기치 못한 오류 방지용
        {
            e.printStackTrace();
```

```
        }
    }
}
```

실습 02. LinkedList를 구현하세요.

LinkedList에 대해 공부하세요.

> 데이터를 저장할때, 노드(데이터와 다음 주소값이 하나로 담긴 단위)를 만들어서 그 다음 자료의 주소값을 노드에 저장해서 다음 노드로 가는 길을 만들어서 연결시키는 추상적 자료형
>
> 장점 : 배열에 비해서 데이터의 추가 삽입 및 삭제가 용이하다. (가운데에 끼워넣기 등)
> 연결된 주소값이 아니기 때문에 공간을 그때그때 설정하기때문에 메모리절약에 이점이 있다.
>
> 단점 : 하나하나 타고 가야하기때문에 인덱스접근이 불가능해서, 쭉 지나가면서 찾아와야하기 때문에 시간복잡도에서의 단점이 있다.
>
> 단일 연결리스트 : 큐(queue) 노드의 포인터가 다음 노드의 시작 주소값을 가르킴
>
> 이중 연결리스트 : 덱(deque) 노드 안에 포인터를 저장하는 공간이 앞과 뒤를 하나씩 가르키고 있음
>
> 자바의 연결리스트는 비동기적으로 구현되었기 때문에, 여러 스레드가 동시에 노드를 추가/삭제 할 수 없다. (외부적으로만 동기화 될 수 있다.)

정수를 저장하는 ListNode 클래스를 구현하세요.

```
public class ListNode
{
    private int data;     // 데이터 값
    public ListNode next; // 다음 주소값

    public ListNode(int data)
        {
        this.data = data;
        this.next = null; // 항상 끝에 null을 주어야 함
    }

    public int getData()  // 데이터를 불러오는 메서드
        {
        return data;
    }
}
```

ListNode add(ListNode head, ListNode nodeToAdd, int position)를 구현하세요.

ListNode remove(ListNode head, int positionToRemove)를 구현하세요.

boolean contains(ListNode head, ListNode nodeTocheck)를 구현하세요.

```
public class ListNode
{
    private int data;
    private ListNode next;

    public ListNode(int data)
        {
        this.data = data; // this란 객체 자신을 가리키는 레퍼런스 변수로 지
        this.next = null;
    }

    public ListNode() {
        this.data = 0;
        this.next = null;
    }

    public ListNode add(ListNode head, ListNode nodeToAdd, int posi
                {
            if (head.next == null)         // 첫번째 노드일 경우
                    {
                head.next = nodeToAdd;     // 다음 노드를 다음 주소값으
                nodeToAdd.next = null;     // 그 다음은 널로 바꿔준다.
                return head;
            }
        for (ListNode node = head; node.next != null; node = node.n
                { // 연결 리스트가 널일때까지
            if (position <= 0) // 넣어야 할 위치를 찾았을 경우
                        {
                nodeToAdd.next = node.next;
                node.next = nodeToAdd;
                break;
            }
                        else
                        {
                position--;
            }
        }
        return head;
    }

    public ListNode remove(ListNode head, int positionToRemove)
```

```
        {
        ListNode removedNode = new ListNode();
        ListNode prevNode = new ListNode();
        if (positionToRemove < 0) // invalid index 처리 + 사이즈보다 클
                {
            System.out.println("없는 인덱스");
            return null;
        }
                if (!head) // 연결리스트가 하나도 없을때 (null이 들어온 경
                {
            System.out.println("없는 연결리스트");
            return null;
        }
        prevNode = head; // 인자가 head 하나밖에 없을때도 처리하기 위해
        removedNode = head.next;
        while (positionToRemove-- > 0) // 인덱스 찾을 때 까지
                {
            prevNode = removedNode;
            removedNode = removedNode.next;
        }
        prevNode.next = removedNode.next; // 저번 인덱스의 다음노드를 지
        return removedNode;
    }

    public boolean contains(ListNode head, ListNode nodeToCheck)
        {
        for (ListNode node = head; node != null; node = node.next)
                {
            if (node.data == nodeToCheck.data) // 데이터가 있으면
                        {
                return true;
            }
        }
        return false;
    }
```

실습 03. 과제 3. Stack을 구현하세요.

int 배열을 사용해서 정수를 저장하는 Stack을 구현하세요.

void push(int data)를 구현하세요.

int pop()을 구현하세요.

```
public class Stack
{
    private int stack[];
    private int top;

    public Stack()
        {
        stack = null;
        top = 0;
    }

    public void push(int data)
        {
        top++;
        int temp[] = new int[top];
        for (int i = 0; i < top - 1; i++)
                {
            temp[i] = stack[i];
        }
        stack = temp;
        stack[top - 1] = data;
    }

    public int pop()
        {
        if (top < 1)
                {
            System.out.println("스택이 비었습니다.");
            return 0;
        }
        int removedNum = stack[top - 1];
        top--;
        if (top == 0)
                {
            stack = null;
            return removedNum;
        }
        int temp[] = new int[top];
        for (int i = 0; i < top; i++)
                {
            temp[i] = stack[i];
        }
```

```
        stack = temp;
        return removedNum;
    }
```

실습 04. 앞서 만든 ListNode를 사용해서 Stack을 구현하세요.

ListNode head를 가지고 있는 ListNodeStack 클래스를 구현하세요.

void push(int data)를 구현하세요.

int pop()을 구현하세요.

```
public class ListNode
{
    private int data;
    private ListNode next;

    public ListNode(int data)
        {
        this.data = data;
        this.next = null;
    }

    public ListNode()
        {
        this.data = 0;
        this.next = null;
    }

        public Stack()
        {
                this.head = new ListNode();
        }

    public void push(int data)
        {
              ListNode node = new ListNode(data);
        node.next = head.next;
        head.next = node;
    }

        public int pop()
        {
        if (head.next == null)
                {
```

```
            System.out.println("스택이 비었습니다.");
            return 0;
        }
        ListNode removedNode = head.next;
        int exitdata = removedNode.data;
        head.next = removedNode.next;  // 지운 노드의 다음 주소를 헤드로
        removedNode.next = null;  //  다음을 참조 할 수 없게 연결을 끊음
        return exitdata;
    }
}
```

실습 05. Queue를 구현하세요.

배열을 사용해서 한번.

```
public class Queue
{
    private int queue[];
    private int front;
        private int back;

    public Queue()
        {
        queue = null;
        front = 0;
                back = 0;
    }

    public void push(int data)
        {
        back++;
                if (queue == null)
                {
                        queue[0] = data;
                        return ;
                }
        int temp[] = new int[back - front + 1];
        for (int i = 0; i < back - front; i++)
                {
            temp[i] = queue[i];
        }
        queue = temp;
        queue[back - 1] = data;
```

```
    }

    public int pop()
        {
        if (back < 1)
                {
            System.out.println("큐가 비었습니다.");
            return 0;
        }
        int removedNum = queue[front];
        front++;
        if (front == back)
                {
            queue = new int[0];
                        front = 0;
                        back = 0;
            return removedNum;
        }
        int temp[] = new int[back - front];
        for (int i = 0; i < back - front; i++)
                {
            temp[i] = queue[i + 1];
        }
                back = back - front;
                front = 0;
        queue = temp;
        return removedNum;
    }
```

ListNode를 사용해서 한번.

```
public class ListNode
{
    private int data;
    private ListNode next;

    public ListNode(int data)
        {
        this.data = data;
        this.next = null;
    }

    public ListNode()
```

```
        {
        this.data = 0;
        this.next = null;
    }

        public Queue()
        {
                this.head = new ListNode();
        }

    public void push(int data)
        {
              ListNode node = new ListNode(data);
                if (!head)
                {
                        head = node;
                        node.next = null;
                }
        head.next = node;
        node.next = null;
    }

        public int pop()
        {
        if (!head)
                {
            System.out.println("큐가 비었습니다.");
            return 0;
        }
        ListNode removedNode = head;
        int exitdata = removedNode.data;
        head = removedNode.next;  // 지운 노드의 다음 주소를 헤드로
        return exitdata;
    }
}
```