2~11장 요약정리

1. 변수와 타입
2. 연산자
3. 조건문과 반복문
4. 참조타입
5. 클래스
6. 상속
7. 인터페이스
8. 중첩 선언과 익명 객체
9. 라이브러리와 모듈
10. 예외처리

요약정리

항목	정의 및 설명	사용 예시
변수와 타입	변수는 데이터를 저장하는 메모리 공간이며, 타입은 데이터의 종류를 나타냅니다.	int age = 25; (정수형 변수 선언 및 초기화)
연산자	수학적 연산이나 비교, 논리 연산을 수행하는 기호 또는 기능을 말합니다.	int result = 5 + 3; (덧셈 연산)
조건문과 반복문	조건문은 주어진 조건에 따라 코드 블록을 실행하거나 건너뛰는데 사용됩니다. 반복문은 주어진 조건이 충족될 때까지 코드 블록을 반복 실행합니다.	if (score >= 60) { /* 코드 실행 */ } (if 조건문 예시)
참조타입	참조타입은 객체의 주소를 저장하고, 해당 객체의 메서드 및 속성에 접근할 수 있도록 합니다.	String name = "John"; (문자열 참조타입 변수 선언 및 초기화)
클래스	객체를 만들기 위한 설계도로, 속성(필드)과 행동(메서드)을 포함합니다.	class Dog { /* 클래스 정의 */ } (Dog 클래스 예시)
상속	부모 클래스의 속성과 메서드를 자식 클래스가 물려받는 것입니다.	class Child extends Parent { /* 코드 */ } (상속 예시)
인터페이스	추상 메서드의 집합으로, 다른 클래스에서 해당 메서드를 구현할 수 있습니다.	interface Shape { /* 메서드 선언 */ } (인터페이스 예시)
중첩 선언과 익명 객체	클래스 또는 인터페이스를 다른 클래스나 메서드 안에서 선언하는 것입니다. 익명 객체는 이름이 없는 객체를 생성할 때 사용됩니다.	class Outer { class Inner { /* 코드 */ } } (중첩 클래스 예시) MouseListener listener = new MouseListener() { /* 코드 */ } (익명 객체 생성 예시)
라이브러리와 모듈	라이브러리는 재사용 가능한 코드의 모음이고, 모듈은 프로그램의 일부로서 독립적인 기능을 수행합니다.	java.util 패키지의 ArrayList 클래스 (라이브러리 사용 예시)
예외처리	프로그램 실행 중 발생하는 예외를 처리하는 방법을 말합니다.	try { /* 코드 */ } catch(Exception e) { /* 코드 */ } (예외처리 구문 예시

1 변수와 타입

변수선언 방법: 변수는 타입과 변수명으로 선언되며, 타입 뒤에 변수명을 지정합니다.

```
int num;
char letter;
double salary;
boolean isStudent;
String name;
```

정수/문자/실수/논리/문자열 타입
- 정수 타입: **int**, **long**, **short**, **byte**
- 문자 타입: **char**
- 실수 타입: **float**, **double**
- 논리 타입: **boolean**
- 문자열 타입: **String**

자동 타입변환: 작은 크기의 타입에서 큰 크기의 타입으로는 자동으로 변환됩니다.

```
int num = 10;
double result = num;
```

강제타입변환: 큰 크기의 타입에서 작은 크기의 타입으로는 강제로 형변환을 해주어야 합니다.

```
double num = 10.5;
int result = (int) num;
```

1 변수와 타입

연산식에서 자동 타입 변환:
• 연산 중에 서로 다른 타입이 있을 때, 자동으로 큰 타입으로 형 변환 후 연산이 수행됩니다.

문자열에서 기본 타입으로 변환:

```
String str = "123";
int num = Integer.parseInt(str);
```

변수 사용 범위: 변수는 선언된 블록 내에서만 유효합니다. 메서드 내에서 선언된 변수는 해당 메서드에서만 사용할 수 있습니다.

콘솔로 변수값 출력:

```
int num = 10;
System.out.println("Number: " + num);
```

키보드 입력 데이터를 변수에 저장:

```
Scanner scanner = new Scanner(System.in);
int inputNum = scanner.nextInt(); // 키보드로부터 정수 입력받아 저장
```

2 연산자

부호/증감 연산자: 변수의 부호를 바꾸거나 값을 증감시킵니다.

```
int num = 10;
num = -num;          // 부호 연산자
num++;               // 증가 연산자
```

산술 연산자: 기본적인 사칙연산과 나머지 연산을 수행합니다.

```
int a = 10;
int b = 5;
int result = a + b;     // 덧셈 연산
```

- **오버플로우와 언더플로우**
  : 변수가 허용 범위를 넘어가거나 최소 값 미만으로 갈 때 발생합니다.

- **정확한 계산은 정수 연산으로**
  : 실수 연산은 부동소수점 방식으로 저장되므로 정확한 계산이 필요할 때는 정수 연산을 사용합니다.

- **나눗셈 연산 후 NaN과 Infinity 처리**
  : 0으로 나누는 경우 NaN(Not a Number) 또는 Infinity(무한대)로 결과가 나타납니다.

- **비교연산자**

: 두 개의 값을 비교하여 참 또는 거짓을 반환합니다.

```
int a = 10;  int b = 20;
boolean result = (a > b); // false 반환
```

2 연산자

논리 연산자: 논리식을 평가하여 참 또는 거짓을 반환합니다.

```
boolean isTrue = true;
boolean result = !isTrue; // false 반환
```

- **비트 논리 연산자**: 비트 단위로 AND, OR, XOR, NOT 연산을 수행합니다.
- **비트 이동 연산자**: 비트를 왼쪽 또는 오른쪽으로 이동시킵니다.

```
int num = 10;
num += 5; // num에 5를 더하여 다시 num에 대입
```

- **삼항(조건) 연산자**: 조건식에 따라 두 가지 값을 반환합니다.

```
int num = 10;
String result = (num > 0) ? "양수" : "음수";
```

- **연산의 방향과 우선순위**: 연산자에 따라 연산의 방향과 우선순위가 다르므로 주의해야 합니다.

3 조건문과 반복문

•**코드 실행 흐름 제어**: 조건문과 반복문을 사용하여 코드의 실행 흐름을 제어합니다.
• **if 문**: 주어진 조건식이 참일 경우에만 코드 블록을 실행합니다.

```
int num = 10;
if (num > 0) {
    System.out.println("양수입니다.");
}
```

• **switch 문**
: 주어진 표현식의 값을 기준으로 여러 개의 case 중 일치하는 경우에 해당하는 코드 블록을 실행합니다.

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

4 참조타입

- **데이터 타입 분류**: 참조 타입은 클래스, 인터페이스, 배열 등으로 구성됩니다.
- **메모리 사용 영역**: 참조 타입 변수는 스택 메모리에는 변수가 저장되고, 힙 메모리에는 객체가 저장됩니다.
- **참조 타입 변수의 ==, != 연산**: 참조 타입 변수의 ==, != 연산은 주소값을 비교합니다.
- **null과 NullPointerException**: 참조 변수가 null을 가리키는 경우 NullPointerException이 발생할 수 있습니다.
- **문자열 타입**: String 클래스로 문자열을 나타냅니다.

```
String str = "Hello, World!";
```

- **배열 타입**: 배열은 동일한 타입의 데이터를 순차적으로 저장합니다.

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

- **다차원 배열**: 배열의 요소로 또 다른 배열을 가질 수 있습니다.

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

- **객체를 참조하는 배열**: 객체를 배열의 요소로 가질 수 있습니다.

```
MyClass[] objArray = new MyClass[5];
```

- **배열 복사**: 배열을 복사하여 다른 배열에 저장할 수 있습니다.

```
int[] sourceArray = {1, 2, 3, 4, 5};
int[] destArray = new int[sourceArray.length];
System.arraycopy(sourceArray, 0, destArray, 0, sourceArray.length);
```

4 참조타입

- **배열 항목 반복을 위한 향상된 for 문**: 배열의 모든 요소를 순차적으로 접근할 때 사용됩니다.
- **main() 메소드의 String[] 매개변수 용도**: 명령행에서 프로그램을 실행할 때 전달되는 인수를 받습니다.
- **열거(Enum) 타입**: 몇 가지 고정된 값 중 하나를 나타내기 위해 사용됩니다.

•**배열 항목 반복을 위한 향상된 for 문**: 배열의 모든 요소를 순차적으로 접근할 때 사용됩니다.

```
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println(num);
}
```

• **main() 메소드의 String[] 매개변수 용도**: 명령행에서 프로그램을 실행할 때 전달되는 인수를 받습니다.

```
public static void main(String[] args) {
    // args 배열을 통해 전달된 인수에 접근 가능
 }
```

• **열거(Enum) 타입**: 몇 가지 고정된 값 중 하나를 나타내기 위해 사용됩니다.

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

5 클래스

- **객체 지향 프로그래밍**: 소프트웨어를 객체들의 모임으로 바라보고, 객체들 간의 상호작용으로 프로그램을 구현하는 프로그래밍 패러다임입니다.
- **객체와 클래스**: 객체는 실제로 존재하는 것으로, 클래스는 객체를 만들기 위한 설계도 또는 틀입니다.
- **클래스 선언**: 클래스는 class 키워드를 사용하여 선언하며, 멤버 변수와 메소드로 구성됩니다.

```
public class MyClass {
 // 멤버 변수와 메소드 선언
}
```

- **객체 생성과 클래스 변수**: 클래스를 바탕으로 객체를 생성할 수 있으며, 클래스 변수는 모든 객체가 공유하는 변수입니다.

```
MyClass obj = new MyClass();
```

- **클래스의 구성 멤버**: 클래스는 필드, 생성자, 메소드 등의 구성 멤버로 이루어집니다.
- **필드 선언과 사용**: 클래스 내부에 데이터를 저장하는 변수를 선언하고 사용합니다.

```
public class MyClass {
    private int num;
    public void setNum(int num) {
        this.num = num;
    }
}
```

5 클래스

•**생성자 선언과 호출**: 객체가 생성될 때 초기화를 위해 사용되는 특별한 메소드입니다.

```
public class MyClass {
    public MyClass() {
        // 생성자 내용
    }
}
```

• **메소드 선언과 호출**: 클래스 내부에서 수행할 동작을 정의하고 호출합니다.

```
public class MyClass {
    public void myMethod() {
        // 메소드 내용
    }
}
```

• **인스턴스 멤버**: 객체마다 가지는 변수나 메소드를 의미합니다.
• **정적 멤버**: 클래스에 고정된 변수나 메소드를 의미합니다.
• **final 필드와 상수**: 값을 변경할 수 없는 필드로, 상수를 표현할 때 사용됩니다.

```
public class MyClass {
    public static final int MAX_VALUE = 100;
}
```

5 클래스

- **패키지**: 클래스를 그룹화하기 위한 폴더 구조로, 클래스의 이름 공간을 관리합니다.
- **접근 제한자**: 멤버의 접근을 제한하는 키워드로, public, private, protected, default가 있습니다.
- **Getter와 Setter**: 외부에서 클래스의 필드에 접근하고 수정하기 위한 메소드입니다.
- **싱글톤 패턴**: 클래스의 인스턴스가 단 한 개만 생성되도록 보장하는 디자인 패턴입니다.

6 상속

• **상속 개념**: 상속은 부모 클래스의 특성을 자식 클래스가 물려받는 것을 의미합니다. 자식 클래스는 부모 클래스의 모든 멤버를 사용할 수 있습니다.

```
class Parent {
    int num;
    void display() {
        System.out.println("Number: " + num);
    }
}

class Child extends Parent {
            // Child 클래스는 Parent 클래스의 멤버를 상속받습니다.
}
```

• **클래스 상속**: extends 키워드를 사용하여 자식 클래스가 부모 클래스를 상속받습니다.

• **부모 생성자 호출**: 자식 클래스에서 부모 클래스의 생성자를 호출하여 부모 클래스의 초기화를 수행합니다.

```
class Parent {
    Parent(int num) {
            // 부모 클래스의 생성자
    }
}
```

```
class Child extends Parent {
    Child(int num) {
        super(num); // 부모 클래스의 생성자 호출
    }
}
```

6 상속

• **메소드 재정의**: 자식 클래스에서 부모 클래스의 메소드를 재정의하여 자식 클래스의 동작을 변경할 수 있습니다.

```
class Parent {
    void display() {
        System.out.println("Parent method");
    }
}
class Child extends Parent {
    void display() {
        System.out.println("Child method"); // 메소드 재정의
    }
}
```

• **final 클래스와 final 메소드**: final 키워드를 사용하여 클래스나 메소드를 상속이나 재정의를 금지할 수 있습니다.
• **protected 접근 제한자**: protected 멤버는 같은 패키지에 속한 클래스와 해당 클래스를 상속받은 클래스에서 접근할 수 있습니다.
• **타입 변환**: 부모 클래스 타입으로 자식 클래스의 객체를 참조할 수 있습니다.

```
Parent obj = new Child(); // 부모 클래스 타입으로 자식 클래스 객체 참조
```

6 상속

• **다형성**: 같은 메소드 호출에 대해 다른 동작을 수행하는 것을 의미합니다.
•**객체 타입 확인**: instanceof 연산자를 사용하여 객체의 실제 타입을 확인할 수 있습니다.

```
Parent obj = new Child();
if (obj instanceof Child) {
    System.out.println("This object is an instance of Child class");
}
```

• **추상 클래스**: 추상 클래스는 추상 메소드를 포함하고, 객체를 생성할 수 없는 클래스입니다.

```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        // draw 메소드 구현
    }
}
```

6 상속

• **봉인된 클래스**: 봉인된 클래스는 더 이상 상속할 수 없도록 제한된 클래스입니다. 이를 위해 final 클래스를 사용합니다.

```
// 봉인된 클래스 정의
public sealed class Shape permits Circle, Rectangle {
        // 클래스 내용
}
 // 봉인된 클래스를 상속받는 하위 클래스 정의
public final class Circle extends Shape {
        // 클래스 내용
}

public final class Rectangle extends Shape {
        // 클래스 내용
}
 // 봉인된 클래스를 상속받지 않는 클래스 정의 (컴파일 오류 발생)
public class Triangle extends Shape {
        // 클래스 내용
}
```

• 봉인된 클래스의 장점은 하위 클래스의 종류를 명확하게 정의할 수 있으며, 코드의 안정성을 높일 수 있다는 것

7 인터페이스

• **인터페이스 역할**: 인터페이스는 클래스의 기능을 정의하는데 사용되며, 다른 클래스에서 해당 기능을 구현할 수 있도록 합니다.
• **인터페이스와 구현 클래스 선언**: 인터페이스를 선언하고, 해당 인터페이스를 구현하는 클래스를 작성합니다.

```
interface Printable {
    void print();
}
class MyClass implements Printable {
    public void print() {
        System.out.println("Printing...");
    }
}
```

• **상수 필드**: 인터페이스 내에서 선언된 변수는 자동으로 상수로 취급됩니다.

```
interface Constants {
    int MAX_VALUE = 100;
}
```

• **추상 메소드**: 인터페이스는 메소드의 선언만을 포함하고, 구현은 없는 추상 메소드를 가질 수 있습니다.

```
interface Printable {
    void print();
}
```

7 인터페이스

• **디폴트 메소드**: 인터페이스에서 메소드의 기본 구현을 제공할 수 있습니다.

```
interface Printable {
    default void print() {
        System.out.println("Printing...");
    }
}
```

• **정적 메소드**: 인터페이스에서 정적 메소드를 정의할 수 있습니다.

```
interface Printable {
    static void showInfo() {
        System.out.println("This is a printable interface");
    }
}
```

• **private 메소드**: 인터페이스 내에서 private 메소드를 정의할 수 있습니다.

```
interface Printable {
    default void print() {
        prepare();
        System.out.println("Printing...");
    }
    private void prepare() {
        System.out.println("Preparing to print...");
    }
}
```

인터페이스

• **다중 인터페이스 구현**: 클래스는 여러 인터페이스를 동시에 구현할 수 있습니다.

```
class MyClass implements Interface1, Interface2 {
    // 인터페이스 구현
}
```

• **인터페이스 상속**: 인터페이스는 다른 인터페이스를 확장할 수 있습니다.

```
interface Printable {
    void print();
}
interface Displayable extends Printable {
    void display();
}
```

• **타입 변환**: 인터페이스를 사용하여 다형성을 구현할 수 있습니다.

```
Printable obj = new MyClass();
```

•**다형성**: 인터페이스를 통해 여러 구현체를 하나의 타입으로 다룰 수 있습니다.

인터페이스

- **객체 타입 확인**: instanceof 연산자를 사용하여 객체가 특정 인터페이스를 구현했는지 확인할 수 있습니다.

```
if (obj instanceof Printable) {    // obj는 Printable 인터페이스를 구현한 객체입니다.
}
```

8 중첩 선언과 익명 객체

• **중첩 클래스**: 클래스 내부에 선언된 클래스로, 외부 클래스의 멤버에 접근할 수 있습니다. 다음은 중첩 클래스의 예시입니다.

```
class Outer {
    class Inner {
        void display() {
            System.out.println("Inner class method");
        }
    }
}
```

• **인스턴스 멤버 클래스**: 외부 클래스의 인스턴스와 관련된 클래스입니다. 인스턴스 생성 시 생성됩니다.

```
class Outer {
    class Inner {        // 인스턴스 멤버 클래스
    }
}
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner(); // 인스턴스 생성 후 접근
```

8 중첩 선언과 익명 객체

• **정적 멤버 클래스**: 외부 클래스의 인스턴스와 관계없이 사용할 수 있는 클래스입니다. static 키워드로 선언됩니다.

```
class Outer {
    static class Nested {  // 정적 멤버 클래스
    }
}
Outer.Nested nested = new Outer.Nested (); // 직접 접근 가능
```

• **로컬 클래스**: 메소드나 초기화 블록 내부에 선언된 클래스입니다. 해당 블록 내에서만 유효합니다.

```
class Outer {
    void display() {
        class Local { // 로컬 클래스 }
    }
}
```

8 중첩 선언과 익명 객체

• **바깥 멤버 접근**: 중첩 클래스는 외부 클래스의 멤버에 직접 접근할 수 있습니다.

```
class Outer {
    private int num;
    class Inner {
        void display() {
            System.out.println(num); // 외부 클래스의 멤버에 접근
        }
    }
}
```

• **중첩 인터페이스**: 클래스 내부에 선언된 인터페이스입니다.

```
class Outer {
    interface InnerInterface { // 중첩 인터페이스
    }
}
```

8 중첩 선언과 익명 객체

• **익명 객체**: 이름이 없는 객체로, 클래스의 정의와 동시에 인스턴스를 생성합니다.

```
interface MyInterface {
     void display();
}
class MyClass {
    void show() {
        MyInterface obj = new MyInterface() {
            public void display() {
                System.out.println("Anonymous object");
            }
        };
        obj.display();
    }
}
```

9 라이브러리와 모듈

항목	설명
라이브러리(Library)	재사용 가능한 코드와 리소스의 모음으로, 특정 기능을 제공하거나 문제를 해결하기 위한 함수, 클래스, 메소드 등이 포함됩니다.

```
// java.util 패키지의 ArrayList 클래스를 사용하는 예시
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Hello");
        list.add("World");
        System.out.println(list);
    }
}
```

항목	설명
모듈(Module)	관련된 코드와 리소스의 모음으로, 라이브러리보다 더 큰 단위로, 응용 프로그램의 일부분을 구성하는 독립적인 단위입니다.

```
// 자바 9부터 사용 가능한 모듈 시스템의 예시
module mymodule {
    requires java.base;
    exports com.mypackage;
}
```

9 라이브러리와 모듈

항목	설명
응용프로그램 모듈화	모듈을 사용하여 응용 프로그램을 여러 개의 독립적인 단위로 분할하여 개발 및 관리를 용이하게 합니다.

```
// 모듈화된 응용 프로그램의 예시
module myapp {
    requires mymodule;
}
```

항목	설명
모듈 배포용 JAR 파일	자바 9부터는 모듈 경로를 포함하는 JAR 파일을 사용하여 모듈을 배포할 수 있습니다.

```
// 모듈 경로를 포함하는 JAR 파일의 예시
jar --create --file mymodule.jar --module-version 1.0 --main-class=com.mypackage.Main -C out .
```

- **jar**: JAR 파일을 생성하거나 관리하는 명령어입니다.
- **--create**: 새로운 JAR 파일을 생성합니다.
- **--file mymodule.jar**: 생성될 JAR 파일의 이름을 지정합니다.
- **--module-version 1.0**: JAR 파일에 포함될 모듈의 버전을 지정합니다.
- **--main-class=com.mypackage.Main**: JAR 파일이 실행될 때 진입점이 되는 메인 클래스를 지정합니다.
- **-C out .**: JAR 파일에 포함될 파일을 지정합니다.
  여기서는 현재 디렉토리의 **out** 디렉토리에 있는 모든 파일 및 디렉토리를 JAR 파일에 포함시킵니다.

이렇게 함으로써 **out** 디렉토리에 있는 파일들을 포함한 JAR 파일이 생성되며, 이 JAR 파일을 실행하면 **com.mypackage.Main** 클래스가 실행됩니다.

10 예외처리

예외와 예외 클래스
예외(Exception)는 프로그램 실행 중 발생할 수 있는 예기치 않은 상황을 나타냅니다. 자바에서는 예외를 객체로 표현하며, 예외 클래스(Exception class)를 사용하여 예외를 다룹니다.

```
public class ExceptionExample {
    public static void main(String[] args) {
        int result = divide(10, 0); // 0으로 나누는 예외 발생
        System.out.println("Result: " + result);
    }
    public static int divide(int num1, int num2) {
        return num1 / num2; // ArithmeticException 발생
    }
}
```

예외 처리 코드
예외를 처리하기 위해 try-catch 블록을 사용합니다.

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0); // 0으로 나누는 예외 발생
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
    public static int divide(int num1, int num2) { return num1 / num2;}
}
```

10 예외처리

리소스 자동 닫기: try-with-resources 구문을 사용하여 리소스를 자동으로 닫을 수 있습니다. 이를 통해 예외가 발생하더라도 리소스를 안전하게 닫을 수 있습니다.

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

public class Main {
    public static void main(String[] args) {
    // try-with-resources 문을 사용하여 파일 리소스를 자동으로 닫음
        try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {
            String line; while ((line = reader.readLine()) != null) {
                    System.out.println(line);
             }
         } catch (IOException e) {
             e.printStackTrace();
         }
     }
}
```

여기서 try 블록 다음에 괄호 안에 BufferedReader 객체를 생성하는 부분이 try-with-resources 구문입니다. 이렇게 하면 BufferedReader 객체가 자동으로 닫힙니다.

감사합니다