1 코딩을 시작하거나 AI로 코드를 생성하세요.

Chapter 01 자바 시작하기

01-1 프로그래밍 언어와 자바

- **자바 소개**:
  - 자바는 객체 지향 프로그래밍 언어로, "Write Once, Run Anywhere"가 가능하도록 설계되었습니다.
  - 강력한 보안 기능과 풍부한 라이브러리 지원을 통해 다양한 애플리케이션을 개발할 수 있습니다.
- **IntelliJ IDEA 설치**:
  - IntelliJ IDEA 다운로드에서 커뮤니티 에디션 다운로드 및 설치.
  - 설치 후 새로운 프로젝트를 생성하여 자바 개발 환경을 구축합니다.
- **환경 변수 설정**:
  - JAVA_HOME과 PATH 설정: 시스템 속성 > 고급 시스템 설정 > 환경 변수에서 설정.

```
JAVA_HOME: C:\Program Files\Java\jdk-15
PATH: %JAVA_HOME%\bin;
```

- **6가지 키워드로 끝내는 핵심 포인트**:
  1. **객체 지향**: 클래스와 객체를 중심으로 코드를 작성.
  2. **플랫폼 독립성**: JVM을 통해 다양한 플랫폼에서 실행.
  3. **안전성**: 강력한 메모리 관리 및 예외 처리.
  4. **동적 로딩**: 필요한 클래스만 로딩하여 실행.
  5. **고성능**: JIT 컴파일러를 통한 최적화.
  6. **다중 스레드**: 병렬 프로그래밍 지원.
- **확인 문제**:
  1. 자바의 주요 특징은 무엇인가?
  2. IntelliJ IDEA에서 자바 프로젝트를 어떻게 생성하는가?

01-2 IntelliJ IDEA 개발 환경 구축

- **IntelliJ IDEA 설치**:
  - IntelliJ IDEA 다운로드에서 커뮤니티 에디션 다운로드 및 설치.
  - 설치 후 새로운 프로젝트를 생성하여 자바 개발 환경을 구축합니다.
- **워크스페이스**:
  - 프로젝트 파일들이 저장되는 폴더입니다. 프로젝트 생성 시 워크스페이스 폴더를 지정합니다.
- **퍼스펙티브와 뷰**:
  - IntelliJ IDEA의 다양한 뷰(프로젝트 뷰, 구조 뷰 등)를 통해 코드 구조를 파악하고 탐색할 수 있습니다.
- **4가지 키워드로 끝내는 핵심 포인트**:
  1. **프로젝트 뷰**: 프로젝트 파일 및 폴더 구조 탐색.
  2. **코드 편집기**: 코드 작성 및 수정.
  3. **터미널**: 명령어 입력 및 실행.
  4. **디버거**: 코드 디버깅 및 문제 해결.
- **확인 문제**:
  1. IntelliJ IDEA에서 프로젝트 뷰를 사용하는 방법은?
  2. 터미널에서 자바 코드를 컴파일하고 실행하는 방법은?

01-3 자바 프로그램 개발 과정

- **바이트 코드 파일과 자바 가상 기계**:
  - 자바 소스 코드(.java)를 컴파일하면 바이트 코드(.class)가 생성되고, JVM이 이를 실행합니다.
- **프로젝트 생성부터 실행까지**:
  - IntelliJ IDEA에서 새 프로젝트 생성 후 간단한 자바 프로그램 작성.

```
public class HelloWorld {
  public static void main(String[] args) {
      System.out.println("Hello, World!");
  }
}
```

  - Run 버튼을 클릭하여 프로그램 실행.
- **명령 라인에서 컴파일하고 실행하기**:
  - 터미널에서 자바 파일 컴파일: `javac HelloWorld.java`
  - 실행: `java HelloWorld`
- **프로그램 소스 분석**:
  - 위 예제에서 `public class`, `public static void main` 등의 의미를 이해.
- **주석 사용하기**:
  - 단일 행 주석: `//`
  - 다중 행 주석: `/* ... */`
- **실행문과 세미콜론(;)**:
  - 자바의 각 명령문 끝에는 세미콜론을 붙입니다.
- **좀 더 알아보기 제공 소스 파일 이용하기**:
  - IntelliJ IDEA의 샘플 프로젝트를 활용하여 더 복잡한 예제를 실습.
- **6가지 키워드로 끝내는 핵심 포인트**:
  1. **컴파일**: 소스 코드를 바이트 코드로 변환.
  2. **JVM**: 바이트 코드를 실행하는 가상 기계.
  3. **클래스**: 자바의 기본 단위.
  4. **메서드**: 클래스 내 기능을 구현.
  5. **main 메서드**: 프로그램 실행 시작점.
  6. **주석**: 코드 설명 및 문서화.
- **확인 문제**:
  1. 자바 프로그램의 기본 구조는?
  2. 주석의 종류와 사용 방법은?

Chapter 02 변수와 타입

02-1 변수

- **변수 선언**:

```
int number;
```

- **값 저장**:

```
number = 10;
```

- **변수 사용**:

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

- **변수 사용 범위**:
  - 지역 변수: 메서드 내에서 선언된 변수.
  - 전역 변수: 클래스 내에서 선언된 변수.
- **4가지 키워드로 끝내는 핵심 포인트**:
  1. **선언**: 변수의 타입과 이름을 정의.
  2. **초기화**: 변수에 초기값 할당.
  3. **스코프**: 변수의 유효 범위.
  4. **타입**: 변수의 데이터 종류.
- **확인 문제**:
  1. 변수 선언과 초기화의 차이는 무엇인가?
  2. 지역 변수와 전역 변수의 차이는?

02-2 기본 타입

- **정수 타입**:

```
int num = 100;
long bigNum = 100000L;
```

- **실수 타입**:

```
float f = 10.5f;
double d = 20.5;
```

- **논리 타입**:

```
boolean isTrue = true;
```

- **5가지 키워드로 끝내는 핵심 포인트**:
  1. **정수**: int, long 등.
  2. **실수**: float, double 등.
  3. **논리**: boolean.
  4. **문자**: char.
  5. **문자열**: String.
- **확인 문제**:
  1. 자바의 기본 데이터 타입은?
  2. 정수형과 실수형의 차이는?

02-3 타입 변환

- **자동 타입 변환**:

```
int i = 100;
double d = i; // 자동 변환
```

- **강제 타입 변환**:

```
double d = 100.5;
int i = (int) d; // 강제 변환
```

- **정수 연산에서의 자동 타입 변환**:

```
byte b = 10;
int i = b + 10; // byte가 int로 자동 변환
```

- **실수 연산에서의 자동 타입 변환**:

```
float f = 10.5f;
double d = f * 2.0; // float가 double로 자동 변환
```

- **문자열을 기본 타입으로 강제 타입 변환**:

```
String s = "123";
int i = Integer.parseInt(s);
```

- **5가지 키워드로 끝내는 핵심 포인트**:
  1. **자동 변환**: 작은 타입에서 큰 타입으로 자동 변환.
  2. **강제 변환**: 큰 타입에서 작은 타입으로 명시적 변환.
  3. **정수 변환**: 정수형 간의 타입 변환.
  4. **실수 변환**: 실수형 간의 타입 변환.
  5. **문자열 변환**: 문자열을 다른 타입으로 변환.
- **확인 문제**:
  1. 자동 타입 변환과 강제 타입 변환의 차이는?
  2. 문자열을 정수로 변환하는 방법은?

02-4 변수와 시스템 입출력

- **모니터로 변수값 출력하기**:

```
int num = 10;
System.out.println(num);
```

- **키보드에서 입력된 내용을 변수에 저장**:

```
import java.util.Scanner;
Scanner scanner = new Scanner(System.in);
int num = scanner.nextInt();
System.out.println("입력한 값: " + num);
```

- **5가지 키워드로 끝내

는 핵심 포인트**:

1. **System.out**: 콘솔 출력 스트림.
2. **System.in**: 콘솔 입력 스트림.
3. **Scanner**: 입력 처리 클래스.
4. **print와 println**: 출력 방법.
5. **입력 값 처리**: nextInt, nextLine 등의 메서드.

- **확인 문제**:
    1. 콘솔에서 입력을 받는 방법은?
    2. System.out과 System.in의 차이는?

Chapter 03 변수와 데이터 입력

03-1 연산자와 연산식

- **연산자의 종류**:
    - 산술 연산자: +, -, *, /, %
    - 비교 연산자: ==, !=, >, <, >=, <=
    - 논리 연산자: &&, ||, !
- **연산의 방향과 우선순위**:
    - 산술 > 비교 > 논리 연산 순으로 우선순위 적용.
- **4가지 키워드로 끝내는 핵심 포인트**:
    1. **산술 연산자**: 기본 수학 연산.
    2. **비교 연산자**: 값 비교.
    3. **논리 연산자**: 조건 논리.
    4. **우선순위**: 연산자의 실행 순서.
- **표로 정리하는 핵심 포인트**: | 연산자 | 설명 | |---|---| | + | 더하기 | | - | 빼기 | | * | 곱하기 | | / | 나누기 | | % | 나머지 |
- **확인 문제**:
    1. 산술 연산자와 비교 연산자의 차이는?
    2. 논리 연산자의 사용 예는?

03-2 연산자의 종류

- **단항 연산자**:

```
int num = 10;
num++;
```

- **이항 연산자**:

```
int a = 10;
int b = 20;
int result = a + b;
```

- **삼항 연산자**:

```
int a = 10;
int b = 20;
int max = (a > b) ? a : b;
```

- **5가지 키워드로 끝내는 핵심 포인트**:

1. **단항 연산자**: 피연산자 하나.
2. **이항 연산자**: 피연산자 둘.
3. **삼항 연산자**: 조건에 따른 값 선택.
4. **증감 연산자**: ++, --.
5. **조건 연산자**: ? :.

- **확인 문제**:
  1. 단항 연산자와 이항 연산자의 차이는?
  2. 삼항 연산자의 사용 예는?

Chapter 04 조건문과 반복문

04-1 조건문: if문, switch문

- **if문**:

```
int a = 10;
if (a > 5) {
    System.out.println("a는 5보다 큽니다.");
}
```

- **if-else문**:

```
int a = 10;
if (a > 5) {
    System.out.println("a는 5보다 큽니다.");
} else {
    System.out.println("a는 5보다 작습니다.");
}
```

- **if-else if-else문**:

```
int a = 10;
if (a > 10) {
    System.out.println("a는 10보다 큽니다.");
} else if (a == 10) {
    System.out.println("a는 10입니다.");
} else {
    System.out.println("a는 10보다 작습니다.");
}
```

- **switch문**:

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("월요일");
        break;
    case 2:
        System.out.println("화요일");
        break;
    case 3:
```

```
            System.out.println("수요일");
            break;
        default:
            System.out.println("기타");
            break;
    }
```

- **4가지 키워드로 끝내는 핵심 포인트**:
    1. **if문**: 조건에 따라 분기.
    2. **else문**: if의 조건이 거짓일 때 실행.
    3. **else if문**: 여러 조건 검사.
    4. **switch문**: 값에 따라 여러 분기 처리.
- **그림으로 정리하는 핵심 포인트**:
    - 조건문의 흐름도를 그려서 이해.
- **확인 문제**:
    1. if-else 문과 switch 문의 차이는?
    2. 조건문에서 break 문의 역할은?

04-2 반복문: for문, while문, do-while문

- **for문**:

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

- **while문**:

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

- **do-while문**:

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 5);
```

- **break문**:

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        break;
    }
```

```
        System.out.println(i);
    }
```

- **continue문**:

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println(i);
}
```

- **5가지 키워드로 끝내는 핵심 포인트**:
  1. **for문**: 반복 횟수 지정.
  2. **while문**: 조건 만족 시 반복.
  3. **do-while문**: 최소 한 번 실행.
  4. **break문**: 반복문 종료.
  5. **continue문**: 현재 반복 건너뛰기.
- **확인 문제**:
  1. for문과 while문의 차이는?
  2. break문과 continue문의 사용 예는?

Chapter 05 참조 타입

05-1 참조 타입과 참조 변수

- **기본 타입과 참조 타입**:
  - 기본 타입: int, double 등.
  - 참조 타입: 클래스, 배열, 인터페이스 등.
- **참조 변수의 ==, != 연산**:
  - 참조 변수의 주소 비교.

```
String a = new String("hello");
String b = new String("hello");
System.out.println(a == b); // false
System.out.println(a.equals(b)); // true
```

- **null과 NullPointerException**:
  - null: 참조 변수가 객체를 참조하지 않는 상태.
  - NullPointerException: null 참조 시 발생하는 예외.

```
String str = null;
System.out.println(str.length()); // NullPointerException
```

- **String 타입**:
  - 문자열을 처리하는 클래스.

```
String greeting = "Hello, World!";
System.out.println(greeting);
```

- **6가지 키워드로 끝내는 핵심 포인트**:
  1. **기본 타입**: 데이터 자체를 저장.
  2. **참조 타입**: 객체의 주소를 저장.
  3. **null**: 객체를 참조하지 않음.
  4. **NullPointerException**: null 접근 시 발생.
  5. **String**: 문자열 처리 클래스.
  6. **equals**: 문자열 비교 메서드.
- **확인 문제**:
  1. 기본 타입과 참조 타입의 차이는?
  2. NullPointerException은 언제 발생하는가?

05-2 배열

- **배열이란?**:
  - 동일한 타입의 데이터 집합을 저장하는 자료 구조.

```
int[] numbers = new int[5];
```

- **배열 선언**:

```
int[] numbers;
```

- **배열 생성**:

```
numbers = new int[5];
```

- **배열 길이**:

```
System.out.println(numbers.length);
```

- **명령 라인 입력**:

```
public class CommandLineExample {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

- **다차원 배열**:

```
int[][] matrix = new int[3][3];
```

- **객체를 참조하는 배열**:

```java
String[] names = new String[3];
```

```
names[0] = "Alice";
names[1] = "Bob";
names[2] = "Charlie";
```

- **배열 복사**:

```
int[] original = {1, 2, 3};
int[] copy = Arrays.copyOf(original, original.length);
```

- **향상된 for문**:

```
for (int number : numbers) {
    System.out.println(number);
}
```

- **7가지 키워드로 끝내는 핵심 포인트**:
  1. **배열 선언**: 배열 변수 정의.
  2. **배열 생성**: 배열 초기화.
  3. **배열 길이**: 배열의 크기.
  4. **다차원 배열**: 2차원 이상의 배열.
  5. **객체 배열**: 객체를 요소로 갖는 배열.
  6. **배열 복사**: 배열 복사 방법.
  7. **향상된 for문**: 배열 순회 방법.
- **확인 문제**:
  1. 배열의 선언과 생성의 차이는?
  2. 다차원 배열을 사용하는 예는?

05-3 열거 타입

- **열거 타입 선언**:

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

- **열거 타입 변수**:

```
Day day = Day.MONDAY;
```

- **4가지 키워드로 끝내는 핵심 포인트**:
  1. **열거 타입**: 열거형 정의.
  2. **열거 변수**: 열거형 변수.
  3. **상수 집합**: 고정된 상수 집합.
  4. **switch와 열거형**: 열거형과 switch문의 사용.
- **확인 문제**:
  1. 열거 타입의 장점은?
  2. 열거 타입을 사용하는 예는?

Chapter 06 클래스

06-1 객체 지향 프로그래밍

- **객체의 상호작용**:
  - 객체 간의 메시지 전달로 동작 수행.

```
class Car {
  void start() {
      System.out.println("Car started");
  }
}
```

- **객체 간의 관계**:
  - 상속, 포함 등의 관계 정의.

```
class Engine {}
class Car {
  Engine engine = new Engine();
}
```

- **객체와 클래스**:
  - 클래스: 객체를 정의하는 틀.
  - 객체: 클래스를 인스턴스화한 것.

```
class Car {
  String model;
  void drive() {
      System.out.println("Driving " + model);
  }
}
Car myCar = new Car();
myCar.model = "Tesla";
myCar.drive();
```

- **클래스 선언**:

```
public class Car {
    // 필드, 생성자, 메서드 정의
}
```

- **객체 생성과 클래스 변수**:

```
Car myCar = new Car();
```

- **클래스의 구성 멤버**:
  - 필드, 메서드, 생성자 등.
- **6가지 키워드로 끝내는 핵심 포인트**:
  1. **클래스**: 객체의 설계도.
  2. **객체**: 클래스의 인스턴스.
  3. **필드**: 객체의 속성.

4. **메서드**: 객체의 동작.
5. **생성자**: 객체 초기화 메서드.
6. **참조 변수**: 객체를 가리키는 변수.

- **확인 문제**:
  1. 클래스와 객체의 차이는?
  2. 객체를 생성하는 방법은?

06-2 필드

- **필드 선언**:

```
class Car {
    String model;
}
```

- **필드 사용**:

```
Car myCar = new Car();
myCar.model = "Tesla";
```

- **2가지 키워드로 끝내는 핵심 포인트**:
  1. **필드 선언**: 클래스 내 변수 정의.
  2. **필드 사용**: 객체를 통해 접근.
- **확인 문제**:
  1. 필드 선언 방법은?
  2. 필드를 사용하는 예는?

06-3 생성자

- **기본 생성자**:

```
class Car {
    Car() {
        System.out.println("Car created");
    }
}
```

- **생성자 선언**:

```
class Car {
    String model;
    Car(String model) {
        this.model = model;
    }
}
```

- **필드 초기화**:

```
Car myCar = new Car("Tesla");
```

- **생성자 오버로딩**:

```
class Car {
    Car() {}
    Car(String model) {
        this.model = model;
    }
}
```

- **다른 생성자 호출: this()**:

```
class Car {
    String model;
    Car() {
        this("Unknown");
    }
    Car(String model) {
        this.model = model;
    }
}
```

- **6가지 키워드로 끝내는 핵심 포인트**:
  1. **기본 생성자**: 매개변수 없는 생성자.
  2. **오버로딩**: 여러 생성자 정의.
  3. **this()**: 다른 생성자 호출.
  4. **필드 초기화**: 생성자를 통한 초기화.
  5. **생성자**: 객체 초기화 메서드.
  6. **다형성**: 다양한 형태의 생성자 사용.
- **확인 문제**:
  1. 기본 생성자와 오버로딩된 생성자의 차이는?
  2. this()의 역할은?

06-4 메서드

- **메서드 선언**:

```
class Car {
    void start() {
        System.out.println("Car started");
    }
}
```

- **return문**:

```
class Car {
    String getModel() {
        return "Tesla";
    }
}
```

- **메서드 호출**:

```
Car myCar = new Car();
myCar.start();
```

- **메서드 오버로딩**:

```
class Car {
    void start() {}
    void start(String model) {
        System.out.println(model + " started");
    }
}
```

- **6가지 키워드로 끝내는 핵심 포인트**:
  1. **메서드 선언**: 메서드 정의.
  2. **return**: 값 반환.
  3. **호출**: 메서드를 실행.
  4. **오버로딩**: 여러 메서드 정의.
  5. **인자**: 메서드 매개변수.
  6. **다형성**: 다양한 형태의 메서드 사용.
- **확인 문제**:
  1. 메서드 오버로딩이란?
  2. return문의 역할은?

06-5 인스턴스 멤버와 정적 멤버

- **인스턴스 멤버와 this**:

```
class Car {
    String model;
    void setModel(String model) {
        this.model = model;
    }
}
```

- **정적 멤버와 static**:

```
class Car {
    static int count;
    Car() {
        count++;
    }
}
```

- **싱글톤**:

```
class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() {}
```

```
    public static Singleton getInstance() {
        return instance;
    }
}
```

- **final 필드와 상수**:

```
class Car {
    final String model = "Tesla";
}
```

- **7가지 키워드로 끝내는 핵심 포인트**:
  1. **this**: 현재 객체 참조.
  2. **static**: 클래스 멤버.
  3. **싱글톤**: 객체의 유일성 보장.
  4. **final**: 상수 필드.
  5. **인스턴스 멤버**: 객체별로 존재.
  6. **정적 멤버**: 클래스별로 존재.
  7. **상수**: 변경 불가한 값.
- **확인 문제**:
  1. 인스턴스 멤버와 정적 멤버의 차이는?
  2. 싱글톤

패턴이란?

06-6 패키지와 접근 제한자

- **패키지 선언**:

```
package mypackage;
```

- **접근 제한자**:
  - public: 모든 클래스에서 접근 가능.
  - private: 해당 클래스 내에서만 접근 가능.
  - protected: 동일 패키지 및 서브 클래스에서 접근 가능.
  - default: 동일 패키지에서 접근 가능.
- **클래스의 접근 제한**:

```
public class Car {}
```

- **생성자의 접근 제한**:

```
class Car {
    private Car() {}
}
```

- **필드와 메소드의 접근 제한**:

```
class Car {
    private String model;
```

```
    public String getModel() {
        return model;
    }
}
```

- **Getter와 Setter 메소드**:

```
class Car {
    private String model;
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}
```

- **4가지 키워드로 끝내는 핵심 포인트**:
    1. **public**: 공개 접근.
    2. **private**: 비공개 접근.
    3. **protected**: 패키지 및 서브 클래스 접근.
    4. **default**: 패키지 접근.
- **확인 문제**:
    1. 접근 제한자의 종류와 역할은?
    2. Getter와 Setter의 사용 예는?

Chapter 07 상속

07-1 상속

- **클래스 상속**:

```
class Vehicle {}
class Car extends Vehicle {}
```

- **부모 생성자 호출**:

```
class Vehicle {
    Vehicle() {
        System.out.println("Vehicle created");
    }
}
class Car extends Vehicle {
    Car() {
        super();
        System.out.println("Car created");
    }
}
```

- **메소드 재정의**:

```
class Vehicle {
    void start() {
        System.out.println("Vehicle started");
    }
}
class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car started");
    }
}
```

- **final 클래스와 final 메소드**:

```
final class Vehicle {}
class Car extends Vehicle {} // 오류 발생

class Car {
    final void start() {}
}
class Tesla extends Car {
    @Override
    void start() {} // 오류 발생
}
```

- **좀 더 알아보기 protected 접근 제한자**:
  - 상속 관계에서 자식 클래스가 부모 클래스의 멤버에 접근할 수 있도록 합니다.

```
class Vehicle {
  protected String model;
}
class Car extends Vehicle {
  void printModel() {
      System.out.println(model);
  }
}
```

- **4가지 키워드로 끝내는 핵심 포인트**:
  1. **상속**: 클래스 간의 계층 관계.
  2. **super()**: 부모 생성자 호출.
  3. **오버라이딩**: 메서드 재정의.
  4. **final**: 변경 불가 선언.
- **확인 문제**:
  1. 상속의 장점은?
  2. 오버라이딩의 의미는?

07-2 타입 변환과 다형성

- **자동 타입 변환**:

```
Vehicle myCar = new Car();
```

- **필드의 다형성**:

```
class Car {
    void drive() {
        System.out.println("Driving a car");
    }
}
class ElectricCar extends Car {
    @Override
    void drive() {
        System.out.println("Driving an electric car");
    }
}
```

- **매개 변수의 다형성**:

```
class Mechanic {
    void repair(Vehicle v) {
        System.out.println("Repairing vehicle");
    }
}
```

- **강제 타입 변환**:

```
Vehicle v = new Car();
Car c = (Car) v;
```

- **객체 타입 확인**:

```
if (v instanceof Car) {
    Car c = (Car) v;
}
```

- **5가지 키워드로 끝내는 핵심 포인트**:
  1. **자동 변환**: 상속 관계에서 자동 변환.
  2. **강제 변환**: 명시적 타입 변환.
  3. **다형성**: 다양한 객체 사용.
  4. **instanceof**: 객체 타입 확인.
  5. **오버라이딩**: 메서드 재정의.
- **확인 문제**:
  1. 자동 타입 변환과 강제 타입 변환의 차이는?
  2. 다형성의 장점은?

07-3 추상 클래스

- **추상 클래스의 용도**:
  - 공통된 기능을 정의하고 하위 클래스에서 구체화합니다.

```
abstract class Animal {
  abstract void sound();
}
class Dog extends Animal {
  @Override
  void sound() {
      System.out.println("Bark");
  }
}
```

- **추상 클래스 선언**:

```
abstract class Animal {
    abstract void sound();
}
```

- **추상 메소드와 재정의**:

```
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}
```

- **2가지 키워드로 끝내는 핵심 포인트**:
  1. **추상 클래스**: 공통 기능 정의.
  2. **추상 메소드**: 하위 클래스에서 구현.

더블클릭 또는 Enter 키를 눌러 수정

1. 수상 클래스와 일반 클래스의 차이는?
2. 추상 메서드의 역할은?

이제 각 챕터와 소주제별로 중요한 내용들을 보다 쉽게 설명하고, 간단한 예제 코드를 추가하여 정리했습니다. 이를 기반으로 학기말 시험 준비에 도움이 되길 바랍니다. 추가적인 질문이나 설명이 필요하다면 언제든지 문의하세요!