목차

1:자바란 무엇인가

썬 마이크로시스템즈에서 1995 년에 개발한 객체 지향 프로그래밍 언어

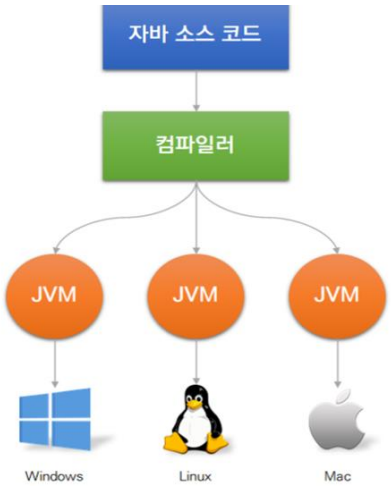
제임스 고슬링

자바의 특징

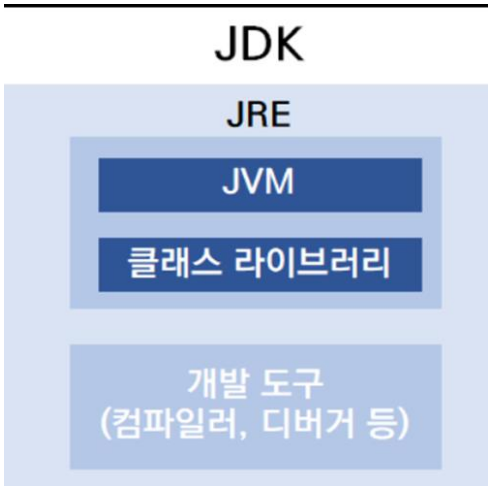
1.플랫폼 독립적: JVM 덕분에 사용하여 어느 플랫폼에서나 쉽게 사용이 가능하다. 다른 언어들은 운영체제에 따라서 코드가 조금씩 달라진다.

2.객체지향 언어: 자바는 C++과 파이썬같이 객체지향언어를 사용한다.
3.컴파일,인터프리터 언어: 자바를 자바 바이트 코드로 컴파일한 후에 JVM 에서 인터프리터 한다. JVM 을 한번 더 쳐야하기 때문에 조금 느리다는 단점이있다.
4.자동 메모리 관리, 멀티 쓰레딩 지원: 사용자가 메모리를 관리하는 것이 아닌 자동으로 메모리가 관리되고 사용하지 않는 메모리는 가비지 컬렉터가 자동으로 회수해 주기 때문에 편리하다. 한 프로세스에 한가지 스레드만 작동하는 것이 아니라 여러 스레드가 한 번에 작동할 수 있어 사용자 응답성이 향상되고 자원을 효율적으로 사용할 수 있다.

JVM? JRE? JDK?



JVM: 자바 가상머신(바이트 코드를 실행시키기 위한 머신)
자바는 운영체제에 독립적, 운영체제가 달라도 자바 프로그램을 실행 가능



JRE: 사용자의 컴퓨터에 자바 프로그램을 실행할 수 있도록 환경을 제공하는 설치 패키지

JDK= JRE + 여러 개발도구(디버거, 컴파일러)

2.객체지향(OOP)

객체지향: 프로그램을 다수의 객체를 만들고 객체가 기능을 가지며 서로 상호작용 하도록 만드는 프로그래밍 언어 ***모르면 암기***

절차지향: 물이 위에서 아래로 흐르는 것처럼 '순차적인 처리'가 중요시되며 프로그램 전체가 유기적으로 연결되도록 만드는 프로그래밍 기법이다.

유지보수성 + 재사용성

객체지향 4 대 원칙

캡슐화

서로 연관있는 속성과 기능들을 하나의 캡슐(capsule)로 만들어 데이터를 외부로부터 보호하는 것

정보은닉

접근 제어자 사용

public: 모두가 접근 가능

protected: 상속이나 같은 패키지 내 클래스에서 접근 가능

default: 같은 패키지 내 클래스에서 접근 가능

private: 본인만 접근 가능

상속화

기존의 클래스를 재활용하여 새로운 클래스를 작성하는 것.

재사용 =>extends

하위클래스가 상위클래스를 재사용하거나 확장한다.

인터페이스
-다중 상속 대신 도입
-해야 할 일을 정의하는 추상 자료

추상화

객체의 공통된 속성 중 필요한 부분을 포착해서 클래스로 정의하는 걸 추상화라고 한다.
추상 클래스와 추상 메소드를 이용한다.

- 클래스 VS 객체
  - **클래스** : 분류에 대한 **개념** -> 같은 특성을 지닌 여러 객체를 총칭하는 집합의 개념 (ex.사람)
  - **객체** : **실체** -> 유일무이한 사물 (ex. 소중한 GDSC 맴버들 하나하나가 객체)

다형성

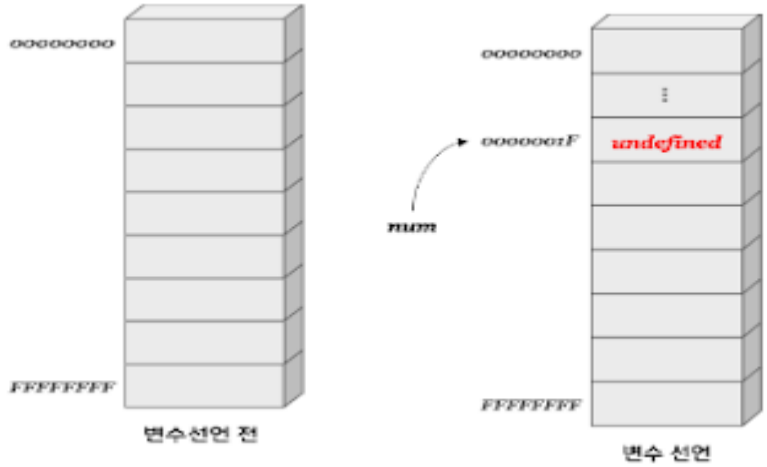
어떤 객체의 속성이나 기능이 상황에 따라 여러 가지 형태를 가질 수 있다. 자바에서는 한 타입의 참조변수를 여러 타입의 객체를 참조할 수 있도록 함으로써 다형성을 표현했다.
사용 편의 => 오버라이딩,오버로딩, 인터페이스, 객체
오버라이딩: 상속받은 자식클래스에서 메서드 재정의
오버로드: 같은 이름의 메서드 중복 정의

3.자바 기본문법

변수와 상수

변수를 한 문장으로 정리하면

"데이터의 저장과 참조를 위해 '할당된 메모리 공간'에 이름을 붙인 게 '변수'이다.



"정수 선언을 위해 메모리 공간을 할당하겠다."
"그 메모리 공간의 이름을 num 으로 하겠다"
다음 두 가지를 충족하는 문장은 아래와 같다.

```
int num;
```

여기서 int 는 정수를 저장할 메모리 공간을 할당받은 것이고
num 은 변수의 이름이다.

```
int num= 30;
int num= 22;
```

다음과 같이 변수는 값을 마음대로 초기화하고 변경할 수 있지만
상수는 한번 그 값이
정해지면 이후로는 변경이 불가능한 변수이다.
상수는 final 이라는 선언을 추가하여 사용한다.

```
final int num=10;
//int num =3;
```

*주석처리 된 부분처럼 상수의 값을 재정의하며 컴파일 오류가 뜬다.

변수의 타입(자료형)

java 의 자료형은 크게 Primitive type(기본자료형)과 Reference type(참조자료형)으로 구분된다. 기본 타입은 실제 값을 변수에 저장하지만 참조 타입은 메모리의 주소값을 변수 안에 저장합니다.

Primitive type(기본자료형)은 정수 자료형, 실수 자료형 문자 자료형으로 구분된다.

정수 자료형: byte, short, int, long 이 있으며 각각 크기가 다르고 그에 따른 메모리 할당량이 다르다. 각각 1 바이트 2 바이트 4 바이트 8 바이트가 할당된다.

```
int num = 10;
```

실수 자료형:float, double 는 소수점 이하의 값을 지니는 실수의 저장 및 표현을 하고 그 크기에 따라서 float 와 doubled 로 나뉜다.

```
float A= 0.1f;
```

* 실수 자료형을 사용할 때는 변수의 값에 f 나 d 를 붙여주어야 한다.

문자 자료형: char 이 있으며 문자의 저장을 위한 자료형이고 자바는 유니코드를 기반으로 문자를 처리한다.

```
char gdsc ='굿';
```

논리 자료형: boolean 은 자바에서 '참'과 '거짓'의 표현을 목적으로 하는 '논리 자료형'이다.
'true'나 'false'의 값만 가진다.
`boolean B = true;`

	타입	할당되는 메모리 크기	기본값	데이터의 표현 범위
논리형	boolean	1 byte	false	true, false
정수형	byte	1 byte	0	-128 ~ 127
	short	2 byte	0	-32,768 ~ 32,767
	int(기본)	4 byte	0	-2,147,483,648 ~ 2,147,483,647
	long	8 byte	0L	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
실수형	float	4 byte	0.0F	(3.4 X 10-38) ~ (3.4 X 1038) 의 근사값
	double(기본)	8 byte	0.0	(1.7 X 10-308) ~ (1.7 X 1038) 의 근사값
문자형	char	2 byte (유니코드)	'\u0000'	0 ~ 65,535

참조형 변수

참조형 변수는 기본형 변수처럼 정해져 있는 게 아니라 사용자가 원한다면 자기가 원하는, 필요한 변수형을 만들어서 사용할 수 있다. 객체, String,Scanner 등이 있다.
1) String 기본적으로 자바에서는 문자열을 처리할 수 있는 자료형이 존재하지 않는다. 기본형 8 가지 중에서 **'문자'**를 처리하는 **char** 만 있지 **문자'열'**을 처리할 수 있는 것이 없다. 이 녀석은 **문자열**을 처리하는데 사용하는 변수다.
2) Scanner
Scanner 는 **사용자의 입력을 받아야 할 때** 사용하는 참조형 변수다.너무나도 당연하지만 기본형 변수에서 사용자의 입력을 받는 참조형 변수이다.

연산자

연산자의 종류는 아래의 표와 같다.

종류	연산자	우선순위
증감 연산자	++, --	1순위
산술 연산자	+, -, *, /, %	2순위
시프트 연산자	>>, <<, >>>	3순위
비교 연산자	>, <, >=, <=, ==, !=	4순위
비트 연산자	&, |, ^, ~	~만 1순위, 나머지는 5순위
논리 연산자	&&, ||, !	!만 1순위, 나머지는 6순위
조건(삼항) 연산자	?, :	7순위
대입 연산자	=, *=, /=, %=, +=, -=	8순위

구분	연산자	설명
증감 연산자	x++	먼저 해당 연산을 수행한 후 피연산자의 값을 1 증가 시킴
	++x	먼저 피연산자의 값을 1 증가 시킨 후 해당 연산을 수행함
	x--	먼저 해당 연산을 수행한 후 피연산자의 값을 1 감소 시킴
	--x	먼저 피연산자의 값을 1 감소 시킨 후 해당 연산을 수행함
구분	연산자	설명
산술 연산자	+	두 수에 대한 덧셈
	-	두 수에 대한 뺄셈
	*	두 수에 대한 곱셈
	/	두 수에 대한 나눗셈
	%	두 수를 나눈 후 그 나머지를 반환한다.

구분	연산자	설명

시프트 연산자	>>	bit 값을 오른쪽으로 이동 (빈 자리는 부호값으로 대입) 한다.
	<<	bit 값을 왼쪽으로 이동 (빈 자리는 0 으로 대입) 한다.
	>>>	bit 값을 오른쪽으로 이동 (빈 자리는 0 으로 대입) 한다.

구분	연산자	설명
비교 연산자	>	크다
	<	작다
	>=	크거나 같다
	<=	작거나 같다
	==	피연산자들의 값이 같다
	!=	피연산자들의 값이 같지 않다

구분	연산자	의미	설명
논리 연산자	&	and (논리곱)	주어진 조건들이 모두 true 일 때만 true 를 나타낸다.
	|	or (논리합)	주어진 조건들 중 하나라도 true 이면 true 를 나타낸다.
	!	not (부정)	true 는 false 로 false 는 true 로 나타낸다.

구분	설명
&&	선조건이 true 일 때만 후조건을 실행하며 선조건이 false 이면 후조건을 실행하지 않는다.
||	선조건이 true 이면 후조건을 실행하지 않으며 선조건이 false 일 때만 후조건을 실행한다.

구분	연산자	설명

대입 연산자	=	연산자를 중심으로 오른쪽 변수값을 왼쪽 변수에 대입한다.
	+=	왼쪽 변수에 더하면서 대입한다.
	-=	왼쪽 변수값에서 빼면서 대입한다.
	*=	왼쪽 변수에 곱하면서 대입한다.
	/=	왼쪽 변수에 나누면서 대입한다.
	%=	왼쪽 변수에 나머지 값을 구하면서 대입한다.
구분	연산자	설명
비트 연산자	&	비트 단위의 AND
	|	비트 단위의 OR
	^	XOR (배타적 OR)
	~	단항 연산자 이며, 비트를 반전한다. 0 은 1 로 1 은 0 으로 만듦

조건문

조건문은 크게 if 문과 switch 문으로 나뉩니다. if 문은 if 문,if else 문, else if 문으로 다시 나뉩니다.

if 문 구조

```
if(조건식){
    //조건식이 참이면 실행
}
```

if 문의 조건식이 참이면 괄호 안에 있는 문장들이 실행되고 거짓이면 if 문을 빠져 나온다.

```
int num = 5;
if(num>3){
    num++;
}
System.out.print("num 의 값은 "+num);
```

if else 문 구조

```
    if(조건식){
     //조건식이 참이면 실행
    }
    else{ // 조건식이 거짓일때 실행
    }
```

if 문이 거짓일 때 else 문이 실행되고 둘 중 하나만 실행된 후 if-else 문을 빠져나온다.

else if 문 구조

```
    if(조건식 1){
    //조건식 1 이 참이면 실행
    }
    else if(조건식 2){
    //조건식 2 가 참이면 실행
    }
    else {
    //조건식 1,2 모두 거짓일 때 실행
    }
```

이렇게 else if 를 사용해서 여러 개의 조건식을 사용하는 다중 조건문을 사용할 수 있다.

```
    if (score>=90){
     grade='A';
    } else if(score>=80) {
     grade='B';
    } else if(score>=70) {
     grade='C';
    }else {
      grade='F';
    }
```

위의 예제는 else if 문을 여러개 사용해서 score 를 통해 grade 를 알아내는 예제이다.

switch 의 구조

```
  switch(변수){
    case 값 1:
    실행문 1;
    break;
    case 값 2:
    실행문 2;
    break;
    ------
    default: //만족하는 값이 없을 때 <default 부분 생략 가능>
    실행문;
    break;
    }
```

변수와 case 문의 값이 일치하면 해당 처리할 문장이 실행되며, 문장 실행 후 break 문을 만나면 switch case 문을 종료한다. default 문은 case 문에 해당하지 않는 조건일 때 실행된다.
switch 를 사용할 때 break 문 중요!

```
char a='g';
String name;
name= "";
switch(a) {
    case 'g':
        name+='g';
    case 'd':
        name+='d';
    case 's':
        name+='s';
    case 'c':
        name+='c';
        break;
}
System.out.print(name);
```

위의 예제와 같이 break 문을 사용하지 않으면 break 문이 나올때까지 switch 문은 실행된다.

반복문

반복문에는 크게 for 과 while 이 있다.
while 문은 증감식이 있는 for 문과 다르게 무한루프에 빠질 수 있다.

for 문 구조

```
for(초기식; 조건식; 증감식){
```

```
//조건이 참이면 실행
}

int i=0;
for(int a=0; a<10; a++){
    i+=a;
}
System.out.print(i);
```

위의 예제처럼 for 문을 사용하면 된다. for 문을 여러번 사용하는 중첩 for 문을 이용하여 별 찍기도 많이하니 한 번쯤 해보는 것도 좋다.

while 문 구조

```
while (조건문) { <수행할 문장 1>; <수행할 문장 2>; <수행할 문장 3>; ... }
```

while 문은 조건문이 참이면 수행할 문장들을 반복한다.

```
int i=0;
int a=0;
while(a<10){
    i+=a
    a++//이 문장이 없다면 무한루프에 빠진다.
}
System.out.print(i);
```

위의 예제는 for 문의 예제를 while 문으로 바꾼 것이다.

배열

배열은 하나의 블록 안에여러 데이터를 모아 집합시켜 저장함으로써 데이터를 구조적으로 다루는 데 사용한다. 배열을 구성하는 각각의 값을 배열 요소(element)라고 하며, 배열에서의 위치를 가리키는 숫자를 인덱스(index)라고 칭한다.

선언방법	예시
타입[] 변수이름;	int[] score; String[] name
타입 변수이름[];	int score[]; String name[

```
    String[] beer = {"Kloud", "Cass", "terra", "Guinness"};
    // 인덱스 번호 : 0 , 1 , 2 , 3      System.out.println(beer[0]); //
Kloud      System.out.println(beer[1]); // Cass      System.out.println(beer[2]); //
terra      System.out.println(beer[3]); // Guinness
```

위의 예제처럼 인덱스 번호는 0 부터 시작한다.

```
    int [] score = { 75, 80, 95, 90};
    int sum=0;
    for(int a=0; a<score.length; a++){
        sum += score[a];

    }
    System.out.print(sum);
```

위의 예제는 for 문과 배열을 사용해 score 에 있는 값들의 합을 구한 것이다. 이렇게 연관된 데이터들을 모아서 변수의 선언을 줄이고, 반복문을 사용해서 계산 과정을 줄이 것이 배열의 장점이다.

클래스와 객체

클래스는 객체를 정의해 놓은 것이고 객체는 실제로 존재하는 것 ,사물과 개념이라고 정의 할 수 있다. 쉽게 말해 클래스가 TV 설계도라면 객체는 TV 라고 비유할 수 있다.
클래스에서 객체를 만드는 방법을 알아보겠다.

```
public class Hello2030{

        public static void main(String[] args) {

                tansan a= new tansan(); // tansan 의 클래스를 a 참조변수에 인스턴스화 했음.                  인스턴스화 하면 자동으로 기본생성자가 호출된다.
                a.name="콜라"; //맴버변수에 데이터를 저장
                a.price=1500;
                a.showPrice();//맴버 함수 호출
        }
}

 class tansan { //tansan 클래스 생성
        String name; //필드에 맴버변수를 생성함
        int price;

        public tansan() { //기본 생성자라고 부르며 클래스에는 하나의 이상의 생성자가 있어야한다.  만약 클래스 내에 매개변수가 있는 메서드가 없다면 컴파일러가 자동으로 기본 생성자를 만들어 준다.
        }
        public void showPrice() {
                System.out.print(name+"의 가격은 " + price + "원 입니다.");
        }
 }
```

위와 같은 방법으로 인스턴스를 생성하면 되고 어려운 부분이자 객체 지향언어의 기본이되는 부분이기에 잘 알아두어야 한다.

상속

상속은 자식클래스가 부모클래스의 필드와 메소드를 물려받는 것이다. 상속받은 클래스가 하위클래스, 자식클래스 또는 서브클래스이고 상속해 주는 클래스가 상위 클래스, 부모클래스 또는 슈퍼클래스이다.

```
class (자식)클래스명 extends (부모)클래스{...}
ex) class A extends B{..}
```

위처럼 extends 를 이용하여 상속할 수 있다.부모 클래스는 자식 클래스에서 정의한 메소드나 필드를 사용하지 못한다.(자식 = 자신 + 부모 / 부모 = 자신)

오버라이딩

오버라이딩은 부모클래스에서 상속받은 메서드의 내용을 재정의하는 것이다.

오버라이딩에는 몇가지 조건이 있는데
매서드의 이름이 같아야 한다.
메서드의 반환타입과 매개변수가 같아야 한다.

```
class drink{
        void coke() {
        }
        void soda() {
        }
}
class tansan extends drink{
      void coke() {
              System.out.print("안녕");
```

```
        }
    }
```
위의 예제처럼 자식클래스에서 부모클래스의 coke 메서드를 오버라이딩해봤다.

오버로딩

오버로딩은 이름이 같은 메서드를 중복해서 선언하는 것을 말한다.
오버로딩의 조건은 메소드의 이름이 같아야 하고 매개변수의 개수나 타입이 달라야 한다.

```
    String ai(Integer a){
    // Integer 는 int 의 객체형(Wrapper 클래스)이다.
        return a.toString();
    //toString()은 객체의 정보를 문자열로 출력하는 메소드이다.
    }
    int ai( int a){
        return a;
    }
    int ai( int a, int b){
         return b;
    }
    void ai() {
        System.out.print("호출되었습니다.");
    }
```
위의 예제와 같이 오버로딩을 사용하면 같은 기능을 하는 메소드를 하나의 이름으로 사용할 수 있고 메소드의 이름을 절약할 수 있는 장점이 있다.
*return 은 현재 메서드를 종료하고 호출했던 메서드로 돌아가는 역할을 해서 모든 메서드는 return 값이 있어야 한다. 예외로 void 는 컴파일 시 return 을 자동으로 추가해서 return 을 생략할 수 있다.

오버로딩, 오버라이딩 예제

```
class drink{
        void coke() {
        }
        void soda() {
        }
}
class tansan extends drink{
      void coke() {   //오버라이딩
              System.out.print("안녕");
      }
int coke(int a) { //오버로딩
         return a;
}
 void soda(){//오버라이딩
}
 void fanta(){
}
void fanta(char a,char b){//오버로딩
}
 }
```

추상클래스

추상클래스는 조상클래스에 추상 메서드만 선언해 놓는 것을 의미한다. 추상 메서드는 초기화는 안되어 있고 메서드의 선언만 되어있는 것이다. 추상클래스와 추상메서드의 선언은 abstract 를 사용하면 된다.

```
abstract class unit{ //추상클래스
      abstract void move(int a, int b); //추상메서드
      void power() {
```

```
        }
}
    class tank extends unit{
        @Override
        void move(int a,int b){ //오버라이딩
            System.out.println(a+b);
        }
        void attack() {

        }
    }
```

위의 예제처럼 abstract 를 사용해서 추상클래스와 추상메서드를 만들었다.
추상클래스는 실체가 없기 때문에 void 를 사용하고 {}대신 ;를 사용해서 문장을 끝내준다. 추상클래스를 상속받은 클래스는 추상 메서드를 반드시 오버라이딩을 해야 한다.
추상클래스는 설계도라고 생각하면 되고 자식클래스에서 설계도에 수행될 내용을 적는다고 생각하면 쉽다.

인터페이스

인터페이스는 객체의 특정 행동의 특징을 정의하는 간단한 문법입니다. 일종의 추상클래스입니다.

```
    interface repair{
        public static final int 타입상수이름 =값; //public static final 생략가능
        public abstract void 메서드이름(매개변수); //public abstract void
생략가능
        int b=0;
        void c();
    }
```

인터페이스는 추상클래스와 달리 상수와 추상메서드를 사용해야한다. 또한 접근제어자와 static 메서드도 사용가능하다.

인터페이스를 상속받을 때는 확장한다는 뜻의 extends 와 달리 구현한다는 뜻의 implements 를 사용한다.

```
class human {

}

interface hero{
    public   abstract void attack();
}

class ironman extends human implements hero{
    @Override
    public void attack() { // 자식메서드는 부모메서드의 접근 제어자보다
크거나
    같아야 하기 때문에 public 을 써야 한다.
    }
    void rocket() {

    }
}
```

위의 예제는 ironman 클래스가 implements 를 이용해서 hero 인터페이스를 구현했다.
추상클래스와 마찬가지로 추상메서드를 오버라이딩해야한다.
인터페이스는 상속과 동시에 구현이 가능하고 인터페이스끼리는 다중 상속도 가능하다.

이제부터 기본적인 자바 문법이 아니라 스프링 프레임워크 사용 시 필요한 자바의 고급과정입니다. 한 번에 이해하려 하지 말고여러 번

전체적으로 "반복"하면서 눈에 익히는 식으로 공부하는게 좋습니다.
실습을 해보고 어떻게 언제 사용하는지를 각각 숙지하셨으면
좋겠습니다.

예외처리

자바에는 우리가 컴파일할 때 발생할 수 있는 컴파일 오류, 실행 중 발생되는 런타임 오류와 실행은 되지만 의도했던 것과 다르게 동작하는 논리적 에러 이렇게 세 종류가 있다. 그중 우리가 알아야 할 것은 런타임 오류이고 에러(Error)와 예외(Exception)로 나뉜다. 에러는 메모리 부족과 같이 발생하면 복구하기 까다롭고 예외는 발생하더라도 수습이 비교적 쉽다. 예외 처리는 런타임 중에 발생할 수 있는 예외들을 미리 막아 프로그램이 정상 작동할 수 있게 하는 것이다.

```
int a=0;
int b=12;
System.out.print(b/a);
```

위의 예제를 실행하면 Exception in thread "main" java.lang.ArithmeticException:이라는 오류가 발생한다. 이때 try-catch 라는 예외처리를 사용하면 된다.

```
int a=0;
int b=12;
try { //예외가 발생할 문장에 사용
   System.out.print(b/a);
}
catch(Exception e) { // 예외가 발생했을 시 catch 안의 문장이 실행
    System.out.println("예외발생!");
}
finally { //예외가 발생하건 발생하지 않건 공통으로 수행되어야 할 코드가 쓰임. 생략 가능
    System.out.println("난 반드시 실행돼");
}
```

위의 코드처럼 try-catch 문을 사용하여 예외 처리를 할 수 있다. catch(Exception e)이 부분에는 Exception 대신 오류 메시지에 나타났던 ArithmeticException 을 써도 되나 자바에서 **모든 예외는 Exception 이라는 클래스를 상속**받기 때문에 Exception 을 사용해도 무방하다.

try-catch 문에는 흐름이 있는데 try 블럭 내에서 예외가 발생하면 발생한 예외와 일치하는 catch 블럭이 있는 확인 한다. 찾게 된다면 그 catch 문 내의 문장들을 수행하고 전체 try-catch 문을 빠져나가 그 다음 문장을 수행한다. try 블럭 내에서 예외가 발생하지 않으면 catch 문을 거치지 않고 try-catch 문을 빠져나간다.

```
try {

    }
catch(ArithmeticException e) {
```

```
        e.getMessage();
        e.printStackTrace();
        }
```

위의 문장처럼 **Exception 클래스에는** printStackTrace(), getMessage() 메서드들이 있다.
printStackTrace()는 예외가 발생한 부분의 메서드 정보와 메시지를 화면에 출력한다.
getMessage()는 예외가 발생한 클래스의 인스턴스에 저장된 메시지를 얻는다.

예외처리에는 try-catch 문 만 있는 게 아니라 고의로 예외를 발생시키는 throw 문과 자신을 호출하는 메소드에 예외처리의 책임을 넘기는 throws 문도 있다.

```
public class Main1 {
    public static void main(String[] args) {
        try {
            zero(10, 0);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException 발생: " + e.getMessage());
        }
    }
    public static void zero(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("0 으로 나눌 수 없습니다.");
        }
        int result = a / b;
        System.out.println("나눈 결과: " + result);
    }
}
```

throws 는 자신을 호출한 메서드에서 대신 예외처리를 하게 한다. 그렇기 때문에 throws 가 있는 메서드를 호출할 때에는 try-catch 문 안에서 호출하고 throws 는 호출하는 쪽에 이런 예외가 나올 수 있다라는 것을 알리기 위한 용도이다.
throw 를 통해 일부러 예외를 발생시켰다. throw 를 통해 프로그래머가 고의로 예외를 발생시킬 수 있다.

Annotation

어노테이션은 자바의 소스코드와 문서를 한 번에 저장하기 위해 만든 것이다. 과거에는 소스코드와 파일 관리방법 문서가 따로 있어서 소스코드를 업데이트할 때마다 따로따로 업데이트 해줘야 했는데 파일 크기가 커지면 커질수록 번거로웠다. 이 문제점을 해결하기위해 어노테이션을 사용한다. 어노테이션에는 표준 어노테이션과 메타 어노테이션이있다.

자주 사용하는 표쥰 어노테이션에 다음 세 가지를 알아보겠다.

1. @Override
2. @Deprecated

3. @SuppressWarnings

@Override 는 컴파일러가 오버라이드를 똑바로 했는데 체크하는 것이다.

```
class Parent{
	void parentMethod(){}
}

class Child extends Parent{
	@Override
   void pparentmethod(){} // 컴파일 에러! 잘못된 오버라이드 스펠링 다름
```

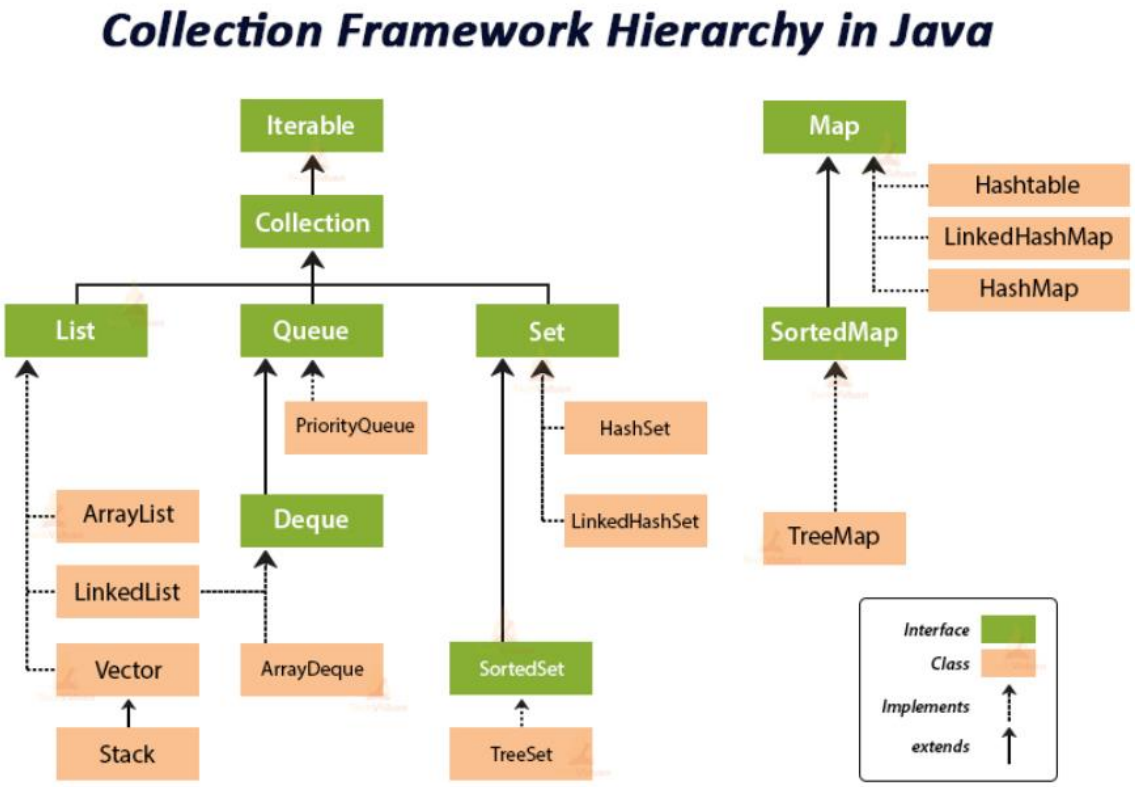
이런 식으로 오버라이드하려고 했지만, 스펠링을 다르게 적으면 컴파일할때 새로운 메서드를 만드는줄 알고 에러가 뜨지 않는다. 이러한 실수를 줄이기 위해 @Override 어노테이션을 사용한다.

@Deprecated 는 더 이상 사용하지 않는 메서드를 가리키는 것이다. 과거에는 사용했지만, 더 빠르거나 성능이 좋은 메서드를 만들었을 경우에 사용한다. 메서드를 지우면 되겠지만 어느 클래스에 어떻게 사용됐을지 모르는 메서드를 지우는 것은 위험하기 때문에 @Deprecated 메서드를 사용한다.

@SuppressWarnings 는 컴파일러가 일반적으로 경고하는 내용 중 "이건 하지마"라고 제외 시킬 때 사용한다.

컬렉션 프레임웍

컬렉션(collection): 여러 객체(데이터)를 모아 놓은 것을 의미
프레임웍(frame work): 표준화, 정형화된 체계적인 프로그래밍 방식
컬렉션 프레임웍: 컬렉션을 쉽고 편리하게 다룰 수 있는 다양한 클래스를 제공



위의 컬렉션 상속도에서 볼 수 있듯이 컬렉션에는 List,Set,Map,Queue 라는 큰 인터페이스들로 나뉘고 각각의 클래스들이 있습니다. 인터페이스들끼리의 사용 방법, 인터페이스의 메서드, 사용하는 이유 등.. 너무 방대하다 보니 이번시간에는 전부 다루지는 못한다.

```
//ArrayList 객체 생성
ArrayList<String> in = new ArrayList<String>();
                in.add("sangjin"); // sangjin 이라는 요소 추가
                in.clear(); //요소 지우기
//HashSet 객체 생성
Set<Integer> num= new HashSet<Integer>();
                  num.add(20); // 20 이라는 요소 추가
//HashMap 객체 생성
Map<String, Integer> hash = new HashMap<>();
                hash.put("학번",20); //key 값과 value 값 입력
```

간단하게 예제들로 컬렉션으로 객체를 생성하고 요소를 추가하는 방법을 알아보았다.
이외에도 무수히 많은 컬렉션 프레임웍이 있으니 스스로 공부!

Lambda

람다식은 간단하게 말해서 메서드를 하나의 식으로 표현한 것이다. 람다식은 익명함수로도 불린다. 람다식은 메서드의 매개변수로 전달되어지는 것이

가능하고, 메서드의 결과로 반환될 수도 있다. 람다식으로 인해 메서드를 변수처럼 사용할 수 있게 되었다.

```
반환타입 메서드이름(매개변수 선언) -> {
        문장들
}
```

람다의 구조는 메서드의 이름과 반환 타입을 제거하고 매개변수 선언부와 {}사이에 ->를 넣어준다. 또한 람다식은 return 문 대신 '식'으로 대신 할 수 있다. 이때는 문장이 아니라 '식'이기 때문에 ;는 붙지 않는다. 또한 매개변수의 반환타입 또한 생략이 가능하다.

```
int square(int x) {
        return x*x;
}
-----------------------------
```

```
x->x*x
```

위의 예제를 아래의 식으로 간단하게 나타낼 수 있다. 이렇게 식을 간결하게 적을 수 있는 장점이 있어 람다식을 사용한다. 무분별한 사용은 가독성을 해치기 때문에 조심하도록 하자.

함수형 인터페이스

함수형 인터페이스는 단 하나의 추상 메서드만 선언된 인터페이스이고 람다식을 다루기 위해 사용한다.

```
@FunctionalInterface
interface Myfunction{
        public abstract int max (int a,int b);
}
```

위의 예제 처럼 함수형 인터페이스는 @FunctionalInterface 라는 어노테이션을 함께 사용하면 해당 인터페이스가 함수형 인터페이스 조건에 맞는지 검사해 준다.

```
    public static void main(String[] args) {
        /*Myfunction f = new Myfunction(){
            public int max(int a, int b){
                return a>b?a:b; }
            }; */
            //람다식을 다루기 위한 참조변수의 타입은 함수형 인터페이스로한다.
                Myfunction f = (a,b)-> a>b?a:b; //위의 주석처리된 식을 람다식으로 변환.
        // #함수형 인터페이스와 반환타입과 개수가 맞다면 내용은 바꿔서 사용할 수 있다.
```

```
                int value = f.max(3,5);
                System.out.println(value);
        }

}
@FunctionalInterface
interface Myfunction{
        public abstract int max(int a, int b);
}
```

위의 주석 처리된 문장은 람다식을 사용하면 한 줄로 정리될 수 있기 때문에 람다식을 사용하고
람다식을 다루기 위해서는 참조변수의 타입은 함수형 인터페이스여야 한다.
람다식의 매개변수 개수와 반환타입이 인터페이스의 추상메서드 매개변수 개수와 반환타입이 같아한다.

```
/*Myfunction f = new Myfunction(){
                public int max(int a, int b){
                    return a>b?a:b; }
            };
```

위의 예제의 부분은 익명 클래스가 사용된 것이다.

익명 클래스 (Anonymous Class)

-new 생성자(){...}의 구조이다.
- 생성자 뒤에 중괄호가 나오고 코드를 오버라이딩하여 보통 구현한다. - 재사용하지 않고 특정 부분에서만 수행할 때 사용한다.
인터페이스나 추상클래스는 인스턴스가 될 수 없기 때문에 상속받는 클래스나 구현받는 클래스로 인스턴스를 만들어야한다. 하지만 추상클래스로 인스턴스를 하고 싶을 때 '추상클래스를 상속받고 있다는 뜻'의 익명 클래스를 만드는 것이다.

스트림

데이터의 연속적인 흐름, 다양한 데이터 소스를 표준화된 방법으로 다루기 위한 것이다.
스트림의 구조는 스트림 생성, 중간 연산, 최종연산으로 나뉩니다.

중간 연산 - 연산결과가 스트림인 연산 . 반복적으로 적용가능
최종 연산 - 연산결과가 스트림이 아님. 단 한번만 적용 가능 (스트림의 요소를 소모)
지연(lazy)되었던 모든 중개 연산들이 최종 연산 시에 모두 수행

스트림 생성

배열 일 때

```
int[] num1 = {1,2,3,4};
```

```
        IntStream mum = Arrays.stream(num1);
```

```
컬렉션 일 때
List<Integer> number1 = Arrays.asList(3,3,3,4,5);
        Integer num = number1.stream();
```

```
IntStream intStream = IntStream.range(1, 5); //1~4 까지 요소를 가짐
```

위의 예제이외에도 다양한 생성 방식들이 존재한다.

중간 연산

```
List<Integer> number3 = Arrays.asList(3,3,4,5,3,8,2);  //요소
        Integer num3 = number3.stream()
                .filter(number -> number.equals(3))//필터(filter)는 요소들을
하나씩 평가
                .mapToInt(Integer::intValue) //맵(map)은 요소들을하나씩 특정
값으로 변환
                 .sum(); //요소들의 합
```

위의 예제를 보면 여러 메서드 호출을 하나의 식으로 표현하고 있습니다. 이를 메서드 체이닝이라 하고 코드를 더 간결하고 가독성이 높게 만들어 줍니다.
중간 연산에서 사용하는 모든 것은 다루지는 못해서 자주 사용하는 map 과 filter 만 알아보았습니다.

최종 연산

스트림은 최종 연산 전까지 중간 연산을 수행하지 않습니다.

1. 요소의 출력 : forEach()
2. 요소의 소모 : reduce()
3. 요소의 검색 : findFirst(), findAny()
4. 요소의 검사 : anyMatch(), allMatch(), noneMatch()
5. 요소의 통계 : count(), min(), max()
6. 요소의 연산 : sum(), average()
7. 요소의 수집 : collect()

이러한 최종 연산을 사용해서 스트림의 연산을 마무리 지어야합니다.
여기서 끝나는게 아니라 스트림을 출력하는 방법 또한 다양하니 구글과 함께 알아보도록 합시다.

Optional

옵셔널을 null 값으로 인한 NPE 를 발생하지 않도록 사용한다.
NPE: NullPointerException 의 줄임말으로, **null 값을 가진 객체를 참조**하려고 했을 때 일어나는 예외이다.

옵셔널 클래스
옵셔널 클래스는 모든 타입의 객체를 담을 수 있는 래퍼 클래스이다.

```
Optional<String> optsj = Optional.of("135");
Optional<String> optsj = Optional.ofNullable(null);
```
이런 식으로 객체를 생성하려면 of 메서드와 ofNullable 메서드를 사용한다.
ofNullable 메서드는 참조변수의 값이 null 일 가능성이 있으면 사용한다.
```
Optional<String> emptyopt = Optional.empty(); //값이 없는 경우
System.out.println(optsj.isPresent());//참조변수의 값이 null 이면 false 아니면 true 를 반환
System.out.println(optsj);// 출력값 Optional[135]
System.out.println(optsj.get());//출력값 135 - 출력 값만 나타냄
```

마지막으로 옵셔널 스트림과 비슷하게 여러 메서드를 연결할 수 있는 메서드 체이닝이 가능하다.
```
List<String> it   = Arrays.asList(
                                "C","Java","C++","Python","Kotlin" );
            Optional<List<String>> itlist = Optional.of(it);
            int sum = itlist
                                .map(List::size)
                                .orElse(0);
            System.out.println(sum);
```

모든 코드에 적극적으로 주석 달기

1. 간단한 기본 문법 문제
정수 n 을 입력 받고 높이가 n 인 별탑을 출력하는 문제
예시 n = 4
```
      *
    ***
  *****
*******
```

2. 간단한 클래스 문제
필드 : name, age
메서드 : introduce() // “제 이름은 name 이고 나이는 age 입니다.”
위 필드와 메서드를 가지는 클래스 Person 을 만들고

Person 객체 2 개를 만들고 두 Person 객체를 자기소개 시키는 프로그램

3. 상속 문제
위에서 만든 Person 클래스를 상속 받는 Student 클래스를 만들고
원하는 필드를 만들고 Person 의 자기소개 메소드에서 사용할 수 있도록
오버라이딩하기 (예시: 제 이름은 name 이고 나이는 age, 학점은 a 입니다.)
Student 객체 하나를 만들어 만든 메서드를 하나 수행하는 프로그램

4. 컬렉션 문제
ArrayList<Person> 객체 하나를 생성하고
마음대로 Person 객체를 만들고 3 개 추가
ArrayList 객체에 든 요소를 모두 출력하는 프로그램

+ 추가 문제 (스트림)
위에서 만든 ArrayList 객체에 든 요소 중에서
age 가 20 이상인 요소만 출력하는 문장 추가

5. 예외처리, Optional 문장 예제 코드 주석달기
블로그에 있는 예제 코드에 주석 달기