**💡 원시타입, 참조타입(Primitive Type, Reference Type)**

* 자바에서는 데이터 타입에 크게 두 가지 원시 타입(Primitive Type)과 참조타입(Reference Type)이 있다.
* 원시 타입은 쉽게 말해, 정수, 실수, 문자, 논리 리터럴등의 실제 데이터 값을 저장하는 타입이고,
* 참조 타입은 객체(Object)의 번지를 참조(주소를 저장)하는 타입으로 메모리 번지 값을 통해 객체를 참조하는 타입이다.

**원시 타입(Primitive Type)**

**텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명**

* 실수형 데이터타입에서는 double형 데이터타입이 기본 데이터타입이다.
* 기본형 데이터타입이 double형 이기 때문에 float형 데이터타입의 실수형 데이터를 사용하고자 하는 경우에도long형과 마찬가지로 실수 데이터 맨 뒤 쪽에 접미사 'f'나 'F'를 붙여줘야 한다.
* float형 데이터타입보다 double형 데이터타입이 두배정도 더 정밀한 데이터를 표현할 수 있다.

**참조 타입(Reference Type)**

* 참조 타입은 원시 타입을 제외한 타입들(문자열, 배열, 열거, 클래스, 인터페이스)을 말한다.
* Java에서 실제 객체는 힙 영역에 저장되며 참조 타입 변수는 스택 영역에 실제 객체들의 주소를 저장하여, 객체를 사용할때 마다 참조 변수에 저장된 객체의 주소를 불러와 사용하는 방식이다.

**정적 메모리 스택(Stack) 영역**

* 스택 영역에는 기본타입 변수가 할당되고 변수의 실제 값들이 저장된다.
* 참조 타입의 변수들은 이 스택 영역에서 힙 영역에 생성된 객체들의 주소 값을 저장하고 있는다.
* 객체 안의 메소드의 작업이 종료되면 할당되었던 메모리 공간은 반환되어 비워진다.

**동적 메모리 힙(Heap) 영역**

* 힙 영역에는 객체와 배열이 생성된다.
* 그리고 참조타입(배열, 클래스, 인터페이스 등)들이 이 객체들의 주소를 스택 영역에 저장한다.
* 기본타입 변수들과는 다르게 크기가 정해져 있지 않다.
* 프로그램 실행시 메모리에 동적으로 할당된다.
* 참조하는 변수가 없으면 자바의 가비지 컬렉터가 제거한다.
* 가비지 컬렉터(Garbage collector) : 메모리의 힙 영역에 할당된 더 이상 사용되지 않는 객체를 제거 하는 역할
* 이렇게 객체를 제거하며 메모리가 관리된다.

**💡 Boxing, Unboxing**

* Boxing(박싱)은 원시 타입을 참조 타입으로 변환 시키는 것을 말하고, Unboxing(언박싱)은 참조 타입을 원시 타입으로 변환 시키는 것을 말한다.
* 자바 1.5 이전에는 일일히 변환 과정을 거쳐주어야 했지만, 자바 1.5부터 추가된 Auto Boxing / Unboxing 기능으로 아래의 예시와 같이 명시적으로 원시타입을 참조타입으로 감싸주지 않아도 자동으로 Boxing / Unboxing 해준다.
* 이러한 Auto Boxing / Unboxing 기능은 메모리 누수의 원인이 될 수도 있다.

**💡 원시 타입과 참조 타입 차이**

**1. Null 포함 가능 여부**

원시타입은 null을 담을 수 없지만, 참조 타입은 가능하다.

**2. 제너릭 타입에서 사용 가능 여부**

원시타입은 제너릭 타입에서 사용할 수 없지만, 참조 타입은 가능하다.

**💡 원시 타입의 장점**

**1. 접근속도**

원시타입은 '스택' 메모리에 값이 존재한다.

반면에 참조타입은 하나의 인스턴스이기 때문에 '스택' 메모리에는 참조값만 있고, 실제 값은 힙 메모리에 존재한다.

그리고 값을 필요로 할 때마다 언박싱 과정을 거쳐야 하니 원시타입과 비교해서 접근 속도가 느려지게 된다.

예외적으로 엄청 큰 숫자를 복사해야 한다면, 참조값만 넘길 수 있는 참조타입이 좋을 수 도 있다.

**2. 메모리 양**

원시 타입보다 참조 타입이 사용하는 메모리양이 압도적으로 높다.

따라서 메모리 사용적으로도 원시 타입이 참조 타입보다 효율적으로 사용할 수 있다.

**💡** **인터페이스 개념과 역할**

극단적으로 동일한 목적 하에 동일한 기능을 수행하게끔 강제하는 것이 바로 인터페이스의 역할이자 개념이다.

자바의 다형성을 극대화하여 개발코드 수정을 줄이고 프로그램 유지보수성을 높이기 위해 인터페이스를 사용한다.

**💡** **인터페이스 문법과 다형성 이해**

인터페이스는 interface 키워드를 통해 선언할 수 있으며 implements 키워드를 통해 일반 클래스에서 인터페이스를 구현할 수 있다.

또한, JAVA8 이전까지는 상수, 추상메소드만 선언이 가능하지만,(상수, 추상메소드만 가능케했다는 것을 통해 그만큼 강제성이 강했다는 것을 유추할 수 있다.)

JAVA8부터 디폴트메소드, 정적메소드가 추가되었다.

**상수 : 인터페이스에서 값을 정해줄테니 함부로 바꾸지 말고 제공해주는 값만 참조해라 (절대적)**

//상수

타입 상수명 = 값;

**추상메소드 : 가이드만 줄테니 추상메소드를 오버라이팅해서 재구현해라. (강제적)**

//추상 메소드

타입 메소드명(매개변수, ... );

**디폴트메소드 : 인터페이스에서 기본적으로 제공해주지만, 맘에 안들면 각자 구현해서 써라. (선택적)**

//디폴트 메소드

default 타입 메소드명(매개변수, ... ){

//구현부

}

**정적메소드 : 인터페이스에서 제공해주는 것으로 무조건 사용 (절대적)**

//정적 메소드

static 타입 메소드명(매개변수) {

//구현부

}

**💡** **많이 쓰는 14가지 핵심 GoF 디자인 패턴의 종류**

디자인 패턴은 프로그램을 개발하는 과정에서 빈번하게 발생하는 디자인 문제를 정리해서 상황에 따라 간편하게 적용할 수 있게 정리한 것입니다. 잘 활용할 수만 있다면 적지 않은 시간과 노력, 시행착오를 줄일 수 있습니다.

**생성 패턴(Creational Pattern)**

객체 인스턴스를 생성하는 패턴으로, 클라이언트와 그 클라이언트가 생성해야 하는 객체 인스턴스 사이의 연결을 끊어 주는 패턴입니다.

**1. 싱글턴 패턴(Singleton Pattern)**

특정 클래스에 객체 인스턴스가 하나만 만들어지도록 해주는 패턴입니다. 싱글턴 패턴을 사용하면 전역 변수를 사용할 때와 마찬가지로 객체 인스턴스를 어디서든지 액세스 할 수 있게 만들 수 있습니다. 클래스 인스턴스를 하나만 만들고 그 인스턴스로의 전역 접근을 제공합니다.

2. 추상 팩토리 패턴(Abstract Factory Pattern)

구상 클래스에 의존하지 않고도 서로 연관되거나 의존적인 객체로 이루어진 제품군을 생산하는 인터페이스를 제공합니다. 구상 클래스는 서브 클래스에서 만듭니다.

3. 팩토리 메소드 패턴(Factory method pattern)

객체를 생성할 때 필요한 인터페이스를 만듭니다. 어떤 클래스의 인스턴스를 만들지는 서브클래스에서 결정합니다. 팩토리 메소드 패턴을 사용하면 클래스 인스턴스 만드는 일을 서브클래스에게 맡기게 됩니다.

**행동 패턴(Behavioral Pattern)**

클래스와 객체들이 상호작용하는 방법과 역할을 분담하는 방법을 다루는 패턴입니다.

1. 템플릿 메소드 패턴(Template Method Pattern)

알고리즘의 골격을 정의합니다. 템플릿 메소드를 사용하면 알고리즘 일부 단계를 서브클래스에서 구현할 수 있으며, 알고리즘의 구조는 그대로 유지하면서 알고리즘의 특정 단계를 서브클래스에서 재정의할 수도 있습니다.

2. 상태 패턴(State Pattern)

상태 패턴을 사용하면 객체의 내부 상태가 바뀜에 따라서 객체의 행동을 바꿀 수 있습니다. 마치 객체의 클래스가 바뀌는 것과 같은 결과를 얻을 수 있습니다.

3. 반복자 패턴(iterator pattern)

컬렉션의 구현 방법을 노출하지 않으면서 집합체 내의 모든 항목에 접근하는 방법을 제공합니다.

**4. 전략 패턴(Strategy Pattern)**

알고리즘군을 정의하고 캡슐화해서 각각의 알고리즘군을 수정해서 쓸 수 있게 해 줍니다. 전략 패턴을 사용하면 클라이언트로부터 알고리즘을 분리해서 독립적으로 변경할 수 있습니다.

런타임 중에 알고리즘을 선택할 수 있게 하는 디자인 패턴

**5. 옵저버 패턴(observer pattern)**

한 객체의 상태가 바뀌면 그 객체에 의존하는 다른 객체에게 연락이 가고 자동으로 내용이 갱신되는 방식으로 일대다 의존성을 정의합니다.

많은 객체 간의 일대다 관계인 경우의 기초

**구조 패턴(Structural Pattern)**

클래스와 객체를 더 큰 구조로 만들 수 있게 구상을 사용하는 패턴입니다.

1. 데코레이터 패턴(Decorator Pattern)

데코레이터 패턴으로 객체에 추가 요소를 동적으로 더할 수 있습니다. 데코레이터를 사용하면 서브클래스를 만들 때보다 훨씬 유연하게 기능을 확장할 수 있습니다.

2. 프록시 패턴(Proxy Pattern)

특정 객체로의 접근을 제어하는 대리인(특정 객체를 대변하는 객체)을 제공합니다.

3. 컴포지트 패턴(Composite Pattern)

컴포지트 패턴으로 객체를 트리구조로 구성해서 부분-전체 계층구조를 구현합니다. 컴포지트 패턴을 사용하면 클라이언트에서 개별 객체와 복합 객체를 똑같은 방법으로 다룰 수 있습니다.