**JAVA 콘솔 입출력: System.out**

**System.out**은 자바에서 콘솔에 데이터를 출력하는 데 사용되는 표준 출력 스트림입니다. 콘솔은 텍스트 기반 입출력을 위한 인터페이스이며, 일반적으로 명령 프롬프트나 터미널이라고도 불립니다. System.out을 사용하면 다양한 형식의 데이터를 콘솔에 출력할 수 있습니다.

**주요 기능:**

* **문자열 출력:** System.out.println("안녕하세요!") 와 같이 문자열을 직접 출력할 수 있습니다.
* **변수 출력:** int num = 10; System.out.println(num); 와 같이 변수의 값을 출력할 수 있습니다.
* **형식화된 출력:** System.out.printf("이름: %s, 나이: %d", "홍길동", 30); 와 같이 printf 메서드를 사용하여 더욱 정교하게 형식화된 출력을 생성할 수 있습니다.
* **오류 메시지 출력:** System.err.println("오류 발생!"); 와 같이 System.err 객체를 사용하여 오류 메시지를 출력할 수 있습니다.

**사용 예시:**

Java

public class ConsoleIO {

public static void main(String[] args) {

System.out.println("콘솔 입출력 예제입니다."); // 문자열 출력

int num = 100;

System.out.println("변수 num의 값은 " + num + "입니다."); // 변수 출력

String name = "홍길동";

int age = 30;

System.out.printf("이름: %s, 나이: %d\n", name, age); // 형식화된 출력

System.err.println("경고 메시지입니다."); // 오류 메시지 출력

}

}

### **추가 정보:**

* **System.in:** 콘솔 입력을 위한 표준 입력 스트림입니다.
* **Scanner 클래스:** 콘솔 입력을 처리하는 데 사용되는 클래스입니다.
* **파일 입출력:** 파일을 읽고 쓰는 데 사용되는 입출력 클래스입니다.

## **자바 콘솔 입출력: Scanner**

자바에서 콘솔 입출력을 수행하는 데 가장 일반적으로 사용되는 클래스는 **Scanner**입니다. Scanner 클래스는 java.util 패키지에 속하며, 사용자로부터 입력 데이터를 간편하게 읽을 수 있도록 다양한 메서드를 제공합니다.

**1. Scanner 생성**

Scanner 객체를 생성하려면 다음과 같이 new 키워드를 사용합니다.

Java

Scanner sc = new Scanner(System.in);

System.in은 표준 입력 스트림을 의미하며, 키보드로 입력되는 데이터를 나타냅니다.

**2. 기본 입력 메서드**

Scanner 클래스는 다양한 데이터 유형을 입력받는 메서드를 제공합니다. 대표적인 메서드와 사용법은 다음과 같습니다.

* **next()**: 문자열 입력 (공백 기준으로 구분)
  + 예시: String name = sc.next();
* **nextInt()**: 정수 입력
  + 예시: int age = sc.nextInt();
* **nextDouble()**: 실수 입력
  + 예시: double score = sc.nextDouble();
* **nextBoolean()**: 불리언 입력 (true 또는 false)
  + 예시: boolean isAdult = sc.nextBoolean();

**3. 입력 문자열 처리**

Scanner 클래스는 입력 문자열을 다양한 방식으로 처리하는 메서드도 제공합니다.

* **nextLine()**: 한 줄의 문자열 입력 (엔터 키 포함)
  + 예시: String address = sc.nextLine();
* **hasNext()**: 다음 입력값이 있는지 확인
  + 예시: if (sc.hasNext()) { ... }
* **hasNextXxx()**: 다음 입력값의 데이터 타입 확인 (예: hasNextInt(), hasNextLine())
  + 예시: if (sc.hasNextInt()) { int num = sc.nextInt(); }

**4. 예제 코드**

다음은 Scanner를 사용하여 사용자로부터 이름, 나이, 성적을 입력받고 평균 성적을 계산하는 예제 코드입니다.

Java

import java.util.Scanner;

public class ScannerExample {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("이름: ");

String name = sc.next();

System.out.print("나이: ");

int age = sc.nextInt();

System.out.print("성적: ");

double score1 = sc.nextDouble();

double score2 = sc.nextDouble();

double average = (score1 + score2) / 2;

System.out.println(name + "님의 평균 성적은 " + average + "입니다.");

}

}

**JAVA 파일 입출력 객체**

자바에서 파일 입출력을 수행하기 위해서는 **객체**를 사용합니다. 객체는 파일의 정보와 기능을 나타내는 데이터 구조입니다.

**주요 객체 종류:**

* **File:** 파일 또는 디렉터리에 대한 정보를 제공합니다. 파일 경로, 이름, 크기, 마지막 수정 시간 등을 가져올 수 있습니다.
* **InputStream:** 파일로부터 데이터를 읽는 데 사용됩니다. 바이트 단위 또는 문자 단위로 읽을 수 있습니다.
* **OutputStream:** 파일에 데이터를 쓰는 데 사용됩니다. 바이트 단위 또는 문자 단위로 쓸 수 있습니다.
* **Reader:** 문자 단위로 파일을 읽는 데 사용됩니다.
* **Writer:** 문자 단위로 파일에 쓰는 데 사용됩니다.

**예제:**

Java

// 파일 객체 생성

File file = new File("test.txt");

// 파일 출력 스트림 생성

FileOutputStream fos = new FileOutputStream(file);

// 파일에 문자열 쓰기

fos.write("Hello, World!".getBytes());

// 스트림 닫기

fos.close();

// 파일 입력 스트림 생성

FileInputStream fis = new FileInputStream(file);

// 파일로부터 문자열 읽기

byte[] b = new byte[10];

int len = fis.read(b);

String str = new String(b, 0, len);

// 스트림 닫기

fis.close();

System.out.println(str);

**추가 정보**

* **객체 직렬화:** 객체를 파일에 저장하고 다시 로드하는 기능입니다. 객체 직렬화를 사용하면 메모리에 있는 객체를 영구적으로 저장하거나 다른 프로그램과 공유할 수 있습니다.
* **NIO (New Input/Output):** 자바 7에서 도입된 새로운 입출력 API입니다. NIO는 기존 입출력 API보다 더 빠르고 효율적이며 비동기 I/O를 지원합니다.

## 

## **JAVA 파일 입출력: FileInputStream과 FileOutputStream**

**개요**

자바에서 파일 입출력을 수행하기 위해서는 **InputStream**과 **OutputStream** 클래스를 사용합니다.

* **InputStream**: 파일로부터 데이터를 읽어들이는 데 사용됩니다.
* **OutputStream**: 파일에 데이터를 쓰는 데 사용됩니다.

**FileInputStream**과 **FileOutputStream**은 **InputStream**과 **OutputStream**의 자손 클래스이며, **바이트 단위**로 파일 입출력을 수행하도록 특화되어 있습니다.

**FileInputStream**

* **생성자**:
  + FileInputStream(File file): 지정된 파일을 대상으로 하는 **FileInputStream** 객체를 생성합니다.
  + FileInputStream(String fileName): 지정된 파일 이름을 가진 파일을 대상으로 하는 **FileInputStream** 객체를 생성합니다.
* **메서드**:
  + read(): 파일에서 한 바이트의 데이터를 읽어 반환합니다.
  + read(byte[] b): 파일에서 최대 b.length 바이트의 데이터를 읽어 b 배열에 저장하고 읽은 바이트 수를 반환합니다.
  + available(): 파일에서 읽을 수 있는 남은 바이트 수를 반환합니다.
  + close(): **FileInputStream** 객체를 닫습니다.

**FileOutputStream**

* **생성자**:
  + FileOutputStream(File file): 지정된 파일을 대상으로 하는 **FileOutputStream** 객체를 생성합니다.
  + FileOutputStream(String fileName): 지정된 파일 이름을 가진 파일을 대상으로 하는 **FileOutputStream** 객체를 생성합니다.
  + FileOutputStream(File file, boolean append): 지정된 파일을 대상으로 하는 **FileOutputStream** 객체를 생성하며, append가 true인 경우 기존 파일에 내용을 추가하고 false인 경우 기존 파일을 덮어씁니다.
* **메서드**:
  + write(int b): 파일에 한 바이트의 데이터 b를 씁니다.
  + write(byte[] b): b 배열에 저장된 데이터를 파일에 씁니다.
  + write(byte[] b, int off, int len): b 배열의 off번째부터 len 바이트만큼의 데이터를 파일에 씁니다.
  + close(): **FileOutputStream** 객체를 닫습니다.

**예제**

Java

import java.io.FileInputStream;

import java.io.FileOutputStream;

public class FileIOExample {

public static void main(String[] args) throws Exception {

// 파일 읽기

FileInputStream fis = new FileInputStream("input.txt");

int b;

while ((b = fis.read()) != -1) {

System.out.print((char) b);

}

fis.close();

System.out.println();

// 파일 쓰기

FileOutputStream fos = new FileOutputStream("output.txt");

String data = "Hello, World!";

fos.write(data.getBytes());

fos.close();

}

}

**JAVA 조건문 구조 및 문법: if~else**

**1. 조건문이란?**

조건문은 프로그램 실행 과정에서 **특정 조건에 따라 코드 실행을 제어**하는 문법입니다. 조건에 따라 다른 코드 블록을 실행하거나, 다른 값을 반환하는 데 사용됩니다.

**2. if~else 문**

**2.1 기본 구조**

if~else 문은 가장 기본적인 조건문 구조이며, 다음과 같은 형식으로 구성됩니다.

Java

if (조건식) {

// 조건이 참일 때 실행할 코드 블록

} else {

// 조건이 거짓일 때 실행할 코드 블록

}

* **조건식:** 참 또는 거짓을 판단하는 논리 표현식입니다.
* **코드 블록:** 중괄호 {} 안에 작성되는 코드들을 의미합니다.

**2.2 예시**

Java

int age = 20;

if (age >= 18) {

System.out.println("성인입니다.");

} else {

System.out.println("미성년자입니다.");

}

**2.3 여러 개의 조건 판단: if~else if**

if~else if 문은 여러 개의 조건을 차례대로 판단하여 코드 블록을 실행하는 구조입니다.

Java

int score = 85;

if (score >= 90) {

System.out.println("A등급입니다.");

} else if (score >= 80) {

System.out.println("B등급입니다.");

} else if (score >= 70) {

System.out.println("C등급입니다.");

} else {

System.out.println("D등급입니다.");

}

**3. 주의 사항**

* **조건식은 반드시 true 또는 false를 반환하는 논리 표현식이어야 합니다.**
* **else 블록은 생략할 수 있지만, 코드 가독성을 위해 명시하는 것이 좋습니다.**
* **여러 개의 else if를 사용할 때는 조건 순서를 신중하게 고려해야 합니다.**

## **JAVA Switch-Case 조건문 구조 및 문법**

**1. 개요**

**Switch-Case 문은 조건에 따라 여러 가지 분기 처리를 수행하는 선택문입니다. 다중 if 문보다 코드가 간결하고 가독성이 좋아 여러 조건을 처리해야 하는 경우 유용하게 사용됩니다.**

**2. 구조**

**Java**

**switch (식) {**

**case 값1:**

**실행문1;**

**break;**

**case 값2:**

**실행문2;**

**break;**

**...**

**default:**

**실행문N;**

**}**

* **switch: Switch-Case 문을 시작하는 키워드입니다.**
* **식: 조건으로 사용할 식을 작성합니다. 식의 결과 값에 따라 분기 처리가 이루어집니다.**
* **case: 특정 값을 나타내는 키워드입니다.**
* **값: Case 블록에서 처리할 값을 작성합니다.**
* **실행문: Case 블록에서 수행할 코드를 작성합니다.**
* **break: Case 블록을 종료하고 switch 문에서 벗어나는 키워드입니다.**
* **default: 모든 Case 조건에 해당하지 않는 경우 수행할 코드를 작성합니다.**

**3. 예제**

**Java**

**int day = 3;**

**switch (day) {**

**case 1:**

**System.out.println("월요일입니다.");**

**break;**

**case 2:**

**System.out.println("화요일입니다.");**

**break;**

**case 3:**

**System.out.println("수요일입니다.");**

**break;**

**default:**

**System.out.println("목요일, 금요일, 토요일, 일요일 중 하나입니다.");**

**}**

**4. 주의점**

* **Switch 식은 정수, 문자열, 열거형 타입 등 다양한 타입을 사용할 수 있습니다.**
* **Case 값은 서로 중복될 수 없습니다.**
* **Break 키워드를 사용하지 않으면 다음 Case 블록의 코드도 함께 실행됩니다.**
* **Default 블록은 반드시 작성할 필요는 없지만, 모든 Case 조건에 해당하지 않는 경우를 처리하기 위해 사용하는 것이 좋습니다.**

**5. 향상된 Switch 문 (Java 14 이상)**

**Java 14부터는 다음과 같은 향상된 Switch 문 기능을 사용할 수 있습니다.**

* **여러 Case 값을 쉼표(,)로 구분하여 간결하게 작성할 수 있습니다.**
* **식으로 표현하여 반환값을 받을 수 있습니다.**

**Java**

**int day = 3;**

**String result = switch (day) {**

**case 1, 7 -> "주말입니다.";**

**case 2, 3, 4, 5 -> "평일입니다.";**

**default -> "알 수 없는 요일입니다.";**

**};**

**System.out.println(result);**

**코드를 사용할 때는 주의가 필요합니다.**

**content\_copy**

**6. Switch 문 vs. If 문**

* **Switch 문은 여러 조건을 간결하고 효율적으로 처리하는 데 유용합니다.**
* **If 문은 더욱 복잡한 조건 처리나 순차적인 조건 판단에 적합합니다.**

## **JAVA 반복문 구조 및 문법: for 문**

**1. 개요**

**for 문은 반복 횟수를 미리 알고 있는 경우에 사용하는 반복문입니다. 초기화, 조건식, 증감식으로 구성됩니다.**

**2. 구조**

**Java**

**for (초기화; 조건식; 증감식) {**

**실행문;**

**}**

* **초기화: 반복 시작 전에 수행되는 문장입니다. 일반적으로 반복 제어 변수를 초기화하는 데 사용됩니다.**
* **조건식: 반복 여부를 결정하는 조건입니다. 조건식이 true인 경우 반복문을 계속 수행하고, false인 경우 반복문을 종료합니다.**
* **증감식: 반복 제어 변수를 증가시키거나 감소시키는 문장입니다. 반복할 때마다 수행됩니다.**

**3. 예시**

**Java**

**// 1부터 10까지 출력**

**for (int i = 1; i <= 10; i++) {**

**System.out.println(i);**

**}**

**4. 향상된 for 문**

**향상된 for 문은 배열이나 컬렉션의 요소를 순환하는 데 사용되는 간편한 for 문 형식입니다.**

**Java**

**// 배열 요소 출력**

**int[] numbers = {1, 2, 3, 4, 5};**

**for (int number : numbers) {**

**System.out.println(number);**

**}**

**5. 주의사항**

* **조건식이 항상 true인 경우 무한 루프에 빠질 수 있습니다.**
* **증감식이 제대로 설정되지 않으면 원하는 반복 횟수대로 반복되지 않을 수 있습니다.**

## **JAVA while 반복문 구조 및 문법**

**1. 기본 구조**

**while 반복문은 조건식이 참인 동안 코드 블록을 반복적으로 실행하는 구조입니다.**

**Java**

**while (조건식) {**

**코드 블록;**

**}**

**구성 요소:**

* **조건식: true 또는 false를 반환하는 논리 표현식입니다.**
* **코드 블록: 중괄호 {}로 묶인 코드 영역입니다. 조건식이 true인 동안 반복적으로 실행됩니다.**

**2. 실행 순서**

1. **조건식을 평가합니다.**
2. **조건식이 true이면 코드 블록을 실행합니다.**
3. **코드 블록 실행이 완료되면 다시 1번으로 돌아갑니다.**
4. **조건식이 false이면 반복문을 종료합니다.**

**3. 예시**

**Java**

**int count = 1;**

**while (count <= 10) {**

**System.out.println(count);**

**count++;**

**}**

**설명:**

* **count 변수는 1부터 시작합니다.**
* **while 조건식은 count가 10보다 작거나 같을 때까지 true를 반환합니다.**
* **System.out.println(count)는 현재 count 값을 출력합니다.**
* **count++는 count 값을 1씩 증가시킵니다.**
* **count가 11이 되면 조건식이 false가 되어 반복문이 종료됩니다.**

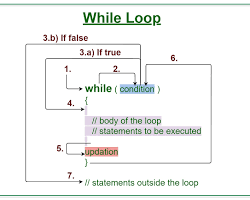
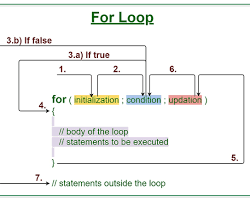
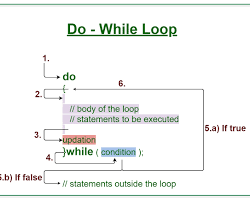
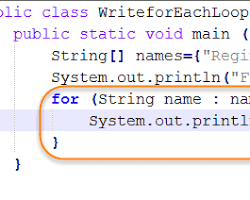
**4. 주의 사항**

* **무한 루프에 빠지지 않도록 조건식이 반드시 false가 되는 조건을 포함해야 합니다.**
* **코드 블록 내에서 count 변수를 변경하는 경우, 조건식에도 반영되어야 합니다.**

## **JAVA 반복문 구조 및 문법 (break)**

### **1. 반복문 종류**

**자바에는 다양한 반복문 구조가 존재하며, 각각 특정한 상황에 맞춰 사용됩니다. 대표적인 반복문으로는 다음과 같은 것들이 있습니다.**

* **while문: 조건식이 참인 동안 반복을 수행합니다. 조건식이 거짓이 되면 반복이 종료됩니다.  
  **[**새 창에서 열기www.geeksforgeeks.org**](https://www.geeksforgeeks.org/java-while-loop-with-examples/) **while loop java**
* **for문: 초기값, 종료 조건, 증감식을 사용하여 반복을 수행합니다. 초기값부터 증감식을 통해 계산된 값이 종료 조건을 만족할 때까지 반복됩니다.  
  **[**새 창에서 열기www.geeksforgeeks.org**](https://www.geeksforgeeks.org/java-for-loop-with-examples/) **for loop java**
* **do-while문: 먼저 반복 블록을 실행하고, 그 후 조건식을 검사합니다. 조건식이 참이면 반복 블록을 다시 실행하고, 거짓이면 반복을 종료합니다.  
  **[**새 창에서 열기www.geeksforgeeks.org**](https://www.geeksforgeeks.org/java-do-while-loop-with-examples/) **do while loop java**
* **for-each문: 배열이나 컬렉션의 요소들을 차례대로 반복합니다.  
  **[**새 창에서 열기medium.com**](https://medium.com/tech-learn-share/for-each-loop-java-b97464691b41) **for each loop java**

### **2. break 문법**

**break 키워드는 반복문을 강제로 종료하는 데 사용됩니다. 즉, 반복 조건을 검사하지 않고 바로 반복 블록을 벗어납니다.**

**문법:**

**Java**

**break;**

**예제:**

**Java**

**for (int i = 0; i < 10; i++) {**

**if (i == 5) {**

**break;**

**}**

**System.out.println(i);**

**}**

**위 예제에서는 i가 5가 되면 break 키워드를 만나 반복문을 종료합니다. 따라서 0부터 4까지만 출력됩니다.**

**주의:**

* **break 키워드는 가장 가까운 반복문만 종료합니다. 즉, 이중 또는 다중 반복문에서 사용하는 경우, 원하는 반복문을 정확하게 지정해야 합니다.**
* **break 키워드와 함께 레이블을 사용하여 특정 반복문을 지정할 수 있습니다.**

**Java**

**outer: for (int i = 0; i < 10; i++) {**

**for (int j = 0; j < 10; j++) {**

**if (j == 5) {**

**break outer; // 'outer' 레이블을 사용하여 외부 반복문 종료**

**}**

**System.out.println(i + " " + j);**

**}**

**}**

**위 예제에서는 j가 5가 되면 outer 레이블을 사용하여 외부 반복문을 종료합니다.**

### **3. break 활용 사례**

* **특정 조건을 만족하면 반복문을 중단하고 싶을 때**
* **예외 상황 발생 시 반복문을 종료하고 싶을 때**
* **특정 반복 횟수만 수행하고 싶을 때**
* **이중 또는 다중 반복문에서 특정 반복문만 제어하고 싶을 때**

### **4. 추가 정보**

* **continue 키워드는 현재 반복 횟수를 건너뛰고 다음 반복으로 넘어가는 데 사용됩니다.**
* **레이블은 반복문, 조건문, switch 문 등을 명확하게 구분하고 제어하는 데 사용됩니다.**

## **JAVA 반복문 구조 및 continue 문법**

**JAVA에서 반복문은 코드 블록을 반복적으로 실행하는 데 사용되는 제어 흐름 문입니다. 반복 횟수를 제어하는 데 사용되는 조건을 기반으로 다양한 유형의 반복문이 있습니다.**

**1. 반복문 종류**

* **for 문: 특정 범위 내의 값을 반복적으로 처리하는 데 사용됩니다.**

**Java**

**for (int i = 0; i < 10; i++) {**

**System.out.println(i);**

**}**

* **while 문: 조건이 참인 동안 코드 블록을 반복적으로 실행합니다.**

**Java**

**int i = 0;**

**while (i < 10) {**

**System.out.println(i);**

**i++;**

**}**

* **do-while 문: 먼저 코드 블록을 실행한 다음 조건을 평가하여 반복 여부를 결정합니다.**

**Java**

**int i = 0;**

**do {**

**System.out.println(i);**

**i++;**

**} while (i < 10);**

* **for-each 문: 배열 또는 컬렉션의 모든 요소를 반복적으로 처리하는 데 사용됩니다.**

**Java**

**int[] numbers = {1, 2, 3, 4, 5};**

**for (int number : numbers) {**

**System.out.println(number);**

**}**

**2. continue 문법**

**continue 문은 현재 반복 루프의 나머지 코드를 건너뛰고 다음 루프 반복으로 넘어가는 데 사용됩니다.**

**예시:**

**Java**

**for (int i = 0; i < 10; i++) {**

**if (i % 2 == 0) {**

**continue;**

**}**

**System.out.println(i);**

**}**

**위 코드는 0부터 9까지의 숫자를 반복적으로 출력하지만, 짝수는 건너뛴 결과 홀수만 출력됩니다.**

**주의 사항:**

* **continue 문은 반복문 내에서만 사용할 수 있습니다.**
* **continue 문을 사용하면 루프 반복의 순서가 변경될 수 있으므로 코드 작성 시 주의해야 합니다.**

**3. continue 문 활용 예시**

* **특정 조건에 맞는 값만 처리하고 싶을 때**
* **반복 루프 내에서 오류가 발생했을 때 루프를 중단하지 않고 다음 루프로 넘어갈 때**
* **코드를 간결하게 만들 때**

## **JAVA 파일에서 클래스 생성 방법**

**자바에서 클래스를 생성하는 방법은 크게 두 가지가 있습니다.**

**1. IDE 사용**

* **Eclipse**
  1. **프로젝트 생성:**
     + **'New' 메뉴 > 'Java Project' 를 선택합니다.**
     + **프로젝트 이름을 입력하고 'Finish' 를 클릭합니다.**
  2. **클래스 생성:**
     + **프로젝트 폴더를 마우스 오른쪽 버튼으로 클릭하고 'New' > 'Class' 를 선택합니다.**
     + **'Source folder' 에 소스 코드를 저장할 폴더를 선택합니다.**
     + **'Name' 에 클래스 이름을 입력하고 'Finish' 를 클릭합니다.**
* **IntelliJ IDEA**
  1. **프로젝트 생성:**
     + **'File' 메뉴 > 'New' > 'Project' 를 선택합니다.**
     + **'Java' 탭을 선택하고 'Next' 를 클릭합니다.**
     + **프로젝트 이름을 입력하고 'Project location' 을 지정합니다.**
     + **'Module' > 'Finish' 를 클릭합니다.**
  2. **클래스 생성:**
     + **'Ctrl' + 'N' 을 누릅니다.**
     + **'Class' 를 선택하고 'Enter' 를 누릅니다.**
     + **'Package' 에 패키지 이름을 입력합니다.**
     + **'Class name' 에 클래스 이름을 입력하고 'OK' 를 클릭합니다.**

**2. 텍스트 편집기 사용**

* **메모장이나 Notepad++ 처럼 간단한 텍스트 편집기를 사용하여 직접 .java 파일을 작성할 수 있습니다.**
* **다음은 간단한 예제입니다.**

**Java**

**package com.example;**

**public class MyClass {**

**public static void main(String[] args) {**

**System.out.println("Hello, World!");**

**}**

**}**

* **위 코드를 MyClass.java라는 이름으로 저장하고, javac 명령을 사용하여 컴파일합니다.**

**javac MyClass.java**

* **컴파일이 성공하면 MyClass.class라는 파일이 생성됩니다.**

**참고:**

* **클래스 이름은 소문자로 시작해야 합니다.**
* **클래스 이름은 파일 이름과 동일해야 합니다.**
* **패키지가 있는 경우, 클래스 선언 앞에 package 키워드를 사용하여 패키지 이름을 지정해야 합니다.**

**주의:**

* **위에 제시된 방법 외에도 다양한 방법으로 클래스를 생성할 수 있습니다.**
* **사용하는 IDE나 텍스트 편집기에 따라 특정 단계가 다를 수 있습니다.**

## **JAVA 클래스에서 필드를 생성하는 역할**

**JAVA 클래스에서 필드는 객체의 상태를 저장하는 데 사용되는 데이터 요소입니다. 필드는 클래스의 멤버 변수라고도 불리며, 다음과 같은 역할을 합니다.**

**1. 객체의 속성 표현:**

**필드는 객체의 고유한 특성을 나타내는 데이터를 저장합니다. 예를 들어, Person 클래스의 경우 이름, 나이, 키 등을 필드로 정의할 수 있습니다. 이러한 필드 값들은 각 Person 객체의 개별적인 속성을 나타냅니다.**

**2. 데이터 저장 및 관리:**

**필드는 객체의 상태를 나타내는 데이터를 저장하는 공간을 제공합니다. 프로그램 실행 중에 객체의 상태는 필드 값의 변화를 통해 반영됩니다. 또한, 여러 필드를 조합하여 객체의 복잡한 상태를 표현할 수 있습니다.**

**3. 정보 은닉 및 접근 제어:**

**필드의 접근 지정자를 사용하여 정보 은닉과 접근 제어를 구현할 수 있습니다. 예를 들어, private 접근 지정자를 사용하면 해당 필드는 클래스 내부에서만 접근 가능하게 제한하고, public 접근 지정자를 사용하면 외부에서도 자유롭게 접근하도록 허용할 수 있습니다.**

**4. 객체의 동작 영향:**

**필드 값은 객체의 메서드 동작에 영향을 미칠 수 있습니다. 메서드는 필드 값을 읽고 변경하거나, 다른 필드 값을 계산하여 반환하는 등의 작업을 수행할 수 있습니다.**

**5. 메모리 관리:**

**필드는 객체에 할당된 메모리 공간에 저장됩니다. 객체가 생성되면 필드에 초기값이 할당되고, 객체가 소멸되면 필드에 저장된 데이터는 메모리에서 해제됩니다.**

**필드 선언 방법:**

**필드는 클래스 선언 내부에 다음과 같은 형식으로 선언됩니다.**

**Java**

**접근지정자 데이터형 변수명;**

**예를 들어, 다음은 Person 클래스에서 이름, 나이, 키를 필드로 선언하는 코드입니다.**

**Java**

**public class Person {**

**private String name;**

**private int age;**

**private double height;**

**// ... (생성자, 메서드 등)**

**}**

**필드 접근 및 활용:**

**필드는 객체 참조를 통해 접근하고 활용할 수 있습니다. 객체 참조를 사용하여 필드 값을 읽거나 변경하고, 필드 값을 사용하여 메서드를 호출할 수 있습니다.**

**Java**

**Person p = new Person();**

**p.name = "홍길동";**

**p.age = 30;**

**p.height = 170.0;**

**System.out.println(p.name); // 홍길동 출력**

**System.out.println(p.age); // 30 출력**

**System.out.println(p.height); // 170.0 출력**

**주의 사항:**

* **필드는 객체의 상태를 나타내는 중요한 데이터 요소이므로 신중하게 선언하고 관리해야 합니다.**
* **잘못된 필드 접근 또는 변경은 예상치 못한 프로그램 동작을 야기할 수 있으므로 주의해야 합니다.**
* **정보 은닉을 위해 필요한 경우 적절한 접근 지정자를 사용하여 필드 접근을 제어해야 합니다.**

**JAVA 클래스에서 필드는 객체의 상태를 저장하고 관리하는 데 중요한 역할을 합니다. 필드를 효과적으로 활용 객체 지향 프로그래밍의 장점을 극대화할 수 있습니다.**

## **JAVA 클래스에서 생성자를 생성하는 역할**

**JAVA 클래스에서 생성자는 다음과 같은 중요한 역할을 합니다.**

**1. 객체 초기화:**

* **생성자는 객체가 생성될 때 자동으로 호출되어 객체의 필드(멤버 변수)를 초기화하는 역할을 합니다.**
* **객체가 생성된 직후 사용 가능한 상태로 만들기 위해 필드에 적절한 값을 할당합니다.**
* **예를 들어, Point 클래스의 생성자는 x 및 y 좌표를 초기화할 수 있습니다.**

**2. 필드 접근 제어:**

* **생성자를 통해서만 필드에 값을 할당하도록 제한하여 데이터 유효성을 보장하고 객체의 상태 일관성을 유지할 수 있습니다.**
* **외부 코드에서 임의로 필드 값을 변경하는 것을 방지하여 객체의 불안정한 동작을 예방할 수 있습니다.**

**3. 객체 구성:**

* **생성자는 객체 생성에 필요한 다른 객체를 생성하고 구성하는 역할도 수행할 수 있습니다.**
* **객체 간의 연관성을 설정하고 초기 상태를 정의하는 데 활용됩니다.**
* **예를 들어, Car 클래스의 생성자는 엔진, 타이어, 도어 등 다른 객체를 생성하고 초기화하여 자동차 객체를 구성할 수 있습니다.**

**4. 객체 생성 방식 제어:**

* **다양한 생성자를 제공함으로써 객체 생성 방식을 제어할 수 있습니다.**
* **객체 생성 시 필요한 정보를 명시적으로 전달하도록 하여 객체의 상태를 보다 정확하게 초기화할 수 있습니다.**
* **예를 들어, Rectangle 클래스는 폭과 높이, 넓이와 둘레를 지정하는 다양한 생성자를 제공하여 객체를 생성할 수 있는 방법을 제공합니다.**

**생성자의 주요 특징:**

* **생성자 이름은 반드시 클래스 이름과 동일해야 합니다.**
* **생성자는 리턴 타입을 가지고 있지 않습니다.**
* **객체가 생성될 때마다 자동으로 호출됩니다.**
* **여러 개의 생성자를 정의할 수 있으며, 매개변수의 개수와 타입에 따라 구분됩니다.**
* **명시적으로 생성자를 정의하지 않은 경우, 컴파일러에서 기본 생성자를 자동으로 추가합니다.**

**생성자를 활용한 객체 생성 예시:**

**Java**

**class Point {**

**int x, y;**

**// 기본 생성자**

**public Point() {**

**this(0, 0);**

**}**

**// 매개변수를 가진 생성자**

**public Point(int x, int y) {**

**this.x = x;**

**this.y = y;**

**}**

**}**

**public class Main { public static void main(String[] args) { // 기본 생성자를 사용한 객체 생성 Point p1 = new Point(); System.out.println("p1: (" + p1.x + ", " + p1.y + ")"); // 매개변수를 가진 생성자를 사용한 객체 생성 Point p2 = new Point(3, 4); System.out.println("p2: (" + p2.x + ", " + p2.y + ")"); } }**

**public class Main {**

**public static void main(String[] args) {**

**// 기본 생성자를 사용한 객체 생성**

**Point p1 = new Point();**

**System.out.println("p1: (" + p1.x + ", " + p1.y + ")");**

**// 매개변수를 가진 생성자를 사용한 객체 생성**

**Point p2 = new Point(3, 4);**

**System.out.println("p2: (" + p2.x + ", " + p2.y + ")");**

**}**

**}**

**위 예시에서 Point 클래스는 기본 생성자와 매개변수를 가진 생성자를 제공합니다. 기본 생성자는 x와 y 좌표를 0으로 초기화하고, 매개변수를 가진 생성자는 전달된 값으로 x와 y 좌표를 초기화합니다.**

**결론:**

**생성자는 JAVA 클래스에서 필수적인 요소이며, 객체 초기화, 필드 접근 제어, 객체 구성, 객체 생성 방식 제어 등 중요한 역할을 합니다. 적절한 생성자를 활용하여 객체의 상태를 정확하게 초기화하고 안정적인 동작을 보장하는 것이 중요합니다.**

## **JAVA 클래스에서 메소드를 생성하는 역할**

**메소드는 클래스의 기능을 구현하는 코드 블록입니다. 객체의 \*\*상태(멤버 변수)\*\*를 변경하거나 정보를 처리하는 역할을 합니다.**

**메소드를 생성하면 다음과 같은 이점이 있습니다.**

* **코드 재사용성 향상: 반복되는 작업을 메소드로 만들어 여러 번 호출하여 코드를 간결하게 작성하고 유지 관리를 용이하게 합니다.**
* **정보 은닉: 메소드의 내부 구현을 숨기고 외부에서 필요한 기능만 제공하여 코드의 복잡성을 줄이고 변경 관리를 용이하게 합니다.**
* **객체 간 협력: 메소드를 통해 서로 다른 객체 간에 데이터를 전달하고 공유하여 응용 프로그램의 기능을 확장합니다.**

**메소드 선언 구조**

**Java**

**접근제어자 리턴타입 메소드명(매개변수목록) {**

**// 메소드 본문**

**}**

* **접근제어자: public, private, protected 등으로 메소드의 접근 범위를 제어합니다.**
* **리턴타입: 메소드가 반환하는 데이터 타입을 지정합니다. void는 값을 반환하지 않는 경우를 의미합니다.**
* **메소드명: 메소드를 식별하는 고유한 이름입니다.**
* **매개변수 목록: 메소드를 호출할 때 전달되는 값을 정의합니다.**
* **메소드 본문: 메소드가 수행하는 actual 작업을 코드로 작성합니다.**

**메소드 호출**

**Java**

**객체.메소드명(인수목록);**

* **객체: 메소드를 호출하는 객체를 참조합니다.**
* **메소드명: 호출하려는 메소드의 이름을 지정합니다.**
* **인수 목록: 메소드에 전달하려는 값을 쉼표로 구분하여 나열합니다.**

**메소드 종류**

* **인스턴스 메소드: 객체의 상태를 변경하거나 정보를 처리하는 메소드입니다. 객체를 통해 호출됩니다.**
* **클래스 메소드: 클래스 자체와 관련된 작업을 수행하는 메소드입니다. 클래스 이름을 통해 호출됩니다.**
* **생성자: 객체를 생성할 때 자동으로 호출되는 메소드입니다. 객체의 초기 상태를 설정하는 역할을 합니다.**

**예시**

**Java**

**public class Point {**

**private int x;**

**private int y;**

**public Point(int x, int y) { // 생성자**

**this.x = x;**

**this.y = y;**

**}**

**public int getX() { // getter 메소드**

**return x;**

**}**

**public void setX(int x) { // setter 메소드**

**this.x = x;**

**}**

**public double getDistance(Point p) { // 인스턴스 메소드**

**double dx = this.x - p.x;**

**double dy = this.y - p.y;**

**return Math.sqrt(dx \* dx + dy \* dy);**

**}**

**public static void main(String[] args) {**

**Point p1 = new Point(10, 20);**

**Point p2 = new Point(30, 40);**

**System.out.println(p1.getX()); // 10**

**p1.setX(50);**

**System.out.println(p1.getX()); // 50**

**System.out.println(p1.getDistance(p2)); // 28.284271247460914**

**}**

**}**

위 코드에서 Point 클래스는 x와 y 좌표를 가진 점을 나타내는 클래스입니다.

* Point(int x, int y): 객체를 생성하고 초기화하는 생성자입니다.
* getX(): x 좌표를 반환하는 getter 메소드입니다.
* setX(int x): x 좌표를 설정하는 setter 메소드입니다.

## **JAVA 메소드 오버로딩**

**메소드 오버로딩**은 동일한 이름을 가진 메소드를 여러 개 정의하는 것을 의미하지만, 각 메소드는 서로 다른 매개변수 목록을 가져야 합니다. 즉, 메소드의 **시그니처**가 달라야 합니다.

**시그니처**는 메소드 이름과 매개변수 목록으로 구성됩니다. 매개변수의 개수, 타입, 순서가 모두 시그니처에 영향을 미칩니다.

따라서, 메소드 이름만 같으면 오버로딩이 된다고 생각하면 오류가 발생할 수 있습니다. 반드시 시그니처까지 같아야 한다는 점을 기억해야 합니다.

**메소드 오버로딩의 장점:**

* 코드 가독성 향상
* 코드 재사용성 향상
* 유연성 증가

**메소드 오버로딩 예제:**

Java

class Calculator {

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

}

위 예제에서 Calculator 클래스에는 두 개의 add 메소드가 있습니다. 하지만 두 메소드는 매개변수의 타입이 다르기 때문에 오버로딩이 가능합니다.

다음과 같이 메소드를 호출할 수 있습니다.

Java

Calculator calculator = new Calculator();

int result1 = calculator.add(10, 20);

double result2 = calculator.add(10.5, 20.5);

**메소드 오버로딩 제약사항:**

* 리턴 타입만으로는 오버로딩을 구분할 수 없습니다.
* 매개변수의 개수만으로는 오버로딩을 구분할 수 없습니다.
* varargs 매개변수는 마지막에 와야 합니다.

**메소드 오버로딩과 오버라이딩 차이점:**

* **메소드 오버로딩:** 동일한 클래스 내에서 동일한 이름의 메소드를 여러 개 정의하는 것
* **메소드 오버라이딩:** 상속 관계에서 상위 클래스의 메소드를 하위 클래스에서 재정의하는 것

## **JAVA 메소드 오버라이딩 (Method Overriding)**

**메소드 오버라이딩**이란 상속 관계에 있는 **부모 클래스**에서 정의된 메소드를 **자식 클래스**에서 **동일한 시그니처**를 가지는 메소드로 **다시 정의**하는 것을 말합니다. 쉽게 말해, 부모 클래스가 제공하는 기본 기능을 자식 클래스의 특성에 맞게 수정하거나 확장하는 기능이라고 생각하면 됩니다.

**메소드 오버라이딩의 장점:**

* **코드 재사용성 향상:** 부모 클래스에서 정의된 기본 기능을 그대로 사용하거나 필요에 따라 자식 클래스에서 수정하여 코드 중복을 줄일 수 있습니다.
* **다형성 구현:** 동일한 메소드 이름으로 다양한 동작을 하는 메소드를 정의하여 코드의 유연성을 높일 수 있습니다.
* **객체 간의 교환성 향상:** 부모 클래스 타입의 객체를 자식 클래스 객체로 대체하여 사용할 수 있어 코드의 유지 관리성을 높일 수 있습니다.

**메소드 오버라이딩의 조건:**

* **메소드 시그니처 일치:** 메소드 이름, 매개 변수 목록, 반환 형식이 **정확히 동일**해야 합니다.
* **접근 제어자:** 오버라이딩된 메소드의 접근 제어자는 **부모 메소드보다 제한적이거나 같아야** 합니다.
* **예외 선언:** 오버라이딩된 메소드는 **부모 메소드에서 선언되지 않은 예외를 추가로 선언할 수 없습니다.**
* **인스턴스 메소드 vs. 정적 메소드:** 인스턴스 메소드는 인스턴스 메소드로, 정적 메소드는 정적 메소드로 오버라이딩해야 합니다.

**메소드 오버라이딩 예시:**

Java

class Animal {

public void speak() {

System.out.println("동물이 울다.");

}

}

class Dog extends Animal {

@Override

public void speak() {

System.out.println("멍멍 짖다.");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Dog();

animal.speak(); // 멍멍 짖다. (Dog 클래스의 오버라이딩된 메소드가 호출됨)

}

}

위 예시에서는 Animal 클래스의 speak() 메소드를 Dog 클래스에서 오버라이딩하여 동물 종류에 따라 서로 다른 울음소리를 출력하도록 구현했습니다.

**주의 사항:**

* 메소드 오버라이딩을 잘못 사용하면 예상치 못한 동작을 유발할 수 있으므로 주의해야 합니다.
* 오버라이딩된 메소드를 호출할 때는 항상 **실제 객체의 클래스**를 고려해야 합니다.
* @Override 어노테이션을 사용하면 컴파일러가 오버라이딩 오류를 체크하는 데 도움이 됩니다.