https://imasoftwareengineer.tistory.com/73?category=769427

# Java

## 패키지(Package)란 무엇인가?

자바 패키지는 자바 클래스를 여러 namespace로 정리하고, 패키지에 소속된 각 class에게 고유한 namespace를 제공한다.

장점

1. 유지보수 : 관련 있는 class들을 패키지로 묶으면 특정 class의 위치를 찾을 때 더 빠르게 찾을 수 있다.
2. NameSpace 기능 : 동일한 이름의 파일에 대해서 namespace 기능을 제공한다.

# namespace란?

동일한 이름의 파일을 구분 할 수 있게 해주는 공간을 의미한다.  
( ex – A.B.C.java, B.D.C.java )

# 빌트 인 패키지(Built-in Package)

자주 쓰이는 패키지는 import를 하지 않아도 사용할 수 있도록 만들어져 있다.

(Ex – java.lang)

## 접근 제어자

객체 지향 디자인/개발 방법의 원칙 중에는 캡슐화라는 항목이 있다.

캡슐화는 어떤 변수나 메서드의 접근을 최대한 제한하는 것.

즉, 다시말해 공개하면 안되는 클래스/변수/메서드, 공개하지 않는 것이 좋은 클래스/변수/메서드 등은 공개하지 않는 것이 캡슐화이다.

접근제어가 없어서 코드를 짜고 코드를 패키지로 만들어서 배포되어 패키지가 제공하는 모든 변수와 메서드가 공개되어 있다면 보안 취약점 문제나, 비즈니스 로직의 상당 부분이 공개되는 문제가 발생할 수 있다.

Public : 같은 프로젝트

Default : 같은 패키지

Protect : 자기 자신과 자기를 상속하는 자식 클래스에게만 변수/메서드를 공개

Private : 같은 class 만

## Static (변수, 메서드, 중첩 클래스)

자바 프로그램을 시작하면 JVM이 클래스 파일을 읽어 필요한 클래스와 변수/메서드들을 메모리에 생성한다.

클래스 – static 변수 – MetaSpace 공간에 생성 / 클래스에 귀속 (단 1개 씩만 생성)

인스턴스 – 평범한 변수 – Heap 공간에 생성 / 오브젝트에 귀속 (오브젝트 개수만큼 생성)

# main 메서드

Public static void main(String[] args)라는 형식을 가진 메서드를 자바가 가장 먼저 실행 시킨다는 특징이 있다.

언제 사용하는가?

1.프로젝트가 하나의 변수를 공유해야 하는 경우 static 변수를 사용한다.

2.오브젝트의 멤버 변수에 구애받지 않는 경우에 오브젝트의 생성 없이 메서드를 사용하고 싶은 경우 static 메서드를 사용한다.

## 자바 객체지향 핵심 개념 내용 정리 필요

## 상속

목적 : 코드의 재사용

### This, super

This : Class 내에서 this란 인스턴스가 생길 당시의 자기 자신 인스턴스를 가르키는 키워드이다.

Super : 자식Class가 상속하는 부모 Class의 instance를 가르킨다.

외부에서 부모 클래스를 상속받는 객체를 생성할 때, 자식 class의 생성자에서 ‘super()’ 키워드로 부모 class의 생성자를 호출하여 부모 class의 인스턴스가 먼저 생성된다.

### New : Class의 생성자를 호출하는 역할을 하는 키워드. 따라서 객체 생성할 때 ‘new 클래스명()’ 으로 인스턴스를 생성할 수 있었던 것

## 추상 클래스, 추상 메서드

핵심 키워드 : 반드시 기능이 달라야 하는 Super 클래스의 메서드를 어떻게 개발자에게 구현하도록 강요할 수 있을까? 에 대한 해답 기능임.

추상클래스 : 객체화 되면 안되는 클래스일 경우 추상 클래스 자체가 인스턴스화 되지 못하도록 막는다.

추상메서드 : 자식클래스에서 기능 구현을 강제하기 위한 메서드인 경우

현실세계의 추상적인 개념과 기능을 표현하기 위해 자바에서는 추상 클래스 / 추상 메서드를 제공한다.

왜냐하면 현실 세계에 존재하지 않는 ‘추상적인 개념’을 위해 존재하는 class가 오브젝트로 생성되지 않게 하기 위하여이다.

Ex) Animal(동물) 클래스는 특정 군을 카테고리화 하기 위해 만들어진 범주이다.’

현실세계에는 ‘동물’ 이라는 오브젝트는 존재하지 않는다. (따라서 프로그램에서도 동물 오브젝트가 생성되면 안된다.)

강아지, 고양이 등 자식 class에서 속성을 상속받기 위한 용도로만 사용되어야 함.

## 자바 인터페이스와 다형성

어떤 경우에 interface를 사용하는가?

* 특정 ‘기능’이 하나의 Class에 귀속되지 않는 경우 인터페이스를 활용할 수 있다.

추상 클래스와 인터페이스의 차이

1. 추상클래스는 상속해서 사용한다. 모든 클래스는 1개의 클래스만 상속이 가능하다. 인터페이스는 여러 개 구현 가능하다.
2. 추상클래스는 멤버 변수를 가질 수 있다. 인터페이스는 멤버 변수를 가질 수 없다. 상수만 가질 수 있다.
3. 추상클래스는 보통 상속가능한 계층이 있는 관계의 경우 사용한다. 인터페이스는 기능이다. 이 기능이 있어야 한다면 누구나 구현할 수 있다.

## Final 키워드 설명

Final이 변수/메서드/클래스 앞에 붙으면, 해당하는 entity는 변경할 수 없는 entity가 된다.

Final 변수 – 변수에 초기화 된 값 변경 불가

Final 메서드 – 상속받은 class에서 해당 method 오버라이드 불가

Final 클래스 – 이 클래스 상속 불가

## Exception

개념 : 예외는 어떤 이유로 컴퓨터가 더 이상 프로그램을 진행 할 수 없는 상태가 되는 것을 의미한다.

문법적인 측면에서 봤을 때 예외란 그냥 Object이다. JVM이 프로그램을 실행하다가 예외가 발생하는 경우 그 예외에 맞는 오브젝트를 만들어 throw라는 문법을 이용해 예외를 발생시킨다.

*throw new* ArrayIndexOutOfBoundsException();

예외가 발생하였을 때 발생한 예외가 처리되지 않는다면, 예외가 발생한 메서드를 호출한 메서드로 전파된다. 그러다 메인 메서드에서도 예외가 처리되지 않으면 프로그램 실행이 종료가 되는 것이다.

## Try-catch-finally

Finally는 보통 IO(Input/Output) 오퍼레이션 또는 Synchronization(Lock)에서 많이 사용한다.

IO 오퍼레이션에는 파일입출력, 네트워크 커넥션, 데이터베이스 커넥션 등이 있다. 파일/네트워크/데이터베이스 등 어떤 리소스를 사용하기 위해 open(또는 connect)이라는 메서드를 사용한다. 프로그램은 open한 리소스를 사용이 끝난 후 close 해야 한다. 그렇지 않으면 리소스를 제공한 입장(운영체제/서버\_에서는 이 리소스는 누군가 사용중이고 close를 하지 않았으므로 이 리소스를 필요로 하는 다른 프로그램에게 제공할 수 없다.

## Generic

#### Generic이 필요한 이유

Generic은 타입에 대한 추상화이다. 오브젝트를 생성할 때 내가 원하는 자료형으로 오브젝트를 생성할 수 있다.

*public static void* main(String[] args) {  
 ArrayList<String> list = *new* ArrayList();  
 list.add("Hello World");  
 String myString = list.get(0);  
 System.out.println(myString);  
}

Generics는 위처럼 <> 꺽쇠 괄호 안에 이 오브젝트가 담고 있을 또는 사용할 자료형을 지정해 주는 것. 하나의 리스트 클래스가 여러 자료형을 받을 수 있도록 하고, 그 자료형을 컴파일 시간에 명시적으로 지정할 수 있게 한다.

그리하여 컴파일러가 타입 체크를 해주고, 타입 캐스팅으로 인한 런타임 에러 방지 할 수 있다.

#### 자바 Generics 클래스

‘클래스이름<T>’ 의 의미는 ‘이 클래스는 타입이 아직 결정되지 않은 변수를 사용한다. 그리고 그 변수의 타입을 T라고 가정할 것이다’ 라는 의미이다.

*import* java.util.ArrayList;  
*import* java.util.Arrays;  
*import* java.util.List;  
  
*public class* FixedList<T> {  
 *private* T[] list;  
 *private int* size;  
  
 *public* FixedList(*int* capacity) {  
 size = 0;  
 list = (T[]) *new* Object[capacity];  
 }  
  
 *public void* add(T data) {  
 *if* (data == *null*) {  
 *throw new* NullPointerException();  
 }  
 list[size++] = data;  
 }  
  
 *public* List<T> asList() {  
 *if* (list == *null*) *return new* ArrayList<>();  
 *return* Arrays.*asList*(list);  
 }  
}

위 예시코드에서 볼 수 있듯이, Generics으로 전달받은 자료형 T를 클래스 내에서 대표적으로

1. Private T[] list; 처럼 변수 선언 시 사용
2. Public void add(T data){} 처럼 메서드 파라메터 자료형 자리에 사용.
3. Public List<T> asList(){} 처럼 자료형 리턴 적는 곳에 사용

## JavaBean 이란?

JavaBean은 JavaBean API Specification에 따른 standard이다.

JavaBean의 목적은 여러가지 다른 오브젝트들을 하나의 Object에 담기 위함이다. JavaBean의 규칙을 소프트웨어 프로토콜이라고 생각하면 된다.

JavaBean이란 아래 3가지 규칙을 지키는 클래스이다.

1. 모든 필드는 private이며, getter/setter 메서드를 통해서만 접근이 가능하다.  
   -> 캡슐화를 위하여
2. Argument가 없는(no-argument) 생성자가 존재한다.  
   -> 편하게 사용하기 위한 일종의 약속,
3. Java.io.Serializable 인터페이스를 구현한다.  
   -> 오브젝트를 네트워크 전송이나 파일에 저장하기 위함, JavaBean은 저장하거나 전송하는 일이 많기 때문에 스탠다드에 포함 된 것으로 추정

좀 더 구체적인 설명은 아래 블로그에

https://imasoftwareengineer.tistory.com/101?category=769427

## 자바 Lambda

소프트웨어 세계에서는 람다란 함수이다. 자바에서는 메서드이다.

그러면 모든 함수나 메서드는 람다인가? 아니다. **매개변수로 전해질 수 있는 함수/메서드**만을 람다라고 한다.

전통적인 자바 프로그램에서는 아래처럼 인터페이스-인터페이스 구현클래스로 나누어 많이 개발한다.

*public interface* Operation {  
 Integer add(Integer x, Integer y);  
}

*public class* OperationImpl *implements* Operation{  
 @Override  
 *public* Integer add(Integer x, Integer y) {  
 *return* x + y;  
 }  
}

*public class* App {  
  
 *public static void* main(String[] args) {  
 Operation op = *new* OperationImpl();  
 *int* result = op.add(1, 3);  
 System.out.println(result);  
 }  
}

위 코드에서 add메서드를 사용하기 위하여 개발자는 add 메서드가 존재하는 인터페이스 생성(Operation), 인터페이스 구현(OperationImpl), 그리고 메서드를 사용하기 위해 인터페이스 오브젝트 생성(new OperationImpl) 과정을 거쳤다. 인터페이스 구현과 오브젝트 생성과정을 하나로 합치고, 별도의 OperationImpl.java 같은 파일 없이도 인터페이스를 구현할 수 있는 방법은 아래 처럼 하면 된다.

*public class* App {  
  
 *public static void* main(String[] args) {  
 Operation op = *new* Operation() {  
 @Override  
 *public* Integer add(Integer x, Integer y) {  
 *return* x + y;  
 }  
 };  
 *int* result = op.add(1, 3);  
 System.out.println(result);  
 }  
   
}

인터페이스 구현체를 안 만들기 위해서는 위 코드처럼 **new 인터페이스 이름{ 오버라이드 메서드 구현 부분 }** 처럼 작성하여 인터페이스를 바로 구현하여 사용할 수 있다.

*public class* App {  
  
 *public static void* main(String[] args) {  
 Operation op = (x, y) -> { *return* x + y; }; // @Override public int add(int x, int y) { return x + y; } 와 같은 뜻

*int* result = op.add(1, 3);  
 System.out.println(result);  
 }  
  
}

람다식으로 간결하게 쓰면 위 처럼 첫 괄호에는 인터페이스 구현부분의 파라미터 이름을 개수에 맞게 넣어주고, ‘->’ 다음에는 메서드의 바디 부분을 { } 꺽쇠괄호 안에 넣어주면 된다.

Operation 인터페이스에서 add 메서드의 인자가 2개였기 때문에 x, y 두 개를 매개변수로 넣는 것이다.

## Lambda의 제약

1. 람다는 인터페이스에 메서드가 한 개만 있을 경우만 사용 가능하다.

왜냐하면 인터페이스에 메서드가 2개 이상 있는 경우 람다식으로 구현부분을 작성할 때 파라메터의 자료형을 명시하지 않기 때문에 특정 메서드로 맵핑을 할 수 없기 때문이다.

1. 람다는 1회 사용을 위해 인터페이스를 구현하는 상황에서 사용하면 효율이 좋다. (딱 한군데서만 쓰고 다른 아무데서도 안쓰는 경우)  
   대표적으로 Comparator (비교 메서드)의 경우

*public class* App {  
  
 *public static void* main(String[] args) {  
 Operation op = (x, y) -> x + y;  
 *printResult*(op);  
 }  
  
 *private static void* printResult(Operation op) {  
 *int* result = op.add(1, 3);  
 System.out.println(result);  
 }  
}

printResult라는 메서드를 만들고 메서드의 매개변수로 Operation을 넘겨준다. 메인에서는 printResults에 넘겨 줄 op 오브젝트를 람다 표현식으로 초기화한다. **매개변수로 전해질 수 있는 함수 / 메서드를 람다라고 부른다.**

*public class* App {  
  
 *public static void* main(String[] args) {  
 *printResult*((x, y) -> x + y);  
 }  
  
 *private static void* printResult(Operation op) {  
 *int* result = op.add(1, 3);  
 System.out.println(result);  
 }  
}

람다 표현을 사용하면 그 자체로 인터페이스의 구현 오브젝트가 생성된다. 람다 표현 자체가 하나의 인터페이스 오브젝트이므로 이를 매개변수로 넘기는 것에도 문제가 없다.

출처: <https://imasoftwareengineer.tistory.com/99?category=769428> [삐멜 소프트웨어 엔지니어]

# 디자인 패턴

위 처럼 Main에서 UserValidator를 정의한후 위처럼 배열에 UserValidator가 있는 만큼 반복문을 돌려주면 10개의 메서드를 부르지 않아도 Validation을 할 수 있다. 이렇게하면 Validator가 몇개이든 상관없고, 나중에 또 다른 UserValidator를 추가하고싶을 때 배열에만 추가하면된다. 이렇게 하면 validate이라는 메서드를 부르지만 각 메서드의 구현부는 NameUserValidator, AgeUserValidator, ...등등 다르다.

출처: <https://imasoftwareengineer.tistory.com/79?category=769427> [삐멜 소프트웨어 엔지니어]

*public interface UserValidator* {  
 *void* validate(User user);  
}

NameUserValidator.java

*public class* NameUserValidator *implements UserValidator* {  
 @Override  
 *public void* validate(User user) {  
 *if*(user.name == *null* || user.name.isEmpty()) {  
 System.out.println("에러: 이름을 입력해주세요.");  
 }  
 }  
}

AgeUserValidator.java

*public class* AgeUserValidator *implements UserValidator* {  
 @Override  
 *public void* validate(User user) {  
 *if*(user.age <= 0 || user.age > 99) {  
 System.out.println("에러: 올바른 나이를 입력해주세요.");  
 }  
 }  
}

이렇게 만든 Validator들을 어떻게 사용할까?

*import* java.util.Scanner;  
  
*public class* Main {  
  
 *static UserValidator*[] userValidator = {*new* NameUserValidator(), *new* AgeUserValidator()};  
  
 *public static void* main(String[] args) {  
 User user = *new* User();  
 System.out.print("## 이름 :");  
 user.name = *new* Scanner(System.in).next();  
 System.out.print("## 나이 :");  
 user.age = *new* Scanner(System.in).nextInt();  
  
 *for* (*int* i = 0; i < userValidator.length; i++) {  
 userValidator[i].validate(user);  
 }  
 }  
}

//실행 결과

## 이름 :d

## 나이 :-1

에러: 올바른 나이를 입력해주세요.