신입 개발자 기술면접 질문 : 자바

신입 개발자 면접에서 자주 나오는 질문과 그에 대한 모범 답안을 준비해 보겠습니다. 여기 몇 가지 예시가 있습니다.

1. 자기소개 부탁합니다.
모범 답안: 저는 컴퓨터 과학을 전공했으며, 대학교에서 다양한 프로젝트에 참여하며 프로그래밍 기술을 키웠습니다. 특히, [특정 기술]에 관심이 많아 이를 중심으로 여러 학습 프로젝트를 진행했습니다. 또한, [인턴십/대회 참가 등의 경험]을 통해 실제 업무 환경에서의 문제 해결 능력을 키우고 협업하는 법을 배웠습니다.

2. 여러 프로그래밍 언어 중 어떤 언어가 가장 자신 있습니까?
모범 답안: 저는 [특정 프로그래밍 언어, 예: Python]을 가장 잘 다룹니다. 이 언어의 [강점, 예: 다양성과 유연성] 때문에 선호합니다. 대학 시절부터 이 언어로 여러 프로젝트를 진행했고, [특정 기술, 예: 머신러닝] 분야에 적용해 보면서 해당 언어의 심화 지식을 쌓았습니다.

3. 팀 프로젝트에서 어떤 역할을 주로 맡았나요?
모범 답안: 팀 프로젝트에서 저는 주로 [역할, 예: 프로젝트 매니저 또는 개발자] 역할을 맡았습니다. 이 역할을 통해 저는 프로젝트의 목표 설정, 일정 관리, 그리고 팀원 간의 소통을 촉진하는 중요한 역할을 했습니다. 이 경험을 통해 저는 명확한 커뮤니케이션과 팀워크의 중요성을 배웠습니다.

4. 실패 경험에 대해 말해주세요. 그리고 그 경험으로부터 무엇을 배웠나요?
모범 답안: [특정 프로젝트]에서 제가 맡았던 부분의 기한을 지키지 못한 적이 있습니다. 이는 제 시간 관리 능력 부족과 기술적 문제에 대한 충분한 예측 부재에서 비롯되었습니다. 이 경험을 통해, 저는 계획을 세울 때 여유를 가지고 접근하는 것의 중요성과 문제 발생 시 빠르게 대응할 수 있는 대비책의 필요성을 배웠습니다.

5. 새로운 기술을 배울 때 어떤 방식으로 접근하나요?
모범 답안: 새로운 기술을 배울 때, 저는 공식 문서를 먼저 살펴보고 기본 개념을 이해하는 것부터 시작합니다. 그 다음, 실제 작은 프로젝트를 만들어보면서 학습한 내용을 실습에 적용해보고, 문제가 생길 때마다 관련 커뮤니티나 포럼을 통해 해결 방법을 탐색합니다. 이 과정을 통해 이론과 실습을 병행하며 효과적으로 새로운 기술을 습득합니다.

이러한 답변들은 면접관에게 여러분의 기술적 능력, 팀워크, 문제 해결 능력, 그리고 학습 의지를 보여줄 수 있습니다. 면접에서는 이러한 질문들에 대한 답변을 준비하는 것이 중요하며, 개인의 경험과 성장 과정을 강조하는 것이 좋습니다.

신입 자바 개발자가 면접에서 만날 수 있는 기술 관련 질문은 자바의 기본 개념, 객체 지향 프로그래밍, 자바의 주요 특징 및 기능, 그리고 실제 개발 경험과 관련된 내용을 포함할 수 있습니다. 다음은 신입 자바 개발자 면접에서 흔히 물어볼 수 있는 몇 가지 질문입니다:

자바 기본 개념
자바의 주요 특징은 무엇인가요? 예: 객체 지향, 플랫폼 독립성, 가비지 컬렉션 등에 대한 설명
객체 지향 프로그래밍(OOP)의 4대 기본 원칙에 대해 설명해주세요. 캡슐화, 상속, 다형성, 추상화
인터페이스와 추상 클래스의 차이점은 무엇인가요?
오버로딩과 오버라이딩의 차이점은 무엇인가요?

자바 고급 개념
자바에서 제네릭을 사용하는 이유는 무엇인가요?
컬렉션 프레임워크의 주요 인터페이스와 클래스에 대해 설명해주세요.
멀티스레딩과 동시성에 대해 설명해주세요. 자바에서 스레드를 어떻게 생성하나요?

자바 성능 및 최적화
가비지 컬렉션(GC)의 원리와 종류에 대해 설명해주세요.
JVM의 구조와 작동 원리에 대해 설명해주세요.

실무 경험과 문제 해결
자바 프로젝트에서 경험한 가장 어려웠던 문제는 무엇이었고, 어떻게 해결했나요?
디자인 패턴에 대해 알고 있는 것이 있나요? 사용해본 패턴에 대해 설명해주세요.

자바 신기술
자바의 최신 버전(예: 자바 11 이상)에서 추가된 주요 기능에 대해 알고 있나요?

이러한 질문들을 통해 면접관은 지원자의 자바에 대한 기술적 이해도와 문제 해결 능력, 그리고 실무 경험과 같은 다양한 측면을 평가할 수 있습니다. 따라서 신입 자바 개발자로서 이러한 주제들에 대한 충분한 이해와 준비가 필요합니다.

이제부터 이러한 것과 관련된 질문에 대한 답변을 만들어 보겠습니다. 아래 예는 표준적인 답변이며 각자 개인의 능력과 성향에 따라 변경해서 자신의 문서를 만들어 나가길 바랍니다.

기술면접 관련 질문 예

□ Java의 특징을 간략하게 설명해 주세요.

Java는 객체지향 프로그래밍 언어입니다. 기본 자료형을 제외한 모든 요소들이 객체로 표현되고, 객체 지향 개념의 특징인 캡슐화, 상속, 다형성이 잘 적용된 언어입니다.

장점

JVM(자바가상머신) 위에서 동작하기 때문에 운영체제에 독립적이다.

GabageCollector를 통한 자동적인 메모리 관리가 가능하다.

단점

JVM 위에서 동작하기 때문에 실행 속도가 상대적으로 느리다.

다중 상속이나 타입에 엄격하며, 제약이 많다.

Java는 널리 사용되는 프로그래밍 언어로, 다음과 같은 주요 특징을 가지고 있습니다:

1. 플랫폼 독립성

Java는 "한 번 작성하면 어디서나 실행될 수 있다(Write Once, Run Anywhere - WORA)"는 철학을 가지고 있습니다. Java 프로그램은 Java Virtual Machine 위에서 실행되기 때문에, JVM이 설치된 모든 운영 체제에서 실행할 수 있습니다. 이로 인해 개발자는 다양한 플랫폼에 대해 걱정하지 않고 애플리케이션을 개발할 수 있습니다.

2. 객체 지향 프로그래밍(OOP)

Java는 순수 객체 지향 프로그래밍 언어입니다. 모든 것을 객체로 처리하며, 클래스를 사용하여 객체의 행동과 상태를 정의합니다. 이 접근 방식은 코드의 재사용성, 확장성 및 유지 보수성을 증가시킵니다.

3. 강력한 메모리 관리

자동 메모리 관리와 가비지 컬렉션 기능을 제공합니다. 개발자는 객체를 생성할 수 있지만, 메모리 해제는 Java의 가비지 컬렉터가 자동으로 처리합니다. 이는 메모리 누수와 같은 문제를 줄이는 데 도움이 됩니다.

4. 멀티스레딩 지원

Java는 멀티스레딩을 내장 지원합니다. 이를 통해 개발자는 동시에 여러 작업을 수행하는 복잡한 애플리케이션을 쉽게 개발할 수 있습니다. 멀티스레드 프로그래밍은 애플리케이션의 성능을 크게 향상시킬 수 있습니다.

5. 보안

Java는 샌드박스 보안 모델을 사용하여 시스템 보안을 강화합니다. Java 애플리케이션은 기본적으로 JVM 내부의 격리된 환경에서 실행되므로, 시스템의 다른 부분에 무단으로 액세스하는 것을 방지합니다.

6. 풍부한 표준 라이브러리

Java는 방대한 표준 라이브러리를 제공하여, 네트워킹, 파일 입출력, 데이터 구조, 그래픽 사용자 인터페이스 구성 등 다양한 작업을 쉽게 할 수 있게 도와줍니다. 이 라이브러리들은 Java 개발을 더욱 효율적이고 빠르게 만듭니다.

7. 포괄적인 도구 지원

다양한 개발 도구 및 IDE(통합 개발 환경)가 Java 개발을 지원합니다. Eclipse, IntelliJ IDEA와 같은 강력한 IDE는 코드 작성, 디버깅, 테스팅을 쉽게 할 수 있도록 돕습니다.

Java의 이러한 특징들은 그것을 웹 애플리케이션, 엔터프라이즈 소프트웨어, 모바일 애플리케이션(Android), 임베디드 시스템 등 다양한 분야에서 인기 있는 선택으로 만들었습니다.

□ JVM의 역할에 대해 설명해 보세요.

JVM은 스택 기반으로 동작하며, Java Byte Code를 OS에 맞게 해석 해주는 역할을 하고 가비지 컬렉션을 통해 자동적인 메모리 관리를 해줍니다. Java Virtual Machine(JVM)은 Java 프로그램을 실행하는 핵심 구성 요소로서, Java의 플랫폼 독립성과 보안성, 그리고 성능 최적화를 가능하게 하는 중요한 역할을 합니다. JVM의 주요 역할은 다음과 같습니다:

1. 플랫폼 독립성 제공

JVM은 Java 바이트코드를 실행하는 가상 기계입니다. Java 소스 코드는 컴파일러에 의해 바이트코드로 변환되며, 이 바이트코드는 모든 JVM에서 실행될 수 있습니다. 이러한 구조 덕분에, Java 애플리케이션은 다양한 운영 체제에서 수정 없이 실행될 수 있습니다("한 번 작성하면, 어디서나 실행된다").

2. 보안

JVM은 여러 보안 기능을 제공하여 애플리케이션과 시스템을 보호합니다. 예를 들어, 클래스 로더는 클래스를 동적으로 로드할 때 보안 검사를 수행하며, 바이트코드 검증기는 코드 실행 전에 바이트코드가 JVM 사양을 준수하는지 검증합니다. 이는 안전하지 않은 코드 실행을 방지합니다.

3. 성능 최적화

현대의 JVM은 Just-In-Time(JIT) 컴파일러를 포함하여 실행 시간 동안 코드의 성능을 향상시킵니다. JIT 컴파일러는 자주 실행되는 바이트코드를 효율적인 네이티브 코드로 변환하여 애플리케이션의 실행 속도를 빠르게 합니다.

4. 메모리 관리

JVM은 가비지 컬렉션(GC)을 통해 자동으로 메모리 관리를 수행합니다. 개발자는 객체를 생성할 수 있지만, JVM이 사용되지 않는 객체를 감지하고 자동으로 메모리를 해제합니다. 이는 메모리 누수를 방지하고 애플리케이션의 안정성을 높입니다.

5. 멀티스레드 지원

JVM은 자체적으로 스레드를 관리합니다. 이는 Java 애플리케이션이 멀티스레딩을 쉽게 구현할 수 있게 해주며, 운영 체제의 스레드 관리 방식과는 독립적으로 작동합니다. JVM의 스레드 관리는 프로그램의 동시성과 성능을 개선합니다.

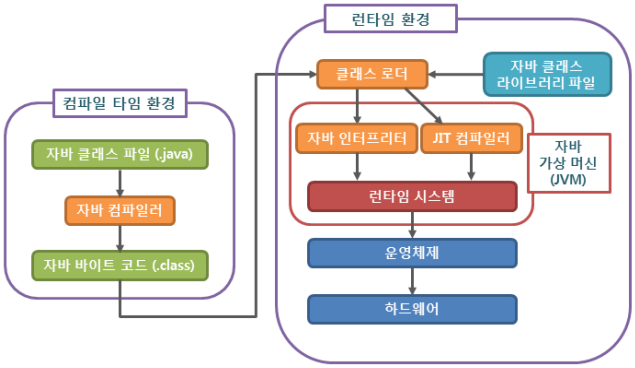
6. 표준화된 인터페이스 제공

JVM은 Java SE 표준을 구현합니다. 이는 개발자가 표준 Java API를 사용하여 플랫폼에 관계없이 일관된 방식으로 프로그래밍할 수 있게 해줍니다.

JVM의 이러한 역할은 Java가 강력한 성능, 높은 이식성, 강력한 보안을 갖춘 언어로 자리 잡는 데 기여했습니다. JVM은 Java 생태계의 핵심이며, Java의 성공적인 크로스 플랫폼 실행, 보안, 성능 최적화를 가능하게 하는 주요 기술입니다.

□ Java의 컴파일 과정에 대해 설명해주세요.

- 개발자가 .java 파일을 생성한다.
- build를 한다.
- java compiler의 javac의 명령어를 통해 바이트코드(.class)를 생성한다.
- Class Loader를 통해 JVM 메모리 내로 로드 한다.
- 실행엔진을 통해 컴퓨터가 읽을 수 있는 기계어로 해석된다.(각 운영체제에 맞는 기계어)



Java의 컴파일 과정은 소스 코드가 실행 가능한 프로그램으로 변환되는 일련의 단계로 구성됩니다. 이 과정은 크게 네 가지 주요 단계로 나눌 수 있습니다:

1) 소스 코드 작성: 프로그래머는 .java 확장자를 가진 파일에 Java 프로그래밍 언어로 소스 코드를 작성합니다. 이 코드는 클래스, 메서드, 변수 등을 포함할 수 있습니다.

2) 컴파일: 작성된 소스 코드 파일을 Java 컴파일러(javac)를 사용하여 컴파일합니다. 컴파일러는 소스 코드를 분석하고, 문법 오류를 검사한 다음, 기계가 이해할 수 있는 중간 형태인 바이트코드로 변환합니다. 이 바이트코드는 .class 확장자를 가진 파일에 저장됩니다. 바이트코드는 특정 하드웨어나 운영 체제에 종속되지 않는 플랫폼 독립적인 코드입니다.

3) 로드: 컴파일된 바이트코드 파일은 Java 가상 머신(JVM)에 의해 로드됩니다. JVM은 .class 파일을 로드하고, 링크하며, 초기화하는 과정을 거칩니다. 이 단계에서 JVM은 동적 로딩을 사용하여 필요할 때 클래스를 로드하고, 런타임에 필요한 다른 클래스들과의 연결을 처리합니다.

4) 실행: JVM은 로드된 바이트코드를 실행합니다. JVM 내부의 인터프리터나 Just-In-Time(JIT) 컴파일러가 바이트코드를 기계어로 변환하여 실제 하드웨어에서 실행될 수 있도록 합니다. 이 과정에서 JVM은 메모리 관리, 가비지 컬렉션, 스레드 관리 등 다양한 작업을 자동으로 처리합니다.

이 과정을 통해 Java 프로그램은 다양한 플랫폼에서 동일한 방식으로 실행될 수 있는 플랫폼 독립성을 가지게 됩니다. Java의 "한 번 작성하면 어디서나 실행된다(Write Once, Run Anywhere)"라는 철학은 바로 이러한 컴파일 과정과 실행 환경 덕분에 가능한 것입니다.

□ Java에서 제공하는 원시 타입(기본형)에는 어떠한 것이 있고, 각각 크기는 몇 바이트를 차지하나요?
정수형 byte, short, int, long
실수형 float, double
문자형 char
논리형 boolean이 있고,

크기는 각각 정수형 1, 2, 4, 8,
실수형 4, 8,
문자형 2,

논리형 1 바이트를 차지합니다.

Java에서 제공하는 원시 타입(기본형)은 크게 논리형, 문자형, 정수형, 그리고 실수형으로 나뉩니다. 각 타입의 크기는 다음과 같습니다:

- 논리형
boolean: 1비트 (논리형 값으로 true 또는 false를 가집니다. 실제 메모리 사용량은 JVM 구현에 따라 다를 수 있으나, 개념적으로는 1비트를 사용합니다.)
- 문자형
char: 2바이트 (16비트, 유니코드 문자를 표현합니다.)
- 정수형
byte: 1바이트 (8비트, -128에서 127까지의 정수 값을 표현합니다.)
short: 2바이트 (16비트, -32,768에서 32,767까지의 정수 값을 표현합니다.)
int: 4바이트 (32비트, 약 -21억에서 21억까지의 정수 값을 표현합니다.)
long: 8바이트 (64비트, 약 -922경에서 922경까지의 정수 값을 표현합니다.)
- 실수형
float: 4바이트 (32비트, 부동소수점 수를 표현합니다. 유효 자릿수는 대략 7자리입니다.)
double: 8바이트 (64비트, 부동소수점 수를 표현합니다. 유효 자릿수는 대략 15자리입니다.)

Java에서는 이러한 원시 타입을 사용하여 효율적인 메모리 관리와 빠른 처리 속도를 달성할 수 있습니다. 원시 타입은 객체가 아니므로 스택 메모리에 저장되며, 직접적인 값(value)을 다룹니다.

□ 오버라이딩(Overriding)과 오버로딩(Overloading)에 대해 간략하게 설명해 보세요.
오버라이딩(Overriding)은 상위 클래스에 있는 메소드를 하위 클래스에서 재정의 하는 것을 말하고,
오버로딩(Overloading)은 매개변수의 개수나 타입을 다르게 하여 같은 이름의 메소드를 여러 개 정의하는 것을 말합니다.

오버라이딩은 하위 클래스가 상위 클래스에서 상속받은 메서드를 재정의하는 것을 의미합니다. 이를 통해 상속받은 메서드의 동작을 하위 클래스에 맞게 변경할 수 있습니다. 오버라이딩된 메서드는 실행 시점에 결정되는 런타임 다형성을 제공합니다. 즉, 같은 메서드 호출이라도 객체의 실제 타입에 따라 다른 메서드가 실행될 수 있습니다.

```
class Animal {
    void sound() {
        System.out.println("This animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("The dog says: bow wow");
    }
}
```

오버로딩은 같은 이름의 메서드를 여러 개 정의하지만, 매개변수의 수나 타입을 다르게하여 구분하는 것을 말합니다. 오버로딩을 사용하면 같은 동작을 하는 메서드라도 다양한 타입이나 매개변수의 수를 가진 입력 데이터에 대해 유연하게 대응할 수 있습니다. 오버로딩된 메서드는 컴파일 시점에 결정되는 컴파일 타임 다형성을 제공합니다.

```
class MathUtil {
    int sum(int a, int b) {
        return a + b;
    }

    int sum(int a, int b, int c) {
        return a + b + c;
    }
}
```

□ 객체지향 프로그래밍(OOP)에 대해 핵심만 간략히 설명해주세요.

우리가 실생활에서 쓰는 모든 것을 객체라 하며, 객체 지향 프로그래밍은 프로그램 구현에 필요한 객체를 파악하고 상태와 행위를 가진 객체를 만들고 각각의 객체들의 역할이 무엇인지를 정의하여 객체들 간의 상호작용을 통해 프로그램을 만드는 것을 말합니다. 즉, 기능이 아닌 객체가 중심이며 "누가 어떤 일을 할 것인가?"가 ← 핵심

특징으로는 캡슐화, 상속, 다형성, 추상화 등이 있고, 모듈 재사용으로 확장 및 유지보수가 용이합니다.

객체지향 프로그래밍(Object-Oriented Programming, OOP)은 소프트웨어 개발 방법론 중 하나로, 프로그램을 객체들의 집합으로 모델링하여 개발하는 방식입니다. 객체지향 프로그래밍의 핵심은 다음 네 가지 기본 원칙에 기반합니다:

캡슐화(Encapsulation): 객체의 데이터(속성)와 그 데이터를 처리하는 함수(메서드)를 하나로 묶어서 관리하는 것을 말합니다. 캡슐화를 통해 객체 내부 구현을 숨기고, 외부에서는 제공되는 인터페이스만을 통해 접근할 수 있게 함으로써 데이터의 안정성과 재사용성을 높입니다.

```java
public class Person {
    // private 변수로 'name'과 'age' 필드를 선언합니다.
    private String name;
    private int age;

    // 'name' 필드에 대한 getter 메서드
    public String getName() {
        return name;
    }

    // 'name' 필드에 대한 setter 메서드
    public void setName(String name) {
        this.name = name;
    }

    // 'age' 필드에 대한 getter 메서드
    public int getAge() {
        return age;
    }

    // 'age' 필드에 대한 setter 메서드
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}
```

상속(Inheritance): 한 클래스(부모 클래스)의 속성과 메서드를 다른 클래스(자식 클래스)가 상속받아 사용할 수 있게 하는 것입니다. 상속을 통해 코드의 재사용성을 높이고, 중복을 줄일 수 있습니다.

```java
// 상위 클래스인 Vehicle 정의
class Vehicle {
    // Vehicle 클래스의 필드
    protected String brand = "Ford"; // Vehicle 클래스와 이를 상속받는 클래스에서 접근 가능

    // Vehicle 클래스의 메서드
    public void honk() {
        System.out.println("Tuut, tuut!");
    }
}

// Vehicle 클래스를 상속받는 Car 클래스 정의
class Car extends Vehicle {
    // Car 클래스의 필드
    private String modelName = "Mustang";

    // Car 클래스의 메서드
    public void showModel() {
        System.out.println("Model name: " + modelName);
    }
}
```

다형성(Polymorphism): 같은 이름의 메서드가 여러 클래스에서 다른 작업을 수행할 수 있게 함으로써, 같은

인터페이스에 대해 다양한 구현을 제공할 수 있는 능력을 말합니다. 이는 오버라이딩(Overriding)과 오버로딩(Overloading)을 통해 달성됩니다.

```java
// 상위 클래스 Animal
class Animal {
    public void sound() {
        System.out.println("The animal makes a sound");
    }
}

// Animal 클래스를 상속받는 Dog 클래스
class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("The dog barks");
    }
}

// Animal 클래스를 상속받는 Cat 클래스
class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("The cat meows");
    }
}
```

추상화(Abstraction): 복잡한 현실 세계의 객체를 필요한 정보만을 추려 간략화하여 프로그램 내의 객체로 모델링하는 것입니다. 추상화를 통해 복잡성을 관리하고, 프로그램의 확장성과 유지보수성을 높일 수 있습니다. 다형성 코드와 같은 코드를 사용합니다.

객체지향 프로그래밍은 이러한 원칙을 통해 소프트웨어의 설계와 구현의 효율성을 높이며, 변경과 확장이 용이한 프로그램을 만들 수 있게 돕습니다.

□ 예외처리를 위한 try-with-resources에 대해 설명해주세요.
try-with-resources는 try-catch-finally의 문제점을 보완하기 위해 나온 개념입니다.
try(...) 안에 자원 객체를 전달하면, try블록이 끝나고 자동으로 자원 해제 해주는 기능을 말합니다.
따로 finally 구문이나 모든 catch 구문에 종료 처리를 하지 않아도 되는 장점이 있습니다.

try-with-resources는 Java 7에서 소개된 문법으로, 자동 리소스 관리를 위해 사용됩니다. 이 구문을 사용하면 try 블록이 종료될 때 자동으로 리소스를 닫아주므로, 리소스를 사용한 후 명시적으로 닫아주는 코드를 작성할 필요가 없어집니다. 이는 주로 파일 입출력(IO), 데이터베이스 연결과 같이 시스템 리소스를 사용하는 경우에 유용하며, 예외가 발생하더라도 안전하게 리소스를 해제하여 리소스 누수를 방지할 수 있습니다.

try-with-resources를 사용하기 위해서는 해당 리소스가 java.lang.AutoCloseable 또는 java.io.Closeable 인터페이스를 구현해야 합니다. 이 인터페이스들은 close() 메서드를 정의하고 있으며, try-with-resources 구문이 종

료될 때 자동으로 이 메서드가 호출됩니다.

기본 사용법
기본적인 try-with-resources 문의 사용법은 다음과 같습니다:

```
try (리소스 선언과 생성) {
    // 리소스를 사용하는 코드
} catch (예외타입 변수명) {
    // 예외 처리 코드
}
```

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFileExample {
    public static void main(String[] args) {
        String line;
        try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

이 코드에서 BufferedReader와 FileReader는 시스템 리소스를 사용하는 클래스이며, AutoCloseable 인터페이스를 구현합니다. try 블록 안에서 BufferedReader 객체가 선언되고 생성되면, try 블록의 코드가 실행된 후 자동으로 BufferedReader의 close() 메서드가 호출됩니다. 따라서 파일을 사용한 후 명시적으로 닫아줄 필요가 없습니다.

try-with-resources를 사용함으로써 코드는 더욱 간결해지고, 리소스 관리가 용이해집니다. 또한, 예외가 발생하더라도 리소스가 정확히 해제되므로 안전성이 향상됩니다.

□ 가비지 컬렉션(Garbage Collection)에 대해 간략하게 설명해 보세요.
가비지 컬렉션(Garbage Collection, GC)은 컴퓨터 프로그래밍에서 사용하지 않는 메모리를 자동으로 회수하는 과정입니다. 프로그램이 실행되는 동안 메모리를 동적으로 할당받아 사용하게 되는데, 필요하지 않게 된 메모리 영역을 계속해서 차지하고 있으면 메모리 낭비가 발생합니다. 가비지 컬렉션은 이러한 메모리를 자동으로 찾아서 해제하여, 메모리를 효율적으로 관리할 수 있게 도와줍니다.

가비지 컬렉션의 과정은 크게 세 가지 단계로 나뉜다:
- 마킹(Marking): 사용 중인 메모리(Reachable Objects)와 사용되지 않는 메모리(Unreachable Objects)를 식별한다.
- 스위핑(Sweeping): 마킹 과정에서 사용되지 않는 것으로 식별된 메모리 영역을 해제한다.
- 콤팩션(Compaction): 메모리 단편화를 줄이기 위해 사용 중인 메모리 영역을 재배치하여 메모리를 연속적

으로 만든다.
이런 메커니즘을 통해 프로그램은 메모리 누수를 방지하고, 효율적인 메모리 사용이 가능해진다. 가비지 컬렉션이 있는 프로그래밍 언어로는 자바(Java), C#, 파이썬(Python) 등이 있다.

장단점은 무엇이 있을까요?
가비지 컬렉션의 장단점을 살펴보면 다음과 같습니다:
장점
- 메모리 관리 자동화: 개발자가 메모리 관리에 신경 쓰지 않아도 되므로, 프로그래밍이 더 간단해지고 실수가 줄어든다.
- 메모리 누수 방지: 프로그램이 사용하지 않는 메모리를 자동으로 해제하여, 장기간 실행되는 애플리케이션에서 메모리 누수를 줄일 수 있다.
- 안정성 향상: 메모리가 안정적으로 관리되므로, 애플리케이션의 안정성이 향상된다.
단점
- 성능 오버헤드: 가비지 컬렉터가 메모리를 확인하고 정리하는 과정에서 추가적인 CPU 자원을 사용하므로 성능 저하가 발생할 수 있다.
비예측성: 가비지 컬렉션이 언제 발생할지 정확히 예측하기 어려워, 실시간성이 중요한 애플리케이션에서는 문제가 될 수 있다.
- 메모리 사용량 증가: 가비지 컬렉터가 메모리를 즉시 해제하지 않고 일정 시간 동안 유지할 수 있기 때문에, 일시적으로 메모리 사용량이 증가할 수 있다.
가비지 컬렉션은 프로그램의 안정성과 개발 편의성을 향상시키는 장점이 있지만, 성능 저하와 비예측성과 같은 단점도 고려해야 합니다. 따라서 특정 애플리케이션의 요구 사항에 따라 가비지 컬렉션을 사용하는 것이 적합한지 판단해야 합니다.

java로 직접 코드로 작성한 예가 있을까요?
Java에서 가비지 컬렉션을 직접 조작하는 것은 불가능하지만, 가비지 컬렉션이 작동하는 예를 보여주는 간단한 코드를 작성할 수 있습니다. 아래의 예제는 Java에서 객체를 생성하고 참조를 제거함으로써 가비지 컬렉션을 유발하는 방법을 보여줍니다.

```java
public class GarbageCollectionExample {
    public static void main(String[] args) {
        // 객체를 생성하고 참조
        MyClass obj = new MyClass("Test");

        // 객체에 대한 참조를 출력
        System.out.println(obj);

        // 객체에 대한 참조를 제거
        obj = null;

        // 가비지 컬렉션 요청
        System.gc();

        // 메모리 정리를 위해 잠시 기다림
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 이 시점에서 obj는 가비지 컬렉션에 의해 메모리에서 제거될 수 있음
        System.out.println("End of program");
    }
```

□ 불변 객체나 final을 굳이 사용해야 하는 이유가 있을까요?

불변 객체나 final 키워드를 사용해 얻는 이점은 다음과 같습니다.

1. Thread-Safe하여 병렬 프로그래밍에 유용하며, 동기화를 고려하지 않아도 된다. 왜냐하면 공유 자원이 불변이기 때문에 항상 동일한 값을 반환하기 때문이다.
2. 어떠한 예외가 발생되더라도 메소드 호출 전의 상태를 유지할 수 있어 예외 발생 전과 똑같은 상태로 다음 로직 처리가 가능하다.
3. 부수효과(변수의 값이 바뀌거나 객체의 필드 값을 설정하거나 예외나 오류가 발생하여 실행이 중단되는 현상)를 피해 오류를 최소화 할 수 있다.
4. 메소드 호출 시 파라미터 값이 변하지 않는다는 것을 보장할 수 있다.
5. 가비지 컬렉션(GC) 성능을 높일 수 있다. GC가 스캔 하는 객체의 수가 줄기 때문에 GC 수행 시 지연시간도 줄어든다.

□ 추상 클래스와 인터페이스를 설명하고, 둘의 차이에 대해서도 이야기 해 주세요.

추상 클래스는 클래스 내 추상 메소드가 하나 이상 포함되거나 abstract로 정의된 경우를 말하고, 인터페이스는 모든 메소드가 추상 메소드로만 이루어져 있는 것을 말합니다.

공통점

- new 연산자로 인스턴스 생성 불가능
- 사용하기 위해서는 하위 클래스에서 확장/구현 해야 한다.

차이점
- 인터페이스는 그 인터페이스를 구현하는 모든 클래스에 대해 특정한 메소드가 반드시 존재하도록 강제함에 있다.
- 추상클래스는 상속받는 클래스들의 공통적인 로직을 추상화 시키고, 기능 확장을 위해 사용한다.
- 추상클래스는 다중상속이 불가능하지만, 인터페이스는 다중상속이 가능하다.

추상 클래스와 인터페이스는 객체 지향 프로그래밍에서 사용되는 개념으로, 상속과 구현을 통해 코드의 재사용성과 유지 보수성을 향상시키는 역할을 합니다.

추상 클래스 (Abstract Class)
- 하나 이상의 추상 메소드(구현이 없고 선언만 있는 메소드)를 포함할 수 있는 클래스입니다.
- 인스턴스를 생성할 수 없습니다. 다른 클래스가 추상 클래스를 상속받아 구현해야 합니다.
- 상속받는 클래스는 추상 클래스의 모든 추상 메소드를 구현해야 합니다.
- 필드(상태)와 구현된 메소드(동작)도 포함할 수 있습니다.

인터페이스 (Interface)
- 모든 메소드가 추상 메소드인 순수한 추상 형태의 클래스입니다. Java 8부터는 default 메소드와 static 메소드를 가질 수 있게 되었습니다.
- 상수만을 필드로 가질 수 있습니다.
- 클래스는 여러 인터페이스를 구현할 수 있으며, 인터페이스는 다른 인터페이스를 상속할 수 있습니다.
- 인터페이스는 구현의 세부 사항을 강제하지 않고 메소드의 시그니처만을 정의합니다.

차이점
- 추상화 정도: 인터페이스는 보통 추상화 정도가 더 높습니다. 인터페이스는 구현을 전혀 포함하지 않을 수 있는 반면, 추상 클래스는 일부 구현을 포함할 수 있습니다.
- 상속과 구현: 클래스는 하나의 추상 클래스만 상속받을 수 있지만(다중 상속 불가), 여러 인터페이스를 구현할 수 있습니다.
- 구성 요소: 추상 클래스는 상태(필드)와 메소드(구현된 메소드와 추상 메소드)를 모두 가질 수 있지만, 인터페이스는 상수와 추상 메소드, default 메소드와 static 메소드도 가질 수 있습니다.
- 접근 제어자: 인터페이스의 메소드와 변수는 기본적으로 public이며, 추상 클래스에서는 접근 제어자를 다양하게 설정할 수 있습니다.

이러한 차이점들로 인해, 추상 클래스와 인터페이스는 서로 다른 경우에 사용됩니다. 추상 클래스는 비슷한 클래스들의 공통적인 부분을 추출하여 상속을 통해 코드를 재사용할 때 사용되고, 인터페이스는 서로 관련 없는 클래스들이 같은 기능을 구현하도록 강제할 때 사용됩니다.

□ GOF 패턴 중에서 싱글톤 패턴에 대해 설명해주세요.
싱글톤 패턴(Singleton Pattern)은 GoF(Gang of Four) 디자인 패턴 중 하나로, 전체 시스템에서 어떤 클래스의 인스턴스가 오직 하나만 존재하도록 보장하는 생성 패턴입니다. 이 패턴의 주된 목적은 단일 인스턴스의 전역 접근을 제어하고, 중복 생성을 방지하여 자원의 낭비를 줄이는 것입니다.

싱글톤 패턴의 특징
- 단일 인스턴스: 해당 클래스의 객체는 프로그램 전역에서 하나만 존재해야 합니다.
- 글로벌 액세스 포인트: 인스턴스에 접근할 수 있는 전역 접근 지점을 제공합니다.
- 자동 생성 방지: 생성자를 private로 선언하여 외부에서의 인스턴스 생성을 막습니다.

구현 방법 : Java에서 싱글톤 패턴을 구현하는 전형적인 방법은 다음과 같습니다:
클래스에 private 생성자를 선언하여 외부에서 인스턴스 생성을 막습니다.
클래스 내부에 private static 변수로 유일한 인스턴스를 보유합니다.
public static 메소드를 통해 이 인스턴스에 접근할 수 있는 글로벌 액세스 포인트를 제공합니다. 이 메소드는 인스턴스가 이미 생성되어 있으면 그것을 반환하고, 생성되어 있지 않으면 새로 생성하여 반환합니다.

예시 코드

```
public class Singleton {
    // 클래스 내부에서 유일한 인스턴스를 private static으로 선언
    private static Singleton instance;

    // 생성자를 private으로 선언하여 외부에서의 인스턴스 생성을 방지
    private Singleton() {}

    // 인스턴스에 접근할 수 있는 public static 메소드
    public static Singleton getInstance() {
        // 인스턴스가 null인 경우에만 생성
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

싱글톤 패턴의 사용
싱글톤 패턴은 공유 자원에 대한 일관된 접근을 제공하거나, 설정, 커넥션 풀, 로깅 등 전역에서 단일 인스턴스가 필요한 경우에 주로 사용됩니다.
싱글톤 패턴은 적절히 사용할 때 유용하지만, 전역 상태를 만들어 애플리케이션의 테스트와 유지 보수를 어렵게 할 수 있으므로 사용에 주의가 필요합니다.

□ 객체지향 프로그래밍(oop)의 설계 원칙에 대해 설명해주세요.
객체지향 프로그래밍(OOP)의 주요 설계 원칙은 소프트웨어의 확장성, 유지보수성 및 재사용성을 개선하기 위해 정립되었습니다. 이러한 원칙들은 SOLID라는 약어로 요약되며, 다음과 같습니다:

Single Responsibility Principle (SRP, 단일 책임 원칙):
한 클래스는 오직 하나의 책임을 가져야 하며, 클래스가 변경되어야 하는 이유는 단 하나여야 합니다. 이 원칙을 통해 각 클래스는 하나의 기능만을 담당하게 되므로, 시스템의 다른 부분에 미치는 영향 없이 해당 클래스를 수정하거나 개선할 수 있습니다.

Open/Closed Principle (OCP, 개방/폐쇄 원칙):
소프트웨어 구성요소(클래스, 모듈, 함수 등)는 확장에는 열려 있어야 하지만, 수정에는 닫혀 있어야 합니다.

즉, 기존의 코드를 변경하지 않고도 시스템의 기능을 확장할 수 있어야 한다는 뜻입니다.

Liskov Substitution Principle (LSP, 리스코프 치환 원칙):
부모 클래스의 객체를 자식 클래스의 객체로 대체할 수 있어야 하며, 이때 프로그램의 정확성을 해치지 않아야 합니다. 이 원칙은 상속 구조에서 자식 클래스가 부모 클래스의 역할을 온전히 수행할 수 있어야 함을 의미합니다.

Interface Segregation Principle (ISP, 인터페이스 분리 원칙):
클라이언트는 사용하지 않는 인터페이스에 의존하게 되어서는 안 됩니다. 한 인터페이스가 너무 많은 역할을 수행하지 않도록 작은 단위로 분리하는 것이 좋으며, 이를 통해 더 명확하고 타깃화된 기능을 제공할 수 있습니다.

Dependency Inversion Principle (DIP, 의존성 역전 원칙):
고수준 모듈(비즈니스 로직을 담고 있는 모듈)은 저수준 모듈(데이터를 저장하고 처리하는 모듈)에 의존해서는 안 되며, 둘 다 추상화에 의존해야 합니다. 이 원칙은 구현 세부 사항보다는 추상화에 의존해야 함을 의미하며, 시스템의 결합도를 낮추고, 유연성과 확장성을 높이는 데 도움을 줍니다.

SOLID 원칙을 준수함으로써 개발자는 더 견고하고, 관리하기 쉬우며, 확장 가능한 소프트웨어 시스템을 설계할 수 있습니다.

□ 자바의 메모리 영역에 대해 설명해주세요.
자바의 메모리 공간은 크게 Method 영역(static 영역), Stack 영역, Heap 영역으로 구분되고, 데이터 타입에 따라 할당되어 있습니다.
1) 메소드(Method) 영역 : 전역변수와 static변수를 저장하며, Method영역은 프로그램의 시작부터 종료까지 메모리에 남아있다.
2) 스택(Stack) 영역 : 지역변수와 매개변수 데이터 값이 저장되는 공간이며, 메소드가 호출될 때 메모리에 할당되고 종료되면 메모리가 해제된다. LIFO(Last In First Out) 구조를 갖고 변수에 새로운 데이터가 할당되면 이전 데이터는 지워진다.
3) 힙(Heap) 영역 : new 키워드로 생성되는 객체(인스턴스), 배열 등이 Heap 영역에 저장되며, 가비지 컬렉션에 의해 메모리가 관리되어 진다.

각 메모리 영역이 할당되는 시점
  Method 영역 : JVM이 동작해서 클래스가 로딩될 때 생성
  Stack 영역 : 메소드가 호출될 때 할당
  Heap 영역 : 런타임시 할당

다시 설명 하면
Java에서 메모리는 주로 런타임 데이터 영역으로 관리되며, 여러 부분으로 구분됩니다. 이러한 메모리 영역들은 Java 프로그램이 실행되는 동안 다양한 목적으로 사용됩니다:

1. 힙(Heap)

객체와 배열이 동적으로 할당되는 영역입니다. Java의 가비지 컬렉터에 의해 관리되며, 객체에 더 이상 참조가 없을 때 메모리가 해제됩니다. 애플리케이션 전체에서 공유되는 메모리 영역으로, 런타임에 크기가 변할 수 있습니다.
2. 스택(Stack)
각 스레드마다 실행 스택이 생성됩니다. 메소드 호출 시 해당 메소드의 로컬 변수와 매개변수가 스택 영역에 저장됩니다. 메소드가 종료되면 해당 메소드를 위해 할당된 메모리가 자동으로 해제됩니다. 스택 메모리는 LIFO(Last In First Out) 방식으로 동작합니다.
3. 메소드 영역(Method Area)
클래스 레벨의 정보(클래스, 인터페이스, 메소드, 상수 등)가 저장되는 영역입니다. 모든 스레드가 공유하는 영역으로, JVM 시작 시 생성되며, 각 클래스와 인터페이스에 대한 메타데이터를 저장합니다.
런타임 상수 풀도 이 영역에 위치하며, 리터럴과 기호 참조를 저장합니다.
4. 네이티브 메소드 스택(Native Method Stack)
Java가 아닌 다른 언어로 작성된 네이티브 메소드를 위한 스택 영역입니다. Java 네이티브 인터페이스(JNI)를 통해 호출된 네이티브 메소드의 실행 정보를 관리합니다.
5. PC 레지스터(Program Counter Register)
스레드 마다 하나씩 존재하는 영역으로, 현재 실행 중인 JVM 명령어의 주소를 가집니다. 스레드가 어떤 부분의 코드를 실행하고 있는지 추적하는 데 사용됩니다.

Java의 메모리 관리 시스템은 이러한 영역들을 효율적으로 사용하여 프로그램의 실행을 지원합니다. 가비지 컬렉션은 주로 힙 영역에서 이루어지며, 메모리 누수를 방지하고 효율적인 메모리 사용을 가능하게 합니다.

□ 클래스와 객체, 인스턴스에 대해 각각 설명하고 용어의 차이에 대해 설명해주세요.
클래스와 객체, 인스턴스 기본이해 및 차이점

클래스 (Class) : 클래스는 객체를 생성하기 위한 템플릿 또는 설계도와 같다. 클래스는 객체의 기본 형태를 정의하며, 해당 객체의 상태를 나타내는 필드(변수)와 상태를 조작할 수 있는 메서드(함수)를 포함한다.
클래스는 데이터와 행위를 하나로 묶는 캡슐화의 주요 단위이다.
예: 자동차 설계도에서 자동차의 특성(속성)과 동작(메서드)을 정의한다.

객체 (Object) : 객체는 클래스에 기반하여 생성된 실체이다. 실제 프로그램 내에서 메모리에 할당된 상태를 가지고 있는 실행 가능한 어떤 것을 말한다.
객체는 클래스의 인스턴스로도 불리며, 클래스에서 정의된 필드와 동일한 속성(상태)와 메서드(행위)를 가진다.
예: 자동차 설계도를 바탕으로 만들어진 실제 자동차.

인스턴스 (Instance) : 인스턴스는 클래스로부터 생성된 객체를 의미한다. 기술적으로 보면, 객체와 인스턴스는 같은 것을 지칭한다.
그러나 '인스턴스'라는 용어는 주로 객체가 메모리에 할당되어 실제로 생성된 상태를 강조할 때 사용된다.
예: 특정 자동차 설계도로 만들어진 특정 자동차(인스턴스)가 실제로 도로 위를 달리고 있을 때, 우리는 그것을 인스턴스라고 부를 수 있다.

용어의 차이

- 클래스는 추상적인 개념으로, 객체의 공통적인 구조와 행동을 정의한 코드 블록이다.
- 객체는 클래스의 정의를 통해 메모리에 할당된 구체적인 실체이다.
- 인스턴스는 객체가 메모리에 할당되어 실제로 생성된 것을 강조하는, 객체와 동일한 실체를 다른 관점에서 부르는 용어이다.

간단히 말해, 클래스는 객체의 설계도이며, 객체는 설계도를 바탕으로 만들어진 실체이고, 인스턴스는 그 실체가 메모리 상에 할당된 것을 특히 강조하는 표현이다.

자바 코드로 작성해서 위 세 개를 구분해 보자.

```
class Car {    // 클래스 정의: Car라는 이름의 클래스를 정의합니다.
    // 필드(상태): Car 클래스의 속성들을 정의합니다.
    String color;
    String manufacturer;

    // 생성자: Car 객체를 생성할 때 초기 상태를 설정합니다.
    Car(String color, String manufacturer) {
        this.color = color;
        this.manufacturer = manufacturer;
    }

    // 메서드(행위): Car 클래스의 객체가 수행할 수 있는 행동을 정의합니다.
    void drive() {
        System.out.println(this.color + " " + this.manufacturer + " is driving.");
    }
}

public class Main {
    public static void main(String[] args) {
        // 객체 생성: Car 클래스를 바탕으로 new 키워드를 사용해 객체를 생성합니다.
        Car myCar = new Car("Red", "Toyota");

        // 객체 사용: myCar 객체의 drive 메서드를 호출합니다.
        myCar.drive();

        // 인스턴스 확인: myCar는 Car 클래스의 인스턴스입니다.
        if (myCar instanceof Car) {
            System.out.println("myCar is an instance of Car class.");
        }
    }
}
```

여기서 Car는 클래스이며 color와 manufacturer라는 상태(필드)와 drive()라는 행위(메서드)를 정의한다.
main 메서드에서 new Car("Red", "Toyota")를 통해 Car 클래스의 객체를 생성하고, myCar라는 참조 변수에 할당한다.
이때 myCar는 Car 클래스의 객체이며, 동시에 Car 클래스의 인스턴스이다.
instanceof 연산자를 사용하여 myCar가 실제로 Car 클래스의 인스턴스인지 확인할 수 있다.
myCar.drive()를 호출함으로써 myCar 객체의 drive 메서드를 실행할 수 있다.

이 코드는 클래스를 정의하고, 그 클래스를 기반으로 객체를 생성하여 인스턴스를 만드는 과정을 보여준다.
한 마디로 클래스는 설계도, 객체는 설계도를 기반으로 생성된 실체, 인스턴스는 그 실체가 메모리에 할당되어 생성된 상태를 의미한다.

□ 생성자(Constructor)에 대해 설명해주세요.
생성자(Constructor)는 객체지향 프로그래밍에서 클래스의 인스턴스, 즉 객체를 생성할 때 호출되는 특별한 종류의 메서드입니다. 생성자의 주요 목적은 새로 생성된 객체의 초기화를 담당하는 것이며, 객체가 올바른 상태로 사용될 수 있도록 필요한 값들을 설정합니다.

자바에서 생성자의 특징은 다음과 같습니다:
- 생성자의 이름은 클래스의 이름과 동일해야 합니다.
- 생성자는 리턴 타입이 없으며, 심지어 void도 사용하지 않습니다.
- 생성자는 오버로드(Overloading)될 수 있습니다, 즉 동일한 클래스 내에 여러 생성자를 정의할 수 있으며, 각각 다른 인자 목록을 가질 수 있습니다.
- 객체 생성 시 new 키워드와 함께 생성자가 호출되며, 이때 생성자가 실행됩니다.
클래스에 생성자가 명시적으로 정의되어 있지 않은 경우, 컴파일러는 인자가 없는 기본 생성자를 제공합니다.
- 생성자는 객체의 속성을 초기화하는데 사용되며, 필요에 따라 초기화 로직을 포함할 수 있습니다.
- 생성자는 다른 생성자를 호출할 수 있는데, 이를 '생성자 체이닝(Constructor Chaining)'이라고 하며, this() 구문을 통해 같은 클래스의 다른 생성자를 호출합니다.

생성자 예제
```
public class Book {

    String title;
    String author;

    // 기본 생성자
    public Book() {
        this.title = "Unknown";
        this.author = "Unknown";
    }

    // 파라미터가 있는 생성자
    public Book(String title, String author) {
```

```
        this.title = title;
        this.author = author;
    }

    public void displayInfo() {
        System.out.println("Book: " + title + ", Author: " + author);
    }

    public static void main(String[] args) {
        // 기본 생성자를 사용하여 객체 생성
        Book unknownBook = new Book();
        unknownBook.displayInfo();

        // 파라미터가 있는 생성자를 사용하여 객체 생성
        Book myBook = new Book("1984", "George Orwell");
        myBook.displayInfo();
    }
}
```

위 예제에서 Book 클래스에는 두 개의 생성자가 정의되어 있습니다. 하나는 기본 생성자이며, 다른 하나는 title과 author를 인자로 받는 생성자입니다. 객체가 생성될 때 이 생성자들 중 하나가 호출되어 객체의 상태를 초기화합니다.

□ 자바에서 Wrapper Class란 무엇인가요? 그리고 Boxing과 UnBoxing은 뭘 의미하는지 설명해보세요.

Wrapper Class란?

Wrapper 클래스는 Java의 기본 데이터 타입(primitive data types)을 객체로 감싸는 클래스입니다. Java는 객체 지향 언어이기 때문에 모든 것을 객체로 다루는 것이 바람직합니다. 하지만 효율성을 위해 기본 데이터 타입을 제공하고, 때로는 이러한 기본 타입들을 객체로 다뤄야 할 필요가 있습니다(예: 컬렉션에서의 사용). 이때 기본 데이터 타입을 객체로 다루기 위해 Wrapper 클래스가 사용됩니다.

각 기본 데이터 타입에는 대응하는 Wrapper 클래스가 존재합니다:

byte -> Byte
short -> Short
int -> Integer
long -> Long
float -> Float
double -> Double
char -> Character
boolean -> Boolean

Boxing : Boxing은 기본 데이터 타입을 해당하는 Wrapper 클래스의 객체로 변환하는 과정입니다. 이는 명시

적으로 생성자를 사용하거나 Java 5부터 제공하는 자동 박싱(auto-boxing)을 통해 수행됩니다.

```
int i = 10;
// Boxing: 기본 타입을 Wrapper 클래스 객체로 변환
Integer integerObject = new Integer(i); // 명시적 Boxing
Integer autoBoxing = i; // 자동 Boxing
```

Unboxing : Unboxing은 Wrapper 클래스의 객체를 다시 기본 데이터 타입으로 변환하는 과정입니다. 이 역시 명시적으로 intValue()와 같은 메서드를 호출하거나 자동 언박싱(auto-unboxing)을 사용할 수 있습니다.

```
Integer integerObject = new Integer(10);
// Unboxing: Wrapper 클래스 객체를 기본 타입으로 변환
int i = integerObject.intValue(); // 명시적 Unboxing
int autoUnboxing = integerObject; // 자동 Unboxing
```

Boxing과 Unboxing은 각각 기본 데이터 타입과 객체 사이의 변환을 가능하게 해 주는 과정입니다. 하지만 불필요한 Boxing과 Unboxing은 성능에 부정적인 영향을 줄 수 있기 때문에, 자동으로 일어나는 이러한 변환에 주의를 기울여야 합니다.

□ 쓰레드에서 Synchronized란 무엇인지 아는 대로 말씀해 보세요.

Synchronized는 자바에서 동기화를 구현하는 키워드로, 멀티 쓰레드 환경에서 여러 쓰레드가 동시에 같은 객체의 메모리를 접근하는 것을 방지하기 위해 사용됩니다. 쓰레드 동기화는 공유 자원에 대한 동시 접근을 제어하여 데이터의 일관성과 무결성을 보장하는 데 중요합니다.

Synchronized의 작동 방식

Synchronized 키워드가 사용될 때, 객체에 대한 잠금(Lock)을 획득하게 됩니다. 한 쓰레드가 synchronized 메소드나 블록에 들어가면, 그 쓰레드는 해당 객체의 잠금을 얻습니다. 잠금을 소유한 쓰레드만이 synchronized 블록의 코드를 실행할 수 있으며, 다른 쓰레드들은 잠금이 해제될 때까지 대기해야 합니다.

Synchronized 사용 방법

1) 메소드 동기화: 메소드 전체를 동기화할 때 메소드 선언에 synchronized 키워드를 추가합니다.

```
public synchronized void method() {
    // 동기화된 코드
}
```

2) 블록 동기화: 특정 코드 블록에 대해 동기화를 구현하고 싶을 때 synchronized 블록을 사용합니다.

```
public void method() {
    synchronized(this) {
        // 동기화된 코드 블록
    }
}
```

여기서 this는 현재 객체의 참조를 의미하며, 다른 객체의 참조로 대체될 수 있습니다.

Synchronized의 중요성 : 멀티 쓰레드 환경에서 여러 쓰레드가 같은 객체의 상태를 동시에 변경하려고 하면, 데이터의 불일치 문제가 발생할 수 있습니다. 이를 경쟁 조건(Race Condition)이라고 합니다. Synchronized를 사용하면 한 번에 하나의 쓰레드만이 객체의 상태를 변경할 수 있기 때문에 경쟁 조건을 방지할 수 있습니다.
성능 고려사항 : 동기화는 필요한 경우에만 사용해야 합니다. Synchronized는 필요한 연산에 대해 독점적인 액세스를 제공하지만, 성능에 부담을 줄 수 있습니다. 모든 쓰레드가 동기화된 메소드나 블록에 순차적으로 접근해야 하기 때문에, 시스템의 처리량(Throughput)이 감소할 수 있습니다. 따라서 성능과 안정성 사이에서 적절한 균형을 찾는 것이 중요합니다.

□ new String()과 리터럴("")의 차이에 대해 설명해주세요.
자바에서 문자열을 생성하는 방법에는 주로 두 가지가 있습니다: new 연산자를 사용하는 방법과 리터럴을 사용하는 방법입니다. 이 두 방법 사이에는 중요한 차이가 있습니다.

1) new String() 사용하기 : new 연산자를 사용하여 String 객체를 생성하면, 메모리의 힙(Heap) 영역에 새로운 문자열 객체가 생성됩니다. 이 방법을 사용하면 동일한 문자열이라도 매번 새로운 객체가 생성됩니다.

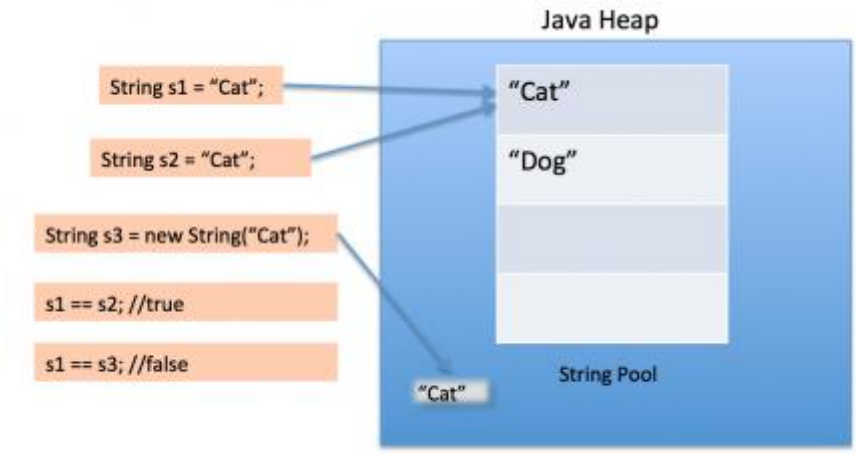
String s1 = new String("Cat");
String s2 = new String("Cat");
위의 코드에서 s1과 s2는 내용은 같지만, 두 개의 서로 다른 String 객체를 참조합니다. s1 == s2의 결과는 false가 됩니다.

2) 리터럴("") 사용하기 : 문자열 리터럴을 사용하면, JVM은 문자열을 문자열 상수 풀(String Constant Pool)에 저장합니다. 이 풀은 메모리 효율성을 위해 JVM의 메소드 영역 내에 존재합니다. 문자열 리터럴을 사용하여 동일한 문자열을 생성하려고 하면, JVM은 먼저 문자열 상수 풀을 확인하여 이미 존재하는지 검사합니다. 이미 존재하는 경우, 그 존재하는 객체의 참조를 반환합니다. 따라서 같은 리터럴로 생성된 문자열은 동일한 메모리 주소를 공유합니다.

String s3 = "Cat";
String s4 = "Cat";
s3과 s4는 둘 다 문자열 "Hello"를 참조하지만, new String()을 사용하지 않았기 때문에 동일한 문자열 상수 풀 내의 객체를 가리킵니다. 따라서 s3 == s4의 결과는 true가 됩니다.

요약
new String("..."): 항상 새로운 String 객체를 생성합니다.
"...": 문자열 상수 풀에서 동일한 문자열에 대한 참조를 재사용합니다.
리터럴 방식은 메모리를 절약할 수 있고, 같은 리터럴 값에 대해 == 연산자로 비교할 때 true를 반환하므로, 일반적으로 문자열을 생성할 때 이 방식이 권장됩니다. 그러나 특정 상황에서는 new String()을 사용하여 명시적으로 새로운 문자열 객체를 생성해야 할 필요가 있을 수도 있습니다.

다시 말해 new String()은 new 키워드로 새로운 객체를 생성하기 때문에 Heap 메모리 영역에 저장되고, ""는 Heap 안에 있는 String Constant Pool 영역에 저장됩니다.



□ 문자열 처리를 위한 클래스에서 String, StringBuffer, StringBuilder의 차이를 설명해주세요.
자바에서는 문자열을 다루기 위해 String, StringBuffer, StringBuilder 클래스를 제공합니다. 이 세 클래스는 사용 목적과 내부 작동 방식에 있어서 중요한 차이점이 있습니다.

1) String 클래스 : String 클래스는 불변(immutable)입니다. 이는 한 번 생성된 String 객체의 내용이 변경되지 않는다는 것을 의미합니다. 문자열의 내용을 변경할 때마다 사실상 새로운 String 객체가 생성되고, 이전 객체는 가비지 컬렉션을 위해 버려집니다. String은 문자열 연산이 적고 읽기만 많은 경우에 적합합니다.

```
String s = "Hello";
s += " World"; // 새로운 String 객체가 생성됨
```

2) StringBuffer 클래스 : StringBuffer는 가변(mutable)하며, 동기화(synchronized)가 적용됩니다. 즉, 멀티 쓰레드 환경에서 안전하며(쓰레드-세이프), 여러 쓰레드가 동시에 접근하여도 데이터 무결성을 유지합니다. StringBuffer는 내부적으로 문자열을 추가하거나 변경할 때 기존의 데이터를 수정할 수 있는 메모리 공간을 가지고 있어, String에 비해 성능이 좋습니다. 그러나 동기화로 인한 오버헤드가 존재합니다.

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World"); // 동일 객체 내부에서 문자열을 변경
```

3) StringBuilder 클래스 : StringBuilder 역시 가변적이며, StringBuffer와 유사하지만 동기화를 제공하지 않습니다. 이는 StringBuilder가 멀티 쓰레드 환경에서 안전하지 않음을 의미하지만(쓰레드-세이프하지 않음), StringBuffer에 비해 더 빠른 성능을 제공합니다. 따라서 단일 쓰레드 환경에서 문자열을 자주 변경해야 할 때 선호됩니다.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // 동일 객체 내부에서 문자열을 변경
```

요약
- String: 불변, 새로운 문자열을 생성할 때마다 새로운 객체가 필요. 메모리와 성능에 부담을 줄 수 있음.
- StringBuffer: 가변, 동기화 적용, 멀티 쓰레드 환경에서 쓰레드-세이프.
- StringBuilder: 가변, 동기화 미적용, 멀티 쓰레드 환경에서 쓰레드-세이프하지 않지만 성능이 좋음.

실제 애플리케이션에서는 이러한 차이를 고려하여 상황에 맞는 클래스를 선택하여 사용합니다.

□ String 객체가 불변인 이유에 대해 아는 대로 설명해주세요.
String 객체가 불변(immutable)인 이유는 여러 가지가 있으며, 자바의 설계 철학과 관련이 있습니다. 불변성은 String을 안전하고, 효율적이며, 사용하기 쉽게 만드는 데 중요한 역할을 합니다.

- 캐싱과 재사용 : 불변 객체는 정보가 변경되지 않기 때문에, 같은 내용의 문자열이 필요할 때 재사용할 수 있습니다. 예를 들어, 자바의 문자열 리터럴은 문자열 상수 풀(String Constant Pool)에 저장되어, 동일한 문자열 리터럴은 메모리에 단 한 번만 존재합니다. 이는 메모리 사용량을 줄이고 성능을 향상시킬 수 있습니다.
- 보안 : String의 불변성은 자바의 보안 기능에 필수적입니다. 예를 들어, 파일 경로나 네트워크 연결의 자원 위치 등 중요한 정보를 String으로 관리할 때, 불변성이 보장되기 때문에 정보가 예상치 못하게 변경되어 보안 문제가 발생하는 것을 방지할 수 있습니다.
- 스레드 안전성 : 불변 객체는 별도의 동기화 없이 멀티스레드 환경에서 안전하게 공유될 수 있습니다. String 객체의 상태는 생성 후 변경되지 않으므로, 동시에 여러 스레드가 String 객체를 읽어도 문제가 발생하지 않습니다.
- 설계 단순화 및 신뢰성 : 불변 객체는 설계를 단순하게 만들며, 예상치 못한 변경으로부터 발생할 수 있는 버그를 줄여줍니다. 한 번 만들어진 String 객체는 유효성을 검증한 후 변하지 않기 때문에, 그 객체를 신뢰하고 코드를 작성할 수 있습니다.
- 해시코드 캐싱 : 불변성은 String 객체의 해시코드를 안전하게 캐싱하는 데 도움을 줍니다. String이 불변이기 때문에, hashCode() 메서드가 호출될 때 계산된 해시코드 값을 내부적으로 캐싱할 수 있고, 이후에 같은 String 객체에 대한 hashCode() 호출은 캐싱된 값을 바로 반환할 수 있어서 성능이 향상됩니다.
- 프로그래밍 편의성 : 문자열을 불변으로 처리하면, 프로그래머는 문자열이 중간에 변경되지 않을 것이라고 가정하고 안전하게 코드를 작성할 수 있습니다. 불변성은 함수형 프로그래밍 패러다임과도 잘 어울리며, 불변 데이터 구조를 선호하는 프로그래밍 스타일을 지원합니다.

이러한 이유들로 인해 String 클래스는 불변으로 설계되었으며, 이는 자바 프로그래밍에서 String을 다루는 방식에 광범위한 영향을 미쳤습니다.

□ 자바의 접근 제한자(Access Modifier)에 대해 간략하게 설명해 주세요.
자바(Java)에서 접근 제한자(Access Modifier)는 클래스, 변수, 메소드, 생성자 등의 멤버에 대한 접근을 제어하는 키워드입니다. 접근 제한자를 사용하면 클래스 내부 또는 외부에서 해당 멤버에 접근할 수 있는 범위를 정할 수 있습니다. 주로 다음과 같은 네 가지 접근 제한자가 사용됩니다:

private: 가장 제한적인 접근 수준으로, private으로 선언된 멤버는 해당 클래스 내부에서만 접근할 수 있습니다. 클래스 외부에서는 접근할 수 없습니다.

default (package-private): 접근 제한자를 명시하지 않으면 기본적으로 default 접근 수준이 적용됩니다. 같은 패키지 내의 클래스에서만 접근할 수 있으며, 다른 패키지의 클래스에서는 접근할 수 없습니다.

protected: protected로 선언된 멤버는 같은 패키지 내의 다른 클래스 또는 다른 패키지의 서브클래스에서 접

근할 수 있습니다.

public: 가장 개방적인 접근 수준으로, public으로 선언된 멤버는 어떤 클래스에서도 접근할 수 있습니다.

이러한 접근 제한자들은 클래스의 캡슐화를 강화하고, 객체 지향 프로그래밍에서의 데이터 은닉 및 추상화를 지원합니다. 즉, 불필요한 정보는 숨기고 필요한 정보만을 외부에 노출시켜 프로그램의 안정성과 유지보수성을 높이는 데 기여합니다.

□ 클래스 멤버 변수 초기화 순서에 대해 설명해주세요.
자바에서 클래스 멤버 변수(필드) 초기화 순서는 다음과 같은 과정을 따릅니다:

1) 정적 변수(static fields) 초기화: 클래스가 로딩될 때 정적 필드는 기본값으로 초기화됩니다(예: int는 0, boolean은 false, 객체 참조는 null). 이후에 정적 초기화 블록(static initializer blocks)이 실행됩니다. 이 초기화는 클래스가 처음 로딩될 때 단 한 번만 수행됩니다.

2) 인스턴스 변수 초기화: 객체가 생성될 때 인스턴스 변수는 기본값으로 초기화됩니다. 각 인스턴스 변수의 선언에 명시적으로 초기화 값이 제공된 경우, 그 값으로 초기화됩니다.

3) 초기화 블록 실행: 클래스 내에 있는 초기화 블록(인스턴스 초기화 블록)이 선언된 순서대로 실행됩니다. 이 블록들은 객체의 생성자보다 먼저 실행됩니다.

4) 생성자 실행: 마지막으로 클래스의 생성자가 실행됩니다. 생성자는 인스턴스 변수를 필요에 따라 초기화하거나 다른 초기화 작업을 수행할 수 있습니다. 생성자 내에서의 초기화는 모든 필드가 기본값이나 명시적 값으로 초기화된 후에 수행됩니다.

이러한 순서를 통해 자바는 객체가 사용되기 전에 모든 필드가 적절히 초기화되도록 보장합니다. 정적 필드와 정적 초기화 블록은 해당 클래스가 처음 사용될 때 한 번만 실행되며, 인스턴스 변수와 초기화 블록은 객체가 생성될 때마다 실행됩니다.

□ static에 대해 설명해주세요. 사용하는 이유에 대해서도 함께 설명해 보세요
static 키워드는 자바에서 정적 멤버(필드, 메소드, 블록)를 선언하는 데 사용됩니다. 정적 멤버는 클래스 당 하나만 생성되며, 해당 클래스의 모든 인스턴스에 의해 공유됩니다. 여기에는 몇 가지 중요한 특징과 사용 이유가 있습니다:

static 키워드의 특징
1) 정적 필드(static fields): 클래스의 모든 인스턴스에 공통된 값을 유지해야 할 때 사용됩니다. 모든 인스턴스가 동일한 값을 공유하며, 이 값은 해당 클래스의 모든 객체에 대해 공통입니다.
2) 정적 메소드(static methods): 인스턴스 변수나 메소드에 접근하지 않고 클래스 수준의 작업을 수행하는 메소드에 사용됩니다. 객체를 생성하지 않고도 호출할 수 있습니다.
3) 정적 초기화 블록(static initialization blocks): 정적 필드를 초기화하거나 정적 초기화 시 복잡한 로직을 실

행할 때 사용됩니다. 클래스가 로딩될 때 단 한 번만 실행됩니다.

static 사용하는 이유
1) 메모리 효율성: 정적 필드나 메소드는 클래스가 로딩될 때 메모리에 한 번만 할당되며, 모든 인스턴스가 이를 공유합니다. 이는 각 인스턴스마다 동일한 값을 별도로 저장할 필요가 없어 메모리 사용을 최적화합니다.
2) 접근성: 정적 멤버는 객체의 생성 없이도 접근할 수 있습니다. 이는 유틸리티 함수나 상수 값을 관리할 때 유용하며, 예를 들어 Math.PI 또는 Math.sqrt()와 같은 Math 클래스의 정적 필드와 메소드가 이에 해당합니다.
3) 데이터 공유: 정적 필드는 해당 클래스의 모든 인스턴스 간에 데이터를 공유하는 데 사용됩니다. 예를 들어, 모든 인스턴스가 공유해야 하는 카운터나 설정 정보 등을 관리할 때 유용합니다.
static 키워드의 사용은 클래스 설계와 프로그램의 동작 방식에 중대한 영향을 미칠 수 있으므로, 필요한 경우와 상황에 맞게 신중하게 사용해야 합니다.

□ Inner Class(내부 클래스)에 대해 설명해주세요.
자바에서 내부 클래스(Inner Class)는 한 클래스 내부에 정의된 다른 클래스를 말합니다. 내부 클래스는 그것을 감싸고 있는 외부 클래스(Enclosing Class)의 멤버에 접근할 수 있는 특별한 권한을 가지며, 특정한 상황에서 유용하게 사용됩니다. 내부 클래스는 여러 종류가 있고, 각각의 용도와 특징이 다릅니다.

내부 클래스의 종류
1) 비정적 내부 클래스(Non-static Nested Class) 또는 인스턴스 내부 클래스(Instance Inner Class)
- 이 내부 클래스는 외부 클래스의 인스턴스와 연결되어 있습니다.
- 비정적 내부 클래스의 인스턴스는 외부 클래스의 인스턴스에 종속적입니다.
- 이 클래스는 외부 클래스의 인스턴스 변수와 메서드에 직접 접근할 수 있습니다.

2) 정적 내부 클래스(Static Nested Class)
- 정적 내부 클래스는 외부 클래스의 인스턴스와 독립적으로 존재할 수 있습니다.
- 이 클래스는 외부 클래스의 정적 멤버에만 접근할 수 있습니다.
- 정적 내부 클래스는 외부 클래스의 정적 필드나 메소드와 관련된 작업에 사용됩니다.

3) 지역 내부 클래스(Local Inner Class)
- 지역 내부 클래스는 메서드 내에서 정의되며, 그 메서드의 실행이 끝나면 유효 범위를 벗어나게 됩니다.
- 특정 메서드 내에서만 사용될 클래스를 정의할 때 사용합니다.

4) 익명 내부 클래스(Anonymous Inner Class)
- 이름이 없는 클래스로, 주로 한 번만 사용되는 인스턴스를 생성할 때 사용됩니다.
- 이 클래스는 확장하려는 클래스 또는 구현하려는 인터페이스의 새 인스턴스를 직접 선언하는 데 사용됩니다.
- GUI 이벤트 처리나 스레드 객체를 간편하게 생성하는 데 자주 사용됩니다.

내부 클래스의 장점

- 캡슐화: 내부 클래스를 사용하면 외부에서는 접근할 필요가 없는 내부 작업을 클래스 내부에 숨길 수 있습니다. 이는 코드의 캡슐화를 증진시킵니다.
- 가독성과 유지보수: 서로 관련 있는 클래스를 논리적으로 묶어서 관리할 수 있어, 코드의 구조를 더 명확하게 하고 유지보수를 용이하게 합니다.
- 특정 범위에서만 사용되는 클래스 정의: 특정한 범위나 상황 내에서만 사용되는 클래스를 정의할 때 유용합니다.

내부 클래스는 특정한 상황에서 코드의 구조와 설계를 개선하는 데 큰 도움을 줄 수 있으나, 사용할 때는 클래스 간의 관계와 범위, 그리고 캡슐화의 원칙을 고려해야 합니다.

□ 리플렉션(Reflection)이 무엇이고 어디에 사용하는지 설명해주세요.
리플렉션이란 구체적인 클래스 타입을 알지 못해도 그 클래스의 메소드, 타입, 변수들에 접근할 수 있도록 해주는 자바 API 입니다.
리플렉션(Reflection)은 자바에서 클래스, 인터페이스, 메소드, 필드 등의 메타데이터를 런타임에 조회하고 조작할 수 있는 기능을 말합니다. 이 기능을 사용하여 프로그램에서 정의된 클래스의 이름, 메소드, 필드, 생성자 등의 정보를 런타임에 알아내고, 이를 통해 동적으로 객체를 생성하거나 메소드를 호출하고, 필드에 접근하거나 수정하는 것이 가능합니다.

리플렉션의 주요 사용 사례:
- 디버깅과 테스팅: 개발 중이나 테스트 단계에서 프로그램의 상태를 검사하거나 수정할 때 유용합니다. 예를 들어, private 메소드나 필드에 접근하여 테스트 코드를 작성할 때 리플렉션을 사용할 수 있습니다.
- 동적인 클래스 로딩: 클래스 이름을 런타임에 받아 해당 클래스의 객체를 생성하고 사용해야 할 때 리플렉션을 사용합니다. 이는 플러그인 아키텍처 또는 확장 가능한 어플리케이션을 개발할 때 유용하며, 실행 시점에 클래스를 동적으로 로드하고 인스턴스화합니다.
- IoC 컨테이너(제어의 역전): 스프링(Spring) 같은 프레임워크에서는 IoC(Inversion of Control) 컨테이너를 구현하는 데 리플렉션을 사용합니다. 이를 통해 객체의 생명주기를 관리하고, 의존성을 주입(Dependency Injection)합니다.
- API와 라이브러리: 다양한 API와 라이브러리에서 리플렉션을 사용하여 유연성을 제공합니다. 예를 들어, JSON 라이브러리는 리플렉션을 이용해 JSON 문자열을 자바 객체로 변환하거나 그 반대로 작업을 수행할 수 있습니다.

리플렉션의 단점:
- 성능 저하: 리플렉션을 사용하는 작업은 직접적인 코드 실행보다 더 많은 시간이 걸립니다. 리플렉션은 런타임에 타입 체크와 접근을 처리하기 때문에 성능 오버헤드가 발생할 수 있습니다.
- 보안 제약: 리플렉션을 사용하면 private 메소드와 필드에 접근할 수 있어, 보안 위험을 초래할 수 있습니다. 이로 인해 코드가 예상치 못한 방식으로 실행될 가능성이 있습니다.
유지보수의 어려움: 리플렉션을 사용한 코드는 읽기 어렵고 디버깅이 복잡해질 수 있으므로, 유지보수가 어려워질 수 있습니다.

리플렉션은 강력한 기능을 제공하지만, 성능과 보안, 유지보수 측면에서 고려해야 할 사항이 많기 때문에 신중하게 사용해야 합니다. 필요한 경우에만 사용하고, 그 사용을 최소화하는 것이 좋습니다.

□ 자바에서 Error와 Exception의 차이를 설명해주세요.

Error는 실행 중 일어날 수 있는 치명적 오류를 말합니다. 컴파일 시점에 체크할 수 없고, 오류가 발생하면 프로그램은 비정상 종료되며 예측 불가능한 UncheckedException에 속합니다.

반면, Exception은 Error보다 비교적 경미한 오류이며, try-catch를 이용해 프로그램의 비정상 종료를 막을 수 있습니다.

자바에서 Error와 Exception은 모두 예외 처리의 형태이지만, 그 목적과 사용법에 차이가 있습니다.

1) Error :

Error는 시스템 레벨의 심각한 문제를 나타내며, 프로그램의 정상적인 흐름을 방해합니다.

주로 자바 런타임 시스템에서 발생하는 것으로, 개발자가 코드를 통해 처리할 수 없는 심각한 문제를 의미합니다.

예를 들어, OutOfMemoryError는 JVM이 더 이상 메모리를 할당할 수 없을 때 발생하고, StackOverflowError는 스택 메모리가 넘쳐흘러 발생합니다.

이런 오류들은 보통 애플리케이션에서 복구되기 어려우므로, 일반적으로 개발자가 직접 처리하지 않습니다.

2) Exception :

Exception은 애플리케이션이 실행 중에 처리할 수 있는 상황을 나타내며, 예외적인 조건이나 상황에서 발생합니다.

Exception은 다시 두 가지로 나뉩니다:

- Checked Exception: 명시적인 예외 처리가 필요한 예외입니다. 컴파일 시점에 처리 여부를 검사합니다. 예를 들어, IOException이나 SQLException 등이 있으며, 이들은 try-catch 블록으로 처리하거나 throws 키워드를 사용하여 호출한 메소드로 예외를 전달해야 합니다.
- Unchecked Exception (RuntimeException): 실행 시점(Runtime)에 발생할 수 있는 예외입니다. 컴파일러가 예외 처리를 강제하지 않습니다. 예를 들어, NullPointerException, ArrayIndexOutOfBoundsException 등이 있으며, 이러한 예외들은 주로 프로그램의 버그 때문에 발생합니다.

정리 : Error는 시스템 레벨에서 발생하는, 프로그램에서 복구할 수 없는 심각한 문제를 나타냅니다.

Exception은 애플리케이션 레벨에서 발생하며, 처리할 수 있는 예외적인 상황을 나타냅니다. Checked Exception은 명시적인 처리가 필요하고, Unchecked Exception은 주로 프로그램의 버그로 인해 발생합니다.

따라서, Error는 처리할 수 없는 반면 Exception은 적절한 예외 처리를 통해 애플리케이션의 흐름을 관리할 수 있습니다.

참고 : 예외처리

예외 처리(Exception Handling)는 프로그램 실행 중에 발생할 수 있는 예외적인 상황(예외)에 대응하여 프로그램의 정상적인 흐름을 유지하고자 하는 프로그래밍 기법입니다. 자바에서 예외 처리는 try, catch, finally, throw, throws 키워드를 사용하여 구현합니다.

예외 처리의 주요 구성 요소

try 블록 : 예외가 발생할 수 있는 코드를 포함합니다. 이 블록 내에서 예외가 발생하면 즉시 실행이 중단되고 해당 예외를 처리할 수 있는 catch 블록으로 제어가 이동합니다.

catch 블록 : try 블록 내에서 발생한 예외를 처리하는 코드를 포함합니다. 각 catch 블록은 처리할 수 있는 예외 타입을 명시합니다. 여러 예외를 처리하기 위해 여러 catch 블록을 사용할 수 있습니다.

finally 블록 : 예외 발생 여부와 관계없이 실행되어야 하는 코드를 포함합니다. 주로 리소스 해제, 파일 닫기, DB 커넥션 닫기 등의 정리 작업에 사용됩니다.

throw : 프로그래머가 직접 예외를 발생시키기 위해 사용합니다. 예외 상황을 인위적으로 생성할 때 사용하며, 생성된 예외 인스턴스를 throw 키워드와 함께 사용합니다.

throws : 메소드 선언 시, 해당 메소드에서 처리하지 않고 호출한 곳으로 예외를 전달(던짐)하기 위해 사용됩니다. 메소드가 발생시킬 수 있는 예외 타입을 명시하여, 메소드를 사용하는 측에서 이를 처리하도록 합니다.

예외 처리의 중요성
- 프로그램의 안정성과 신뢰성 향상: 예외 처리를 통해 예외적인 상황을 적절히 관리함으로써 프로그램의 비정상적인 종료를 방지하고 더 안정적으로 운영할 수 있습니다.
오류 진단과 디버깅 용이성: 예외 처리를 사용하면 오류 발생 위치와 원인을 쉽게 파악할 수 있어, 프로그램의 디버깅과 유지보수가 용이해집니다.
- 리소스 관리: finally 블록을 통해 예외 발생 여부와 상관없이 리소스를 안전하게 해제하고, 메모리 누수를 방지할 수 있습니다.

예외 처리는 프로그램의 안정성과 신뢰성을 높이는 중요한 요소로, 효과적인 예외 처리 로직을 설계하는 것이 중요합니다.

□ CheckedException과 UnCheckedException의 차이를 설명해주세요.
자바에서 예외(Exception)는 크게 Checked Exception과 Unchecked Exception으로 나눌 수 있으며, 이 둘은 처리 방법과 목적에 있어서 차이가 있습니다.

1) Checked Exception은 컴파일 시점에 처리를 강제하는 예외입니다. 이러한 예외는 일반적으로 프로그램 외부의 요인으로 인해 발생하며, 개발자가 예외 상황을 예측하고 명시적으로 처리할 수 있도록 설계되었습니다. 예를 들어, 파일을 읽거나 쓸 때 발생할 수 있는 IOException이나, 데이터베이스 접근 시 발생할 수 있는 SQLException 등이 Checked Exception에 해당합니다.
Checked Exception을 발생시킬 수 있는 메소드를 사용할 때는 try-catch 블록으로 예외를 처리하거나, throws 키워드를 사용해 해당 예외를 메소드의 호출자에게 전달할 의무가 있습니다.

2) Unchecked Exception은 런타임 시점에 발생하는 예외로, 컴파일러가 예외 처리를 강제하지 않습니다. 주로 프로그램의 버그나 잘못된 로직으로 인해 발생합니다. RuntimeException 클래스와 그 서브클래스들이 Unchecked Exception에 해당합니다.
예를 들어, 배열의 범위를 넘어서 접근하려고 할 때 발생하는 IndexOutOfBoundsException이나, null 참조를 통해 메소드를 호출하려고 할 때 발생하는 NullPointerException 등이 이에 속합니다.
Unchecked Exception은 개발자가 예측하지 못한 상황에서 발생하는 것이라고 가정하기 때문에 명시적인 처리를 강제하지 않으며, 이를 명시적으로 처리하는 것은 개발자의 선택에 달려 있습니다.

차이점 요약

- 처리 강제 여부: Checked Exception은 명시적인 처리가 필요하지만, Unchecked Exception은 그렇지 않습니다.
- 발생 시점: Checked Exception은 주로 프로그램 외부의 상황으로 인해 발생하는 반면, Unchecked Exception은 프로그램 내부의 로직 오류로 인해 발생합니다.
- 목적: Checked Exception은 예외적인 상황을 안전하게 처리하도록 강제하여 프로그램의 안정성을 높이는 반면, Unchecked Exception은 주로 프로그램의 오류를 나타내며, 이는 주로 개발자의 실수에서 비롯됩니다.

이러한 차이를 이해하는 것은 자바에서 효과적인 예외 처리 전략을 수립하는 데 중요합니다.

□ Optional API에 대해 설명해주세요.

Optional은 자바 8부터 도입된 클래스로, null이 될 수 있는 객체를 감싸는 래퍼(wrapper) 클래스입니다. Optional을 사용하면 null을 직접 다루는 것보다 깔끔하고 명시적인 방법으로 null 가능성이 있는 객체를 처리할 수 있습니다. 이를 통해 NullPointerException을 방지하고, 코드의 가독성과 안정성을 높일 수 있습니다. null로 인한 예외가 발생하지 않도록 도와주고, Optional 클래스의 메소드를 통해 null을 컨트롤 할 수 있습니다.

```
public class OptionalExample {
    public static void main(String[] args) {
        String str = "Hello, Optional!";

        // Optional 객체 생성
        Optional<String> optionalStr = Optional.of(str);

        // 값이 존재하는 경우에만 출력
        optionalStr.ifPresent(System.out::println);

        // 값이 없는 경우 기본 문자열을 반환하여 출력
        String nullSafeString = optionalStr.orElse("Default String");
        System.out.println(nullSafeString);

        // null을 허용하는 Optional 객체 생성
        String nullString = null;
        Optional<String> nullOptional = Optional.ofNullable(nullString);

        // null인 경우 대체 문자열 출력
        String result = nullOptional.orElse("null 대체 문자열");
        System.out.println(result);
    }
}
```

Optional은 자바에서 null 처리를 보다 세련되게 할 수 있게 해주는 유틸리티로, 함수형 프로그래밍 스타일을

적극적으로 활용할 수 있게 도와줍니다. 하지만, 모든 상황에서 Optional을 사용하는 것이 아니라, null을 명시적으로 처리해야 하는 상황에서 사용하는 것이 적절합니다.

□ 컬렉션에 대해 설명해주세요
자바의 컬렉션(Collection)은 데이터의 집합이나 그룹을 효율적으로 저장하고 처리하기 위한 프레임워크입니다. 컬렉션 프레임워크는 다양한 인터페이스와 이를 구현하는 클래스로 구성되어 있으며, 데이터를 다루기 위한 일관된 방법을 제공합니다. 주요 컬렉션 인터페이스에는 List, Set, Queue, Map 등이 있습니다.

주요 컬렉션 인터페이스
List : 순서가 있는 데이터의 집합을 다루며, 중복된 요소를 허용합니다. ArrayList, LinkedList, Vector 등의 클래스가 List 인터페이스를 구현합니다.
예: List<String> list = new ArrayList<>();

Set : 중복된 요소를 허용하지 않는 데이터의 집합입니다. 순서를 보장하지 않는 HashSet과 순서를 보장하는 LinkedHashSet, 정렬된 순서를 유지하는 TreeSet 등이 있습니다.
예: Set<String> set = new HashSet<>();

Queue : FIFO(First In First Out) 방식이나 우선순위에 따라 요소를 처리하는 데이터 구조입니다.
LinkedList, PriorityQueue 등이 있습니다.
예: Queue<String> queue = new LinkedList<>();

Map : 키와 값의 쌍으로 이루어진 데이터 집합을 다룹니다. 키는 중복될 수 없으며 각 키는 하나의 값과 매핑됩니다. HashMap, TreeMap, LinkedHashMap 등의 구현체가 있습니다.
예: Map<String, Integer> map = new HashMap<>();

컬렉션 프레임워크의 특징
- 일관된 구조: 컬렉션 프레임워크는 다양한 컬렉션 타입을 일관된 인터페이스로 제공하여 사용의 일관성을 보장합니다.
- 알고리즘 재사용성: 정렬, 검색 등의 공통 알고리즘을 컬렉션 타입에 상관없이 재사용할 수 있습니다.
- 데이터 구조와 알고리즘의 분리: 데이터 구조의 구현과 이를 조작하는 알고리즘을 분리하여 각각 독립적으로 확장하고 개선할 수 있습니다.

컬렉션 프레임워크는 데이터를 효율적으로 관리하고, 처리하는 데 필수적인 요소입니다. 자바에서 제공하는 다양한 컬렉션 클래스와 인터페이스를 사용하여 애플리케이션의 필요에 맞게 데이터를 저장하고 처리할 수 있습니다.

□ 제네릭에 대해 간략히 개념정리하고, 왜 사용해야 하는지 설명해보세요.
제네릭(Generic)은 자바에서 타입(type) 안정성을 제공하고, 캐스팅(casting) 문제를 줄이기 위해 도입된 기능입니다. 제네릭을 사용하면 클래스, 인터페이스, 메소드를 정의할 때 타입(type)을 파라미터로 받을 수 있습니다. 이를 통해 다양한 타입의 객체를 다룰 수 있는 범용적인 프로그래밍이 가능해집니다.

제네릭의 주요 장점
1) 타입 안정성(Type Safety) : 제네릭을 사용하면 컴파일 시점에 타입 체크를 수행할 수 있습니다. 이는 런타임에 발생할 수 있는 ClassCastException을 예방할 수 있습니다.
예를 들어, List<String>은 문자열만 포함할 수 있으며, 다른 타입의 객체를 추가하려고 하면 컴파일 오류가 발생합니다.
2) 코드 중복 감소 : 한 번의 제네릭 클래스 또는 메소드 정의로 다양한 타입의 객체를 처리할 수 있습니다. 이는 코드 중복을 줄이고 유지보수를 용이하게 합니다.
3) 캐스팅 제거 : 제네릭을 사용하지 않을 경우, 컬렉션에서 객체를 추출할 때마다 캐스팅이 필요합니다. 하지만 제네릭을 사용하면 캐스팅이 필요 없어 코드가 간결해지고, 오류 가능성이 줄어듭니다.

사용 예제 : 제네릭이 없는 경우와 있는 경우를 비교하여 살펴보겠습니다.
1) 제네릭을 사용하지 않는 경우:

```
List list = new ArrayList();
list.add("hello");
String str = (String) list.get(0); // 캐스팅 필요
```

2) 제네릭을 사용하는 경우:

```
List<String> list = new ArrayList<>();
list.add("hello");
String str = list.get(0); // 캐스팅 불필요
```

제네릭을 사용하면, 컴파일 시점에 타입을 체크할 수 있어 프로그램의 안정성을 높이고, 타입 캐스팅의 번거로움을 줄일 수 있습니다. 이러한 이유로 제네릭은 자바 프로그래밍에서 널리 사용되며, 강력한 타입 체크를 통해 보다 안정적인 코드를 작성할 수 있게 도와줍니다.

□ 직렬화(Serialize)에 대해 설명해주세요.
시스템 내부에서 사용되는 객체 또는 데이터를 외부의 시스템에서도 사용할 수 있도록 바이트(byte) 형태로 데이터 변환하는 기술이며, 반대로 직렬화된 바이트 형태의 데이터를 다시 객체로 변환하는 과정을 '역직렬화'라고 합니다.

좀 더 구체적으로 설명해 보겠습니다.
직렬화(Serialization)는 객체의 상태를 포함한 데이터를 연속적인 바이트 스트림으로 변환하는 과정을 말합니다. 이 변환된 데이터는 파일에 저장하거나 네트워크를 통해 다른 시스템으로 전송할 수 있습니다. 반대 과정, 즉 바이트 스트림을 다시 객체로 변환하는 것을 역직렬화(Deserialization)라고 합니다.

직렬화의 목적
- 데이터의 영속성(Persistence): 객체의 상태를 영구적으로 저장하기 위해 파일 시스템 등에 저장할 수 있습니다.
- 객체의 전송(Networking): 객체를 네트워크를 통해 다른 시스템으로 전송할 수 있습니다. 이를 통해 분산 시스템에서 객체를 교환하고 함수 호출을 원격으로 실행할 수 있습니다.

자바에서의 직렬화 : 자바에서는 Serializable 인터페이스를 구현함으로써 객체를 직렬화할 수 있습니다. Serializable 인터페이스는 메소드를 포함하지 않는 마커 인터페이스(marker interface)로, 직렬화를 지원한다는 것을 JVM에 알려줍니다. 객체가 Serializable 인터페이스를 구현하면, 자바의 직렬화 메커니즘을 사용하여 객체를 바이트 스트림으로 변환할 수 있습니다.

직렬화 과정
1) 직렬화 가능한 객체 정의: 객체는 Serializable 인터페이스를 구현해야 합니다.
2) ObjectOutputStream 사용: ObjectOutputStream 클래스를 사용하여 객체를 바이트 스트림으로 변환합니다.
3) 데이터 저장 또는 전송: 이 바이트 스트림을 파일에 저장하거나 네트워크를 통해 전송할 수 있습니다.

역직렬화 과정
1) 바이트 스트림 읽기: 저장된 데이터 또는 네트워크를 통해 전송된 바이트 스트림을 읽습니다.
2) ObjectInputStream 사용: ObjectInputStream 클래스를 사용하여 바이트 스트림을 원래의 객체로 역직렬화합니다.

주의사항
직렬화 과정에서 transient 키워드를 사용한 필드는 직렬화에서 제외됩니다. 직렬화된 객체의 클래스가 수정되면, 역직렬화 과정에서 문제가 발생할 수 있습니다. 이를 위해 serialVersionUID라는 고유 식별자를 사용하여 클래스의 버전을 관리할 수 있습니다.
직렬화는 객체의 상태를 저장하고, 분산 처리나 객체의 지속성 관리에 중요한 역할을 하는 기술입니다.

□ SerialVersionUID를 선언해야 하는 이유에 대해 설명해주세요.
JVM은 직렬화와 역직렬화를 하는 시점의 클래스에 대한 버전 번호를 부여하는데, 만약 그 시점에 클래스의 정의가 바뀌어 있다면 새로운 버전 번호를 할당하게 됩니다. 그래서 직렬화할 때의 버전 번호와 역직렬화를 할 때의 버전 번호가 다르면 역직렬화가 불가능하게 될 수 있기 때문에 이런 문제를 해결하기 위해 SerialVersionUID를 사용합니다. 만약 직렬화할 때 사용한 SerialVersionUID의 값과 역직렬화 하기 위해 사용했던 SVUID가 다르다면 InvalidClassException이 발생할 수 있습니다.

부연 설명을 하자면
serialVersionUID는 직렬화된 객체를 역직렬화할 때 클래스의 버전을 확인하는 데 사용되는 고유 식별자입니다. 자바 직렬화 메커니즘은 직렬화와 역직렬화 과정에서 객체의 클래스 버전을 일치시키기 위해 serialVersionUID를 사용합니다.

serialVersionUID 선언의 중요성
1) 버전 호환성 유지: 객체가 직렬화된 후, 해당 객체의 클래스 정의가 변경되었을 경우 (필드 추가/삭제, 클래스 구조 변경 등), 역직렬화 과정에서 InvalidClassException이 발생할 수 있습니다. serialVersionUID를 사용하면 클래스의 버전이 명시적으로 관리되어 이러한 문제를 예방할 수 있습니다.
2) 직렬화된 객체의 일관성 보장: 역직렬화 시, JVM은 클래스의 serialVersionUID와 직렬화된 객체의 serialVersionUID를 비교하여 일치하는지 확인합니다. 이 값이 일치하면 역직렬화를 수행하고, 일치하지 않으면 예외를 발생시킵니다. 이를 통해 다른 버전의 클래스로 인해 발생할 수 있는 문제를 방지합니다.

3) 명시적인 버전 관리: 개발자는 serialVersionUID를 통해 클래스의 버전을 명시적으로 관리할 수 있습니다. 이는 특히 라이브러리나 프레임워크 등을 개발할 때 중요하며, 호환성을 유지하기 위한 목적으로 사용됩니다.

serialVersionUID 사용 예 ----------

```
import java.io.Serializable;

public class MyClass implements Serializable {
    private static final long serialVersionUID = 1L; // 명시적으로 serialVersionUID 선언

    // 클래스 내용
}
```

사용할 이유

- 호환성 유지: 이전에 직렬화된 객체와의 호환성을 유지하기 위해 필요합니다. 특히 분산 시스템에서는 클라이언트와 서버 간에 객체의 호환성을 보장하기 위해 중요합니다.
- 안정적인 직렬화 프로세스: serialVersionUID를 명시적으로 선언함으로써 직렬화와 역직렬화 과정에서의 예상치 못한 오류를 방지하고, 안정적인 객체 교환을 보장합니다.

serialVersionUID는 직렬화된 객체와 클래스의 호환성을 보장하고, 직렬화 프로세스의 안정성을 높이기 위해 중요한 요소입니다. 따라서 직렬화를 사용하는 클래스에서는 serialVersionUID를 명시적으로 선언하는 것이 좋습니다.