

야자수 음료를 마시는 방법을 \*\*클래스 메서드(static 메서드)\*\*와 \*\*인스턴스 메서드\*\*로  
비유하면 다음과 같습니다.

## 1. 클래스 메서드 (static 메서드)

→ "야자수 기계를 이용해서 음료를 마시는 방법"

- 야자수 기계를 사용하면 어떤 야자수 열매든 기계를 통해 주스를 만들 수 있음
- 특정 열매(객체)와 관계없이 기계(클래스) 자체가 동작
- 기계를 사용하려면 객체를 생성하지 않아도 (=인스턴스를 만들지 않아도) 사용 가능

💡 Java 코드 예시:

```
class CoconutJuice {  
  
    static void useMachine() {  
  
        System.out.println("야자수 기계를 사용하여 주스를 만듭니다.");  
  
    }  
  
}  
  
  
// 객체 생성 없이 클래스 이름으로 직접 호출  
  
CoconutJuice.useMachine();
```

---

## 2. 인스턴스 메서드

→ "야자수 열매에 직접 빨대를 꽂아서 마시는 방법"

- 특정 야자수 열매(객체)를 가져와야 함
- 각 열매마다 상태(크기, 당도 등)가 다를 수 있음
- 반드시 열매(객체)가 존재해야 사용할 수 있음

💡 Java 코드 예시:

```
class Coconut {  
  
    void drinkDirectly() {  
  
        System.out.println("야자수 열매에 직접 빨대를 꽂아 마십니다.");  
  
    }  
  
}
```

```
}
```

```
// 객체 생성 후 인스턴스 메서드 호출
```

```
Coconut myCoconut = new Coconut();
```

```
myCoconut.drinkDirectly();
```

---

### 🔥 결론

- **클래스 메서드 (static 메서드):** 특정 객체 없이 사용할 수 있음 (ex. 야자수 기계)
- **인스턴스 메서드:** 특정 객체(야자수 열매)를 생성한 후 사용할 수 있음 (ex. 빨대를 꽂아 마시는 방식)

## > 09. 인스턴스 메소드 넣기

인스턴스 메소드는 인스턴스를 통해서만 접근할 수 있고, 특정 인스턴스의 인스턴스 필드에 접근하기 위해서 사용한다.

이전에 클래스를 관련있는 데이터를 묶어 새로운 데이터를 만들어 놓은 것이라 이야기하였다.

클래스는 관련있는 데이터뿐 아니라 관련있는 메소드도 묶을 수 있다. 결과적으로 클래스는 관련있는 데이터와 메소드를 묶은 것이다.

메소드는 크게 클래스 메소드와 인스턴스 메소드가 있다.

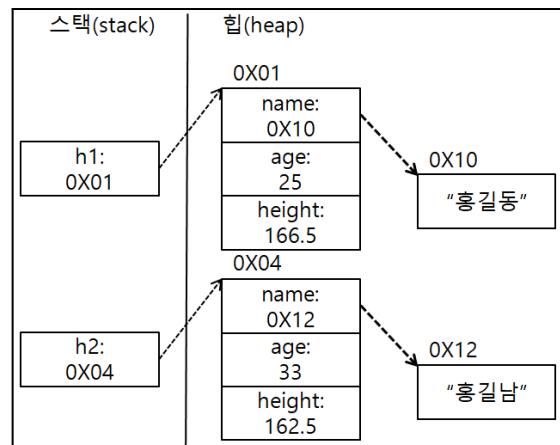
이전 챕터에서 설명한 클래스 메소드는 클래스 필드를 조작하는 용도로 사용하고 다음에 설명할 인스턴스 메소드는 인스턴스 필드를 조작하는 용도로 사용된다.

그동안 클래스를 데이터 형태로 만들어 사용하였다. 클래스안에 데이터를 저장하기 위해서 선언한 변수들을 인스턴스 필드라고 한다. 인스턴스 필드는 new 연산자로 할당 할때마다 저장공간이 생성되고 인스턴스를 통해서 여러 저장 공간중 하나의 저장공간에 접근 할 수 있다.

인스턴스 메소드는 인스턴스를 통해서 각각의 인스턴스 필드에 접근할 수 있다.

```
Human h1=new Human();
h1.name="홍길동";
h1.age=25;
h1.height=166.5;
System.out.println("name:"+h1.name);
System.out.println("age:"+h1.age);
System.out.println("height:"+h1.height);

Human h2=new Human();
h2.name="홍길남";
h2.age=33;
h2.height=162.5;
System.out.println("name:"+h2.name);
System.out.println("age:"+h2.age);
System.out.println("height:"+h2.height);
```



어느때 클래스 메소드와 인스턴스 메소드를 사용할 것인가?

둘의 공통점 관련 있는 클래스에서 사용한다.

클래스 필드를 조작할때는 클래스 메소드로 구현한다.

인스턴스 필드를 조작할때는 인스턴스 메소드로 구현한다.

인스턴스 필드를 사용할 일이 없다면 클래스메소드

인스턴스 필드를 사용할 일이 있다면 인스턴스 메소드

상위 코드를 가지고 클래스 필드와 인스턴스 필드로 데이터를 출력해보자.

1. 클래스 메소드 매개변수, string,int,double
2. 클래스 메소드 static 매개변수 Human
3. 인스턴스 메소드 매개변수 없음

클래스안에 static을 붙여서 생성한 전역 변수를 클래스 필드라고 한다. 클래스 필드 들은 클래스이름.필드이름을 통해서 모든 지역에서 접근 할 수 있는 전역 변수가 된다. 클래스 안에 static이 붙은 메소드를 클래스 메소드라 한다. 클래스 메소드는 전역 메소드로 클래스이름.메소드 형태로 모든 지역에서 접근 하여 실행 시킬 수 있는 메소드 이다.

인스턴스 메소드는 클래스의 인스턴스(객체)에 속하는 인스턴스 필드에 접근할 수 있는 메소드이다. 인스턴스.메소드 형태로 접근하면 메소드 안에서 해당 인스턴스의 필드에 접근해서 여러가지 작업을 할 수 있다. 클래스 메소드에서 클래스 필드에 접근 할 수 있지만 인스턴스 필드에 접근 할 수 없다.

Human클래스 안에 인스턴스 메소드를 넣어서 해당 인스턴스에 세팅된 인스턴스 필드의 값을 인스턴스 메소드를 이용해서 필요한 새로운 값을 생성 하고자 할 때 사용한다.

다음에 인스턴스 메소드를 추가하는 코드를 작성해 보자.

인스턴스 메소드 생성 방법은 클래스 메소드 생성 방법에서 static을 제거하면 된다.

다음 코드를 확인해 보고 인스턴스 메소드 사용방법을 익혀보자.

//Dog.java 파일

```
public class Dog {  
    // 인스턴스 필드 선언  
    String name;      int age;  
    // 생성자(Constructor) - 객체를 초기화하는 메소드  
    public Dog(String name, int age) {this.name = name;this.age = age;}  
    // 인스턴스 메소드 - 객체의 상태를 출력하는 메소드
```

```

public void displayInfo() {
    System.out.println("Dog's name: " + name);
    System.out.println("Dog's age: " + age + " years");
}
// 인스턴스 메소드 - 객체의 나이를 증가시키는 메소드
public void celebrateBirthday() {
    age++;
    System.out.println(name + " is " + age + " years old.");
}
// 다른 메소드들도 여기에 추가될 수 있음
}

//Main.java 파일

public class Main {
    public static void main(String[] args) {
        // 첫 번째 Dog 클래스의 인스턴스 생성
        Dog myDog = new Dog("Buddy", 3);
        // 인스턴스 메소드 호출
        System.out.println("== My Dog ==");
        myDog.displayInfo();
        myDog.celebrateBirthday();
        myDog.displayInfo();
        // 두 번째 Dog 클래스의 인스턴스 생성
        Dog yourDog = new Dog("Max", 2);
        // 인스턴스 메소드 호출
        System.out.println("\n== Your Dog ==");
        yourDog.displayInfo();
        yourDog.celebrateBirthday();
        yourDog.displayInfo();
        //강아지 이름이 happy, 나이는 4인 데이터를 가지는 Dog클래스의 인스턴스에서 메소드를
        //통해서 원하는 결과를 얻고자 한다면 인스턴스 메소드를 사용하면 된다.
    }
}
2.

public class Main {
    public static void main(String[] args) {
        // 20개의 Book 클래스의 인스턴스 생성 및 메소드 호출
        for (int i = 1; i <= 20; i++) {
            Book book = new Book("Title" + i, "Author" + i);
            System.out.println("== Book " + i + " ==");
            book.displayInfo();
            System.out.println();
        }
    }
}
class Book {
    // 인스턴스 필드 선언
    String title;    String author;
}

```

```

// 생성자(Constructor) - 객체를 초기화하는 메소드
public Book(String title, String author) {
    this.title = title;           this.author = author;
}
// 인스턴스 메소드 - 객체의 정보를 출력하는 메소드
public void displayInfo() {
    System.out.println("Book Title: " + title);
    System.out.println("Book Author: " + author);
}
}

```

다음을 설명해보자.

1. 클래스 메소드를 이용한 카운터 만들어 보기
  2. 인스턴스 메소드를 이용한 카운터 만들어 보기
  3. 카운터를 여러개 만들려면 왜 인스턴스로 만들어야 하나 생각해 보기
  4. 클래스 필드와 인스턴스 필드 이해해 보기
- 카운터 클래스를 통해서 클래스 메소드와 인스턴스 메소드를 만들어 보자.  
 클래스와 관련있고 인스턴스 필드와 관련이 없으면 클래스필드로 만든다.  
 클래스와 관련있고 인스턴스 필드 와 관련이 있으면 인스턴스 필드로 만든다.

```

public class Counter {
    // 클래스 변수 (모든 인스턴스가 공유)
    public static int totalCount = 0;

    // 인스턴스 변수 (각 인스턴스가 고유하게 가짐)
    private int count = 0;

    // 인스턴스 메소드: 각 인스턴스의 count를 1 증가
    public void incrementCount() {
        count++;
    }

    // 인스턴스 메소드: 각 인스턴스의 count를 1 감소
    public void decrementCount() {
        if (count > 0) {
            count--;
        }
    }

    // 인스턴스 메소드: 각 인스턴스의 count를 원하는 값으로 설정
    public void setCount(int count) {
        if (count >= 0) {
            this.count = count;
        }
    }

    // 인스턴스 메소드: 각 인스턴스의 count를 반환
}

```

```
public int getCount() {
    return count;
}

// 클래스 메소드: totalCount를 1 증가
public static void incrementtotalCount() {
    totalCount++;
}

public static void main(String[] args) {
    Counter c1 = new Counter();
    Counter c2 = new Counter();

    // 각 인스턴스의 count 조작
    c1.incrementCount();
    c2.incrementCount();
    c2.incrementCount();

    System.out.println("c1의 count:" + c1.getCount()); // c1 인스턴스의
count 출력
    System.out.println("c2의 count: " + c2.getCount()); // c2 인스턴스의
count 출력

    // totalCount 조작
    Counter.incrementtotalCount();
    Counter.incrementtotalCount();
    System.out.println("전체 count (증가 후): " + Counter.totalCount );

    // 각 인스턴스의 count 설정
    c1.setCount(5);
    c2.setCount(3);
    System.out.println("c1의 count (설정 후): " + c1.getCount());
    System.out.println("c2의 count (설정 후): " + c2.getCount());
}

}
```

```

3 public class Java0520_2 {
4     public static double height=20;
5     public static double width=40;
6     public static void area() {//넓이
7         System.out.println("넓이:"+height*width);
8     }
9     public static void periphery() {//둘레
10        System.out.println("둘레:"+ (height*2+width*2));
11    }
12    public static void main(String[] args) {
13        //직사각형 넓이와 둘레를 구하는 프로그램을 만들어 보자.
14        area();
15        periphery();
16    }
17}

```

왼쪽 코드는 전역변수를 이용하여 사각형의 넓이와 둘레를 구하는 프로그램이다. 확인해 보자.

이런 방식의 코드는 한개의 사각형에 대해 계산 결과를 얻고자 한다면 문제가 없지만

여러개의 사각형에 대해 결과를 얻고자 한다면 다음 코드처럼 각각의 사각형 관련 데이터와 메소드를 따로 만들어야 한다.

코드를 보면 비슷한 데이터와 메소드가 모여 있어서 유지 보수하기 불편하다.

이런 형태의 코드도 큰 문제는 없지만 클래스를 이용해서 관련 변수와 메소드를 묶으면 사용하기 편한 프로그램을 구현 할 수 있다.

상위 코드를 살펴보면 4번 라인 부터 11번 라인

```

3 public class Java0520_3 {
4     public static double height=20;
5     public static double width=40;
6     public static void area() {//넓이
7         System.out.println("넓이:"+height*width);
8     }
9     public static void periphery() {//둘레
10        System.out.println("둘레:"+ (height*2+width*2));
11    }
12    public static double height1=30;
13    public static double width1=40;
14    public static void area1() {//넓이
15        System.out.println("넓이:"+height1*width1);
16    }
17    public static void periphery1() {//둘레
18        System.out.println("둘레:"+ (height1*2+width1*2));
19    }
20    public static double height2=20;
21    public static double width2=20;
22    public static void area2() {//넓이
23        System.out.println("넓이:"+height2*width2);
24    }
25    public static void periphery2() {//둘레
26        System.out.println("둘레:"+ (height2*2+width2*2));
27    }
28    public static void main(String[] args) {
29        //직사각형 넓이와 둘레를 구하는 프로그램을 만들어 보자.
30        area();      periphery();
31        area1();    periphery1();
32        area2();    periphery2();
33    }
34}

```

까지가 하나의 사각형 관련 코드이고 이후 비슷한 코드가 반복 되는 것을 확인할 수 있다. 4~11라인에 있는 관련 있는 코드를 묶어서 사용할 수 있다면 코드를 관리하기 쉬워 질 것이다. 이런 이유에서 클래스를 사용하여 관련있는 필드와 메소드를 묶어서 프로그램 한다.

다음 Rectangle 클래스는 반복되는 형태의 코드를 없애기 위해 상위 코드에서 반복 되는 클래스 필드와 클래스 메소드를 static을 제외해서 인스턴스 변수와 인스턴스 메소드로 변경하였다.

인스턴스 필드로 변경한 이유는 new연산자로 할당 할때 마다 다른 크기의 사각형 height, width 저장을 저장 할 공간을 가지고 있어야 하기 때문이다. 인스턴스 필드는 new 할 때 마다 저장공간이 힙에 잡히지만 클래스 필드들은 프로그램이 시작될때 한번 메소드에 잡하고 이후 추가로 잡을 수 있는 방법이 없다.

인스턴스 필드로 height와 width를 두고 이를 필드를 조작하는 인스턴스 메소드 area periphery 메소드를 클래스 안에 넣어서 클래스로 만든 것이다.

```
3 class Rectangle{  
4     public double height=0;  
5     public double width=0;  
6     public void area() {//넓이  
7         System.out.println("넓이:"+height*width);  
8     }  
9     public void area(double height) {//넓이  
10        System.out.println("넓이:"+height*width);  
11    }  
12    public void periphery() {//둘레  
13        System.out.println("둘레:"+ (height*2+width*2));  
14    }  
15 }
```

이전 까지는 필드와 메소드에 모두 static을 넣어서 선언 하였는데 Rectangle클래스에서는 static를 제거하고 선언 하였다. 차이점은 static을 붙이면 클래스 멤버여서 new를 이용한 생성 과정 없이 전역에서 바로 접근 가능 하지만 static이 붙어 있지 않은 인스턴스 멤버는 new 연산자를 사용하여 생성된 인스턴스를 통해서만 접근할 수 있다.

결과적으로 클래스 멤버는 전역에서 접근 가능하고 선언과 동시에 사용할 수 있고, 인스턴스 멤버는 필요할때 마다 new연산자를 사용하여 필요할 때마다 여러개 생성하여

인스턴스 변수를 통해서 사용할 수 있다.

다음 이미지는 Rectangle 클래스로 사각형의 넓이와 둘레를 구현한 프로그램이다.

19번 라인을 보면 클래스의 변수를 선언하고 인스턴스를 생성하였다. r1이 클래스 변수 또는 인스턴스 변수에 해당한다.

```
16 public class Java0520_4 {  
17     public static void main(String[] args) {  
18         //직사각형 넓이와 둘레를 구하는 프로그램을 만들어 보자.  
19         Rectangle r1=new Rectangle();  
20         r1.height=40;    r1.width=40;  
21         r1.area();      r1.periphery();  
22         Rectangle r2=new Rectangle();  
23         r2.height=20;    r2.width=40;  
24         r2.area();      r2.periphery();  
25         Rectangle r3=new Rectangle();  
26         r3.height=20;    r3.width=20;  
27         r3.area();      r3.periphery();  
28     }  
29 }
```

20번 라인을 보면 생성된 인스턴스에 . 점을 찍어 해당 인스턴스 필드의 값을 설정하였다.

21번 라인을 보면 생성한 인스턴스에 .을 찍어 원하는 메소드를 실행 시켰다. 변수에 접근 하듯이 점을 찍어 메소드에 접근할 수 있다.

상위 코드들에서처럼 클래스 멤버를 이용하면 새로운 사각형의 넓이와 둘레를 구하고 싶을 때마다 클래스 필드와 클래스 메소드를 추가해야 하지만, 인스턴스 필드와 인스턴스 메소드를 이용하면 새로운 계산이 필요할 때마다 new 연산자를 사용하여 원하는 결과 값을 얻을 수 있다.

다음 예제로 클래스필드와 인스턴스 필드를 구분해 보자.

1.

```
public class Main {  
    public static void main(String[] args) {  
        // static 메소드 호출  
        double result1 = MathOperation.addStatic(5, 3);  
        System.out.println("Static Addition: " + result1);  
        // 인스턴스 메소드 호출  
        MathOperation mathInstance = new MathOperation();  
        double result2 = mathInstance.addInstance(5, 3);  
        System.out.println("Instance Addition: " + result2);  
    }  
}  
class MathOperation {  
    // static 메소드 - 클래스 레벨에서 호출 가능  
    public static double addStatic(double num1, double num2) {  
        return num1 + num2;  
    }  
    // 인스턴스 메소드 - 객체 생성 후에 호출 가능  
    public double addInstance(double num1, double num2) {  
        return num1 + num2;  
    }  
}
```

2.

```
public class Main {  
    public static void main(String[] args) {  
        // 정적(static) 메소드 호출  
        double interest = BankAccount.calculateInterest(1000, 0.05);  
        System.out.println("Interest (Static Method): $" + interest);  
        // 인스턴스 메소드 호출  
        BankAccount account1 = new BankAccount(1000);  
        double balanceAfterDeposit = account1.deposit(500);  
        System.out.println("Balance After Deposit (Instance Method): $" +  
balanceAfterDeposit);
```

```

        }
    }

class BankAccount {
    private double balance;
    // 생성자(Constructor) - 객체를 초기화하는 메소드
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
    // 정적(static) 메소드 - 클래스 레벨에서 호출 가능
    // 원금과 이자율을 받아서 이자를 계산하여 반환
    public static double calculateInterest(double principal, double rate) {
        return principal * rate;
    }
    // 인스턴스 메소드 - 객체 생성 후에 호출 가능
    // 입금을 수행하고 입금 후의 잔고를 반환
    public double deposit(double amount) {
        this.balance += amount;
        return this.balance;
    }
}

```

3.

```

public class Main {
    public static void main(String[] args) {
        // 정적(static) 메소드 호출
        double averageGrade = Student.calculateAverageGrade(90, 85, 92);
        System.out.println("Average Grade (Static Method): " + averageGrade);
        // 인스턴스 메소드 호출
        Student student1 = new Student("Alice");
        student1.addGrade(95);
        student1.addGrade(88);
        student1.addGrade(91);
        double student1Average = student1.calculateAverage();
        System.out.println(student1.getName()
            + "'s Average Grade (Instance Method): " + student1Average);
    }
}

class Student {
    private String name;
    private int totalGrades;
    private int numberofGrades;

    // 생성자(Constructor) - 객체를 초기화하는 메소드
    public Student(String name) {
        this.name = name;
        this.totalGrades = 0;
        this.numberofGrades = 0;
    }
}

```

```

// 정적(static) 메소드 - 클래스 레벨에서 호출 가능
// 세 과목의 성적을 받아서 평균을 계산하여 반환
public static double calculateAverageGrade(int grade1,int grade2,int grade3){
    return (grade1 + grade2 + grade3) / 3.0;
}
// 인스턴스 메소드 - 객체 생성 후에 호출 가능
// 성적을 추가하고 총점과 성적 개수를 업데이트
public void addGrade(int grade) {
    this.totalGrades += grade;
    this.numberOfGrades++;
}
// 인스턴스 메소드 - 객체의 평균 성적을 계산하여 반환
public double calculateAverage() {
    if (numberOfGrades == 0) {
        return 0; // 성적이 없으면 0을 반환하여 나누기 오류를 방지
    }
    return (double) totalGrades / numberOfGrades;
}
// 인스턴스 메소드 - 학생의 이름을 반환
public String getName() {
    return name;
}
}

```

4.

```

public class Main {
    public static void main(String[] args) {
        // 정적(static) 메소드 호출
        String carType = Car.getCarType();
        System.out.println("Car Type (Static Method): " + carType);
        // 인스턴스 메소드 호출
        Car car1 = new Car("Sedan", "Blue");
        car1.displayCarInfo();
    }
}
class Car {
    private static String carType = "Compact";
    private String color;
    // 생성자(Constructor) - 객체를 초기화하는 메소드
    public Car(String type, String color) {
        carType = type; // 정적(static) 변수를 생성
        this.color = color;
    }
    // 정적(static) 메소드 - 클래스 레벨에서 호출 가능
    // 자동차의 유형을 반환
    public static String getCarType() {
        return carType;
    }
}

```

```

// 인스턴스 메소드 - 객체 생성 후에 호출 가능
// 자동차의 정보를 출력
public void displayCarInfo() {
    System.out.println("Car Type (Instance Method): " + carType);
    System.out.println("Car Color: " + color);
}
}

5.

public class Main {
    public static void main(String[] args) {
        // 정적(static) 메소드 호출
        int totalPlayers = BaseballPlayer.getTotalPlayers();
        System.out.println("Total Baseball Players (Static Method): " +
totalPlayers);
        // 인스턴스 메소드 호출
        BaseballPlayer player1 = new BaseballPlayer("Mike Trout",
"Outfielder");
        player1.displayPlayerInfo();
    }
}

class BaseballPlayer {
    private static int totalPlayersCount = 0;
    private String name;
    private String position;
    // 생성자(Constructor) - 객체를 초기화하는 메소드
    public BaseballPlayer(String name, String position) {
        this.name = name;
        this.position = position;
        totalPlayersCount++;
    }
    // 정적(static) 메소드 - 클래스 레벨에서 호출 가능
    // 생성된 총 야구 선수 수를 반환
    public static int getTotalPlayers() {
        return totalPlayersCount;
    }
    // 인스턴스 메소드 - 객체 생성 후에 호출 가능
    // 야구 선수의 정보를 출력
    public void displayPlayerInfo() {
        System.out.println("Player Name: " + name);
        System.out.println("Position: " + position);
    }
}

```

## 다양한 클래스 예제

아래는 6가지 구성 요소(클래스 필드, 인스턴스 필드, 생성자, 클래스 메서드, 인스턴스 메서드, 정적 블록)를 사용하여 학생(Student), 자동차(Car), 책(Book), 학교(School),

\*\*직원(Employee)\*\*를 표현한 예제 5개입니다.

---

### 1. 학생(Student) 예제

```
public class Student {  
    // 클래스 필드  
    static int totalStudents = 0;  
  
    // 인스턴스 필드  
    String name;  
    int age;  
  
    // 정적 블록  
    static {  
        System.out.println("학생 클래스가 로드되었습니다!");  
    }  
  
    // 생성자  
    public Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
        totalStudents++;  
    }  
  
    // 클래스 메서드  
    public static void printTotalStudents() {  
        System.out.println("전체 학생 수: " + totalStudents);  
    }  
  
    // 인스턴스 메서드  
    public void introduce() {
```

```
        System.out.println("이름: " + name + ", 나이: " + age);  
    }  
  
    public static void main(String[] args) {  
        System.out.println("== 학생 객체 생성 ==");  
        Student student1 = new Student("Alice", 20);  
        Student student2 = new Student("Bob", 22);  
  
        student1.introduce();  
        student2.introduce();  
  
        System.out.println("== 전체 학생 수 ==");  
        Student.printTotalStudents();  
    }  
}
```

---

## 2. 자동차(Car) 예제

```
public class Car {  
  
    // 클래스 필드  
    static int totalCars = 0;  
  
    // 인스턴스 필드  
    String model;  
    String color;  
  
    // 정적 블록  
    static {  
        System.out.println("자동차 클래스가 로드되었습니다!");  
    }  
}
```

```
// 생성자

public Car(String model, String color) {
    this.model = model;
    this.color = color;
    totalCars++;
}

// 클래스 메서드

public static void printTotalCars() {
    System.out.println("전체 자동차 수: " + totalCars);
}

// 인스턴스 메서드

public void showDetails() {
    System.out.println("모델: " + model + ", 색상: " + color);
}

public static void main(String[] args) {
    System.out.println("==== 자동차 객체 생성 ====");
    Car car1 = new Car("Tesla Model S", "Red");
    Car car2 = new Car("BMW X5", "Black");

    car1.showDetails();
    car2.showDetails();

    System.out.println("==== 전체 자동차 수 ====");
    Car.printTotalCars();
}
```

---

### 3. 책(Book) 예제

```
public class Book {  
    // 클래스 필드  
  
    static int totalBooks = 0;  
  
    // 인스턴스 필드  
  
    String title;  
    String author;  
  
    // 정적 블록  
  
    static {  
        System.out.println("책 클래스가 로드되었습니다!");  
    }  
  
    // 생성자  
  
    public Book(String title, String author) {  
        this.title = title;  
        this.author = author;  
        totalBooks++;  
    }  
  
    // 클래스 메서드  
  
    public static void printTotalBooks() {  
        System.out.println("전체 책 수: " + totalBooks);  
    }  
  
    // 인스턴스 메서드  
  
    public void displayInfo() {  
        System.out.println("제목: " + title + ", 저자: " + author);  
    }  
}
```

```
public static void main(String[] args) {  
    System.out.println("== 책 객체 생성 ==");  
    Book book1 = new Book("Java Programming", "James Gosling");  
    Book book2 = new Book("Effective Java", "Joshua Bloch");  
  
    book1.displayInfo();  
    book2.displayInfo();  
  
    System.out.println("== 전체 책 수 ==");  
    Book.printTotalBooks();  
}  
}
```

---

#### 4. 학교(School) 예제

```
public class School {  
    // 클래스 필드  
    static int totalSchools = 0;  
  
    // 인스턴스 필드  
    String name;  
    String location;  
  
    // 정적 블록  
    static {  
        System.out.println("학교 클래스가 로드되었습니다!");  
    }  
  
    // 생성자  
    public School(String name, String location) {
```

```
    this.name = name;
    this.location = location;
    totalSchools++;
}

// 클래스 메서드

public static void printTotalSchools() {
    System.out.println("전체 학교 수: " + totalSchools);
}

// 인스턴스 메서드

public void showDetails() {
    System.out.println("학교 이름: " + name + ", 위치: " + location);
}

public static void main(String[] args) {
    System.out.println("== 학교 객체 생성 ==");

    School school1 = new School("Green High School", "Seoul");
    School school2 = new School("Blue Middle School", "Busan");

    school1.showDetails();
    school2.showDetails();

    System.out.println("== 전체 학교 수 ==");
    School.printTotalSchools();
}
}
```

---

## 5. 직원(Employee) 예제

```
java
```

복사편집

```
public class Employee {  
    // 클래스 필드  
    static int totalEmployees = 0;  
  
    // 인스턴스 필드  
    String name;  
    String department;  
  
    // 정적 블록  
    static {  
        System.out.println("직원 클래스가 로드되었습니다!");  
    }  
  
    // 생성자  
    public Employee(String name, String department) {  
        this.name = name;  
        this.department = department;  
        totalEmployees++;  
    }  
  
    // 클래스 메서드  
    public static void printTotalEmployees() {  
        System.out.println("전체 직원 수: " + totalEmployees);  
    }  
  
    // 인스턴스 메서드  
    public void showInfo() {  
        System.out.println("직원 이름: " + name + ", 부서: " + department);  
    }  
}
```

```

}

public static void main(String[] args) {
    System.out.println("== 직원 객체 생성 ==");
    Employee emp1 = new Employee("John", "HR");
    Employee emp2 = new Employee("Emma", "IT");

    emp1.showInfo();
    emp2.showInfo();

    System.out.println("== 전체 직원 수 ==");
    Employee.printTotalEmployees();
}

}

```

다음 문제들을 클래스로 풀어보자.

1. 삼각형, 원 클래스를 만들어 넓이 둘레를 구하는 프로그램을 만들어보자.
2. 국,영,수과목 점수를 저장하고 총점과 평균을 출력하는 Student 클래스를 만들어 사용해보자.
3. 다음 이미지 클래스 처럼 동작하는 Car과 CellPhone 클래스를 만들어 보자.

```

Car c1=new Car("소나타");
c1.statement(); //현재 차종: --- 속력은 ---입니다.
c1.speedUp(); //차의 속도를 10 높임
c1.speedUp(); //차의 속도를 10 높임
c1.statement(); //현재 차종: --- 속력은 ---입니다.
c1.speedDown(); //차의 속도를 10 낮춤
c1.statement(); //현재 차종: --- 속력은 ---입니다.

```

```

CellPhone phone=new CellPhone("홍길동", "010-1111-1111");
phone.sendInput("홍길남", "010-2222-2222", "안녕");
phone.sendMsgButton();
phone.sendInput("홍길남", "010-2222-2222");
phone.sendButton();

```

4. 이전에 만든 은행 프로그램은 하나의 은행을 관리 할 수 있는 형태이다 여러개의 은행을 관리 할 수 있는 Bank클래스로 만들어 보자.

```
2 class Student{  
3     public double kor=0;public double eng=0;public double math=0;  
4     public Student(double kor, double eng, double math) {  
5         this.kor = kor; this.eng = eng; this.math = math;  
6     }  
7     //총점 평균  
8     public double totalSum() {  
9         return kor+eng+math;  
10    }  
11    public double avg() {  
12        return totalSum()/3;  
13    }  
14 }  
15 public class Java052403 {  
16     public static void main(String[] args) {  
17         Student st=new Student(93,85,66);  
18         System.out.println(st.avg());  
19     }  
20 }
```

```
2 class Car{  
3     public String carKind="";  
4     public int speed=0;  
5     public Car(String carKind) {  
6         this.carKind = carKind;  
7     }  
8     public void statement() {  
9         System.out.println(String.format(  
10             "현재 차종:%s 속력은 %d입니다.",this.carKind,this.speed));  
11     }  
12     public void speedUp() {  
13         this.speed=this.speed+10;  
14     }  
15     public void speedDown() {  
16         this.speed=this.speed-10;  
17     }  
18 }  
19 public class Java52404 {  
20     public static void main(String[] args) {  
21         Car c1=new Car("소나타");  
22         c1.statement();//현재 차종:--- 속력은 ---입니다.  
23         c1.speedUp();//차의 속도를 10 높임  
24         c1.speedUp();//차의 속도를 10 높임  
25         c1.statement();//현재 차종:--- 속력은 ---입니다.  
26         c1.speedDown();//차의 속도를 10 낮춤  
27         c1.statement();//현재 차종:--- 속력은 ---입니다.  
28     }  
29 }
```

```

2 class CellPhone{
3     public String masterName="";
4     public String masterPhoneNumber="";
5     public String sendName="";
6     public String sendPhoneNumber="";
7     public String message="";
8     public CellPhone(String masterName, String masterPhoneNumber) {
9         this.masterName=masterName;
10        this.masterPhoneNumber=masterPhoneNumber;
11    }
12    public void sendInput(String sendName, String sendPhoneNumber, String message){
13        this.sendName=sendName; this.sendPhoneNumber=sendPhoneNumber;
14        this.message=message;
15    }
16    public void sendMsgButton() {
17        System.out.println(
18            String.format("%s(%s)님이 %s(%s)님에게 '%s' 이라는 메시지를 보냈습니다.", 
19            masterName, masterPhoneNumber, sendName, sendPhoneNumber, message)
20        );
21    }
22    public void sendInput(String sendName, String sendPhoneNumber) {
23        this.sendName=sendName; this.sendPhoneNumber=sendPhoneNumber;
24    }
25    public void sendButton() {
26        System.out.println(
27            String.format("%s(%s)님이 %s(%s)님에게 전화를 겁니다.", 
28            masterName, masterPhoneNumber, sendName, sendPhoneNumber)
29        );
30    }

```

```

30 public class Java52405 {
31     public static void main(String[] args) {
32         CellPhone phone=new CellPhone("홍길동", "010-1111-1111");
33         phone.sendInput("홍길남", "010-2222-2222", "안녕");
34         phone.sendMsgButton();
35         phone.sendInput("홍길남", "010-2222-2222");
36         phone.sendButton();
37     }
38 }

```

1. 여러개의 은행을 관리하는 프로그램을 만들어 보자.

# 05. 클래스(class)

---

## > 01. 클래스

---

클래스는 관련있는 코드들을 묶어서 프로그램 안에서 객체 처럼 사용하고자 할 때 이용한다.

클래스는 제품을 만들기 위한 설계도(template)와 같습니다. 인스턴스는 설계도를 가지고 만든 실제 제품들과 같다. 제품이 필요하면 설계도를 이용해 제품을 만들어 사용한다. 제품이 인스턴스 설계도가 클래스에 해당한다. 인스턴스는 클래스를 `new`를 이용해 메모리 힙에 생성되어 사용할 수 있는 상태를 의미합니다.

제품이 필요하면 설계도로 원하는 만큼 제품을 만들어 사용하는 것 처럼, 프로그램에서 데이터가 필요하면 클래스로 선언한 다음 원하는 만큼 인스턴스를 생성해서 사용할 수 있다.

클래스는 인스턴스 필드(fields), 인스턴스 메소드(method), 생성자, 클래스 필드, 클래스 메소드, 정적블록을 가지고 있습니다.

**인스턴스 필드(Fields):** 클래스의 각 인스턴스에 저장하는 변수입니다. 인스턴스마다 다른 저장 공간에 값을 가진다.

**생성자(Constructor):** 객체를 만들 때 호출되는 특별한 메서드입니다. 클래스 내부에 생성자를 정의하면 인스턴스 생성 시 인스턴스 필드 초기화 작업을 수행할 수 있습니다.

**인스턴스 메서드(Methods):** 인스턴스 메서드는 해당 클래스의 인스턴스 필드를 조작하거나 다른 인스턴스 메서드를 호출하거나 클래스 메소드를 호출하는 작업을 한다.

**접근 제어자(Access Modifiers):** 클래스의 필드와 메소드 앞에 접근 권한을 지정하여 다른 곳에서 접근 여부를 정할 수 있다. 자바에서는 `public`, `private`, `protected`, `default`와 같은 접근 제어자를 제공합니다.

`public`은 프로그램 모든 지역에서 사용할 수 있고, `protected`는 상속된 클래스에서만 사용할 수 있고, `private`는 선언된 클래스 내부에서만 접근할 수 있고 외부에서는 접근 할 수 없고, `default`는 같은 패키지에서만 접근 가능하다. `public`과 `private` 2개만 사용하고 나머지는 사용할 일이 없으니 나중에 확인 해보자.

`this` : 해당 클래스의 여러 인스턴스 중 자기 자신의 인스턴스 주소를 의미한다. `this`.필드로 기술하면 여러 인스턴스 중 자기 자신의 인스턴스 필드에 접근할 수 있다.

`super` : 상속관계에서 부모 인스턴스의 주소를 의미한다.

`static` : `static` 키워드는 클래스 멤버를 정의할 때 사용된다. 클래스 멤버는 클래스 필드, 클래스 메소드가 있다.

**클래스 필드 (Static Fields)**: 클래스 수준에 선언되는 필드로서, 다른 클래스 메소드나 인스턴스 메소드가 공유하는 필드입니다. 객체 인스턴스를 생성하지 않고도 클래스 이름을 통해 접근 가능하고 전역의 의미를 가진다. 모든 지역에서 접근 가능한 전역의 의미를 가진다.

**클래스 메서드 (Static Methods)**: 객체 인스턴스를 생성하지 않고도 클래스 이름을 통해 직접 호출할 수 있는 메서드입니다. 모든 지역에서 접근 가능한 전역의 의미를 가진다. 인스턴스를 생성하지 않고 접근할 수 있는 메소드 여서 다른 인스턴스 필드, 인스턴스 메소드는 접근할 수 없고 다른 클래스 필드, 클래스 메소드를 접근 할 수 있다.

**정적 블록 (Static Blocks)**: 클래스 필드를 초기화하는 용도로 사용된다.

다음 예제들을 하나씩 확인해 보자.

```
public class Person {
    // 인스턴스 필드 선언
    String name;
    int age;
    // 생성자 선언
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // 인스턴스 메소드 선언
    public void printInfo() {
        System.out.println("이름: " + this.name + ", 나이: " + this.age);
    }
}
public class Main {
    public static void main(String[] args) {
        //main
        Person person = new Person("John", 30); //생성자를 이용해 필드를 초기화
        //다음 힙에 생성한후 person에 힙에 생성된 주소를 담았다.
        person.printInfo(); //주소 연산자를 이용하여 person인스턴스의
        printInfo메소드
        Person p1=new Person("p1",29);
        Person p2=new Person("p2",23);
        //printInfo메소드에서 this로 선언된 필드들이 어떤 값으로 출력 되는지 다음
        //예제로 확인해 보자.
        //p1인스턴스에서 메소드를 호출하므로 p1이 가지고 있는 필드 정보가 출력된다.
        p1.printInfo();
        //p2인스턴스에서 메소드를 호출하므로 p2가 가지고 있는 필드 정보가 출력된다.
```

```
    p2.printInfo();
}
}
```

2.

```
class BankAccount {
    private String accountNumber;
    private String accountOwner;
    private double balance;

    public BankAccount(String accountNumber, String accountOwner, double balance) {
        this.accountNumber = accountNumber;
        this.accountOwner = accountOwner;
        this.balance = balance;
    }
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Insufficient balance");
        }
    }
    public void printInfo() {
        System.out.println("Account number: " + accountNumber);
        System.out.println("Account owner: " + accountOwner);
        System.out.println("Balance: " + balance);
    }
}
//main
public class Main {
    public static void main(String[] args) {
        //BankAccount account = new BankAccount(); //기본생성자가 없어서 에러
        BankAccount account = new BankAccount("123-456-789", "John", 1000.0);
        account.deposit(500.0);
        account.withdraw(200.0);
        account.printInfo();
    }
}
```

3. `static`으로 선언 하면 모든지역에서 접근 가능하다.

```
// Car.java
```

```
public class Car {  
    // 인스턴스 변수  
    public String brand;  
    public String model;  
    public int speed;  
  
    // 클래스 변수  
    public static int totalCars = 0;  
  
    // 생성자  
    public Car(String brand, String model) {  
        this.brand = brand;  
        this.model = model;  
        this.speed = 0;  
        totalCars++;  
    }  
  
    // 인스턴스 메소드: 가속  
    public void accelerate(int speedIncrease) {  
        speed += speedIncrease;  
        System.out.println(brand + " " + model + " is accelerating. Current  
speed: " + speed + " km/h");  
    }  
  
    // 인스턴스 메소드: 정지  
    public void brake() {  
        speed = 0;  
        System.out.println(brand + " " + model + " has stopped.");  
    }  
  
    // 클래스 메소드: 전체 자동차 수 조회  
    public static void showTotalCars() {  
        System.out.println("Total number of cars: " + totalCars);  
    }  
}  
  
// Main.java  
public class Main {  
    public static void main(String[] args) {  
        // 객체 생성  
        Car car1 = new Car("Toyota", "Camry");  
        Car car2 = new Car("Honda", "Civic");  
  
        // 인스턴스 메소드 호출  
        car1.accelerate(50);  
        car2.accelerate(40);
```

```

        car1.brake();

        // 클래스 메소드 호출
        Car.showTotalCars(); // 출력: Total number of cars: 2
    }
}

```

4. 클래스 필드를 정적 블럭을 이용해서 초기화 할 수 있다.

```

// MainClass.java
public class MainClass {
    public static void main(String[] args) {
        // 다른 클래스의 메서드 호출
        StaticBlockExample.runMainFunction();
    }
}

// StaticBlockExample.java
public class StaticBlockExample {
    // 정적 필드
    private static int staticField;

    static {
        // 정적 블럭에서 필드 초기화
        staticField = initializeStaticField();
        System.out.println("Static block executed. staticField is set to " +
staticField);
    }

    public static void runMainFunction() {
        // 다른 코드 작성
        System.out.println("Main method: Accessing staticField, value is " +
staticField);
    }

    private static int initializeStaticField() {
        // 필드 초기화를 위한 복잡한 로직이 있다고 가정
        return 42;
    }
}

```

프로그램 변화를 살펴보면 과거 언어 이전에는 메소드가 없는 형태로 프로그램을 구현하다가 함수 기반 언어가 나오면서 메소드를 이용한 형태로 프로그램을 구현 하였다. 이후 객체지향 언어들이 나오면서 필드와 메소드를 포함한 클래스를 사용한 객체 형태 프로그램을 구현하게 되었다.

클래스는 관련있는 프로그램을 묶어 놓은 것이라고 하였다. 이전 챕터에서 클래스 없이 프로그램을 구현 하였고 특별히 클래스가 없어도 프로그램 구현하는데 문제가 없다.

클래스가 어렵다면 이전처럼 클래스 없이 구현한 다음 클래스로 업그레이드 해보자. 나중에는 클래스가 쉽고 편해질 것이다. 차후 개발 할 때 클래스는 지속적으로 사용되니 관련 용어들을 확인해 보자.

**클래스:** 관련있는 필드(변수)와 메소드(함수)를 묶어서 객체 형태 프로그램을 만들 때 사용하는 프로그램 문법을 `class`(클래스)라 한다. 클래스는 현실 세계의 객체를 프로그램에서 형상화 한것이다. 여기서 형상화는 사용할 수 있도록 표현했다는 이야기이다.

**객체:** 현실 세계에 존재하는 모든 것 사람, 머리, 다리, 팔, 몸통, 눈, 코, 입 등등 식별 가능 하다면 모두 객체에 해당 된다. 객체가 모여서 또 다른 객체가 만들어 지기도 한다. 프로그램에서 인스턴스와 같은 의미로 사용되기도 한다.

**인스턴스:** 클래스를 이용한 프로그램에서 실제 데이터 저장 공간을 메모리에 할당 받아 프로그램에서 사용할 수 있는 상태를 의미한다. 클래스를 사용할 수 있도록 메모리에 저장 공간을 생성하는 과정이고 이때 `new`연산자를 사용한다. `new`로 할당된 클래스 변수를 인스턴스라 부른다.

클래스를 사용하려면 프로그래머가 원하는 형태로 클래스를 선언한 다음 필요할때 생성해서 사용한다. 따라서, 클래스 선언은 한번만 하면되고 생성은 필요할때 마다 `new` 연산자를 사용하여 여러개 생성할 수 있다. 이때 생성된 클래스 변수를 인스턴스라 한다.

보통 클래스 설명할 때 봉어빵틀, 봉어빵 이야기를 많이 한다. 클래스 선언은 봉어빵을 어떤 모양으로 만들것인지 결정하는 봉어 빵틀 처럼 1개만 있으면 되고, 클래스 선언을 이용해 원할 때 언제든 여러개의 인스턴스를 생성 하듯이 봉어빵틀을 이용해 원할 때 언제든지 여러개의 봉어빵을 만들 수 있다. 봉어빵 틀은 클래스 선언, 봉어빵은 인스턴스에 해당한다.

필요한 물품의 설계도를 가지고 필요할때 설계도에 따라 물건을 만들 듯이 클래스 선언부를 가지고 필요할때 마다 인스턴스를 생성하여 사용한다.

`Circle c=new Circle();` 여기서 `c`를 인스턴스 또는 클래스 변수라 하고 `new`를 통해 힙에 생성된 클래스의 저장 공간을 인스턴스라 하며 인스턴스 `c`에는 인스터스 주소가 들어 있다. `c`를 일반적으로 인스턴스, 클래스 변수, 인스턴스 변수, 객체라 한다. 인스턴스를 만드는 과정을 인스턴스화라 한다.

다음 클래스 구성 요소를 확인해 보자.

클래스 구성 요소는 필드, 생성자, 메소드로 구성되어 있다. 필드는 클래스 필드와 인스턴스 필드로 구성되어 있다. 생성자는 인스턴스 필드를 초기화 하는 용도로 사용되고, 정적 블록 클래스 필드를 초기화 하는 용도로 사용 된다. 메소드는 객체 생성 없이 클래스 이름으로 모든 지역에서 직접 접근하여 사용 할 수 있는 클래스 메소드와 인스턴스를 통해서만 접근 할 수 있는 인스턴스 메소드로 구성되어 있다.

```

class Circle{//클래스 구성요소
    //필드
        //클래스필드    //static이 붙어 있음, 전역에서 사용가능 , 한개만생성
        //인스턴스필드 //static이 없음, 필요할때 new연산자를 사용 힘에 생성, 여러개 생성
    //생성자
        //정적 블록    //static{로직}    클래스필드의 값을 초기화하는데 사용
        //생성자        //public Circle(){} 인스턴스필드의 값을 초기화하는데 사용
    //메소드
        //클래스메소드   //클래스필드를 조작
        //인스턴스메소드 //인스턴스 필드를 조작
}.

```

전역이란? 프로그램을 사용할 수 있는 모든 지역을 의미한다. 클래스 필드와 클래스 메소드는 모든 지역에서 접근 가능하다.

클래스 메소드에서는 클래스 필드와 클래스 메소드만 접근 가능하고 인스턴스 메소드에서는 클래스 필드와 클래스 메소드, 해당 인스턴스 필드와 인스턴스 메소드에 접근할 수 있다.

```

23 class Circle{
24     //클래스 필드
25     private static int totalCount=0;
26     public static final double PI=3.14;
27     //인스턴스필드
28     public int serialNumber=0;
29     public double r=0;
30     //정적블럭
31     static {
32         Circle.totalCount=0;
33         //this.r=5;      //인스턴스 필드에 접근할 수 없다.
34         //Circle.PI=3.141592; //final변수를 변경할 수 없다.
35     }
36     //생성자
37     public Circle() { this(5); }
38     public Circle(int r) {
39         this.r=r; Circle.totalCount++;
40         createSN(); //시리얼넘버생성
41     }
42     //클래스 메소드
43     public static int getTotalCount() {
44         return Circle.totalCount; //전체 생성된 원의 개수
45     }
46     //인스턴스 메소드
47     public double area() {
48         return 2*this.r*Circle.PI; //원의 둘레를 구하는 메소드
49     }
50     //인스턴스 메소드
51     public int getSerialNumber() { //시리얼넘버얻기
52         return this.serialNumber;
53     }
54     private void createSN() { //시리얼넘버생성
55         this.serialNumber=Circle.totalCount;
56     }
57 }

```

다음 클래스는 생성된 원을 관리하는 Circle 클래스이다.

25, 26번이 클래스 필드이고 28, 29번이 인스턴스 필드이다.

37~41번은 생성자들이다. 생성자 중에서도 37~41 번을 기본 생성자라 한다.

43~45번이 클래스 메소드이고 나머지들이 인스턴스 메소드이다.

필드들은 선언과 함께 변수처럼 초기값을 할당 할 수 있다.

필드명을 만들때에는 변수명 짓는 방법과 동일하고 소문자로 시작해서 새로운 의미의 단어가 추가 될때 마다 첫자를 대문자로나마지 문자는 소문자로 기술

한다.

필드는 일반 변수와 비슷하고 해당 클래스 관련 데이터를 저장하는데 사용한다. 필드는 크게 클래스 필드와 인스턴스 필드로 구성되어 있다. 인스턴스가 여러개 생성될 때마다 새로운 다른 저장 공간을 생성하고 싶다면 인스턴스 필드를 사용하고, 인스턴스를 여러개 생성하여도 저장 공간이 1개만 필요하다면 클래스 필드를 사용한다. 인스턴스 필드는 인스턴스 변수를 통해서, 클래스 필드는 클래스 이름을 통해서 접근할 수 있다.

필드값을 초기화하지 않으면 숫자관련자료형은 0, boolean자료형은 false, 참조자료형은 null이 들어 있다.

다음 예를 통해서 인스턴스 필드와 클래스 필드의 사용 용도를 확인해 보자.

모든 타이어 제품은 1만 키로 이상 운행하면 타이어를 교체하여야 한다. 타이어 교체 여부를 확인하는 프로그램을 구현한다고 할 때 필요한 정보는 다음과 같다. 타이어의 교체 기준 거리와 각 타이어별 실제 주행거리가 필요하다. 타이어의 운행 기준 거리는 모든 타이어가 동일 하므로 타이어마다 따로 기록할 필요없이 한번만 기술하면 되므로 클래스 필드로 선언하고 제품별 실제 주행 거리는 타이어마다 기술하여야 하기 때문에 인스턴스 필드로 구현하여야 한다.

클래스 필드와 인스턴스 필드 선언 방법은 다음과 같다.

클래스 필드 선언 방법	접근 제한자 static 자료형 필드명
인스턴스 필드 선언 방법	접근 제한자 자료형 필드명

다음 사용 방법을 확인해 보자.

클래스 필드는 클래스이름.접근하고자하는필드로 다음과 같이 기술한다. Circle.PI  
인스턴스 필드의 경우는 인스턴스에 .(점)을 찍어서 접근한다. 인스턴스 이름이 c1이라면 c1.r, 이름이 c2라면 c2.r이라고 접근한다. 프로그램 내부에서 클래스 필드는 1개만 생성되고 인스턴스 필드는 인스턴스 개수 만큼 생성된다.

클래스 필드와 클래스 메소드의 사용 범위는 자신의 클래스 생성자, 정적블록, 클래스 메소드, 인스턴스 메소드 뿐만 아니라 다른 클래스의 클래스 생성자, 정적블록, 클래스 메소드, 인스턴스 메소드에서 사용 가능하다. 즉, 코드 작성이 가능한 모든 지역에서 사용 가능하여 전역 필드, 전역 메소드라 하기도 하고, static이 붙어 있어서 static 필드 static 메소드라 하기도 한다.

인스턴스 필드는 해당 인스턴스를 통해서만 접근 가능하다.

접근 제한자는 생성자, 메소드, 필드 등의 사용 범위를 결정하는데 사용한다.

접근 제한자는 대부분의 클래스 요소들의 맨 첫부분에 붙어서 사용 되는데 지금 까지 대부분 public으로 선언하여 제한 없이 필드나 메소드를 접근하여 사용 하였지만 접근 제한자를 사용하여 사용 범위를 제한 할 수 있다.

접근 제한자 종류는 public, private, protected, default(아무것도 기술하지 않은 경우)가 존재하며 public은 해당 멤버가 모든 지역에서 접근 가능하다는 이야기이고 private는 해당 클래스에서만 사용 할 수 있다는 이야기이다. 나머지 protected와 default는 복잡한 내용이 있는데 실질적으로 사용할 일이 없다. 관심이 있다면 웹을 통해서 공부해 보고 protected 나 default는 public이나 private로 변경하여 사용하자. 간단히 설명해 보자면 protected는 자기 자신과 상속한 자식 클래스에서만 사용할 수 있고, default는 같은 패키지 안에서만 사용할 수 있다.

public과 private만 사용하자.

클래스 내부에서는 public과 private로 선언된 멤버는 아무런 제약 없이 사용 할 수 있다. 반면 클래스 내부가 아닌 외부에서 사용하면 접근 제한자에 따라 접근 여부가 달라진다. 여기서 내부라 함은 해당 클래스에 있는 클래스 메소드, 인스턴스 메소드를 의미한다. 외부라 함은 다른 클래스에 있는 클래스 메소드, 인스턴스 메소드를 의미한다.

public이면 어디서든 접근 가능하고 private이면 외부에서 접근 불가하다.

```
// Car.java
public class Car {
    // private 멤버 변수
    private String engineType;
    // public 필드
    public String modelName;
    // public 메서드
    public void startEngine() {
        System.out.println("Engine started. Engine type: " + engineType);
    }
    // 내부에서만 접근 가능한 메서드
    private void setEngineType(String type) {
        this.engineType = type;
    }
    // 내부에서만 접근 가능한 메서드를 호출하는 예제
    public void setCarEngineType(String type) {
        setEngineType(type);
        System.out.println("Engine type set internally to: " + engineType);
    }
    // 외부에서는 접근할 수 없는 private 메서드
    private void internalMethod() {
        System.out.println("This is an internal method.");
    }
    // 외부에서는 호출 가능한 public 메서드
    public void publicMethod() {
```

```

        System.out.println("This is a public method.");
        internalMethod(); // 내부에서만 호출 가능한 private 메서드 호출
    }
}

// MainClass.java
public class MainClass {
    public static void main(String[] args) {
        // Car 클래스의 인스턴스 생성
        Car myCar = new Car();
        // 외부에서 public 필드에 직접 접근하여 값 설정
        myCar.modelName = "Sedan"; // 외부에서 public 필드에 직접 접근 가능
        // 외부에서 public 필드에 직접 접근하여 값 출력
        System.out.println("Car model: " + myCar.modelName);
        // 외부에서 public 메서드를 통한 엔진 시작
        myCar.startEngine(); // 주석: 외부에서 public 메서드 호출 가능
        // 내부에서 호출 가능한 메서드를 통한 엔진 타입 설정
        myCar.setCarEngineType("Gasoline");
        // 외부에서 호출 가능한 public 메서드를 통한 private 메서드 호출
        myCar.publicMethod();

        // myCar.engineType = "Diesel"; // 외부에서 private 필드에 직접 접근 불가능
        // myCar.internalMethod(); // 외부에서 private 메서드 호출 불가능
    }
}

```

`private` 접근 제한자는 같은 클래스 내부에서는 문제없이 접근할 수 있지만 다른 클래스에서는 접근할 수 없다. 필드, 메소드, 생성자 모두 적용할 수 있고 클래스 내부에서만 접근 가능하다.

```

39  public static void main(String[] args) {
40
41      Circle c1=new Circle();
42      Circle c2=new Circle(9);
43      //private 접근불가
44      //System.out.println(Circle.totalCount);
45      System.out.println(Circle.getTotalCount());
46      Circle c3=new Circle(10);
47      System.out.println(Circle.getTotalCount());
48
49      System.out.println(Circle.PI);
50      //Circle.PI=3.141592; 접근할수 있으나 상수여서 변경 불가능
51
52      System.out.println(c1.getSerialNumber());
53      System.out.println(c2.getSerialNumber());
54      //c1.createSN() 접근불가
55
56  }
57 }

```

접근 관계 유무를 확인해 보자.

`Circle.totalCount` 는 `private`로 선언되어 있어서 접근 불가능 하다. 44번을 확인해 보자.

49번을 확인해 보면 `public` 으로 선언한 `Circle.PI` 접근 가능하다. 인스턴스가 `c1`이라 할 때

`c1.getSerialNumber()`는 `public`으로 선언되어 있어서 접근 가능하다. `c1.createSN()`은

확인하면 `private`로 선언되어 있어서 접근 불가능하다.

같은 클래스 안에서 `this.`이나 `Circle.`은 같은 클래스 안에서는 중복된 이름이 있어서 식별이 불가능한 특별한 경우를 제외하고는 생략 가능 하지만, 생략시 혼돈을 줄수 있기 때문에 되도록 생략하지 말자. `this.`은 인스턴스 자기 자신의 주소를 의미 하고, `Circle.`은 클래스 필드를 의미한다.

전체적인 `Circle`클래스 설명은 다음과 같다.

37~41번 까지 같은 이름으로 생성자를 2개 만든 것이다. 생성자는 리턴값 기술없이 클래스와 동일한 이름으로 “접근제한자 클래스이름(){}”과 같은 형태로 만든다.

생성자 이름은 같고 매개 변수가 다른 생성자 2개를 만든 것을 확인할 수 있다. 생성자나 메소드는 이름으로 식별하기 때문에 이름이 같으면 안된다. 하지만 이름이 같더라도 매개 변수 자료형이나 개수가 다르면 같은 이름 이더라도 식별이 가능해 사용할 수 있다. 이런 경우를 전문 용어로 오버로딩(오버로드overload)라 한다. 오버로드시 주워야 할 것이 매개 변수 개수나 자료형과 관계가 있지 매개 변수 이름과는 관계가 없다. 생성자는 오버로드가 가능하여 객체 생성시 생성자의 매개변수에 따라 다른 생성자 함수가 실행 된다.

`Circle c1=new Circle();`과 같이 `c1`를 생성하면 37번 코드의 생성자가 실행이 되고 `Circle c2=new Circle(5);`과 같이 `c2`를 생성하면 38번코드의 생성자가 실행이 된다.

클래스에 생성자는 반드시 기술되어 있어야 해서 개발자가 생성자를 만들지 않으면 컴파일러가 `Circle`클래스 생성자를 자동으로 `public Circle() {}`과 같은 기본 생성자를 만들어 넣는다. 하지만 사용자가 생성자를 하나라도 기술 한다면 기본 생성자는 자동으로 생성되지 않는다. 상위 코드에서 37번 라인의 코드가 없다면 사용자가 다른 생성자를 구현하였기 때문에 자동으로 추가되지 않아 `Circle c1=new Circle();` 해당 코드는 컴파일 에러가 난다. 하지만, 37~41라인까지 삭제해서 모든 생성자가 삭제 된다면 기본생성자를 기술하지 않았더라도 자동으로 컴파일러가 기본 생성자를 추가해 줘서 `Circle c1=new Circle();` 부분이 에러가 나지 않는다.

37라인의 `this(5)`은 해당 클래스의 생성자중 매개변수 `int` 1개를 처리할 수 있는 생성자를 다시 호출해서 실행 하라는 의미이다. 반드시 생성자 메소드 최상단에 기술해야 하며 중간에 기술할 수 없다.

메소드는 필드와 마찬가지로 클래스 메소드와 인스턴스 메소드가 있다. 클래스 메소드를 만들때에는 `static` 키워드를 사용 한다. 다음을 확인해 보자.

## 클래스 메소드 선언 방법

접근제한자 `static` 자료형 메소드명 (매개변수 ...) {}

## 인스턴스 메소드 선언 방법

접근제한자 자료형 메소드명 (매개변수 ...) {}

클래스 필드와 마찬가지로 클래스 메소드는 클래스이름으로 접근 가능하고 인스턴스 메소드는 인스턴스를 통해서 접근 할 수 있다.

상위 Circle관련 예제 코드에 넓이와 둘레 `toString` `equals` `hashcode` `setter` `getter`를 추가해 보고 Circle인스턴스를 생성해 여러 필드와 메소드를 사용해 보자.

인스턴스 필드의 경우 해당 클래스 내부의 인스턴스 메소드나 생성자된 인스턴스 변수를 통해서만 접근 가능하다. 이외의 지역에서는 인스턴스 필드를 직접 접근할 수 없고 인스턴스를 통해서만 인스턴스 필드에 접근할 수 있다.

클래스 메소드에서 인스턴스 필드와 메소드에 접근할 수 없는 이유는 인스턴스 생성 시점보다 클래스 메소드 생성시점이 먼저여서 클래스메소드 생성후 실행 시점에 인스턴스 필드와 메소드가 존재하지 않기 때문이다.

클래스 메소드안에서 다른 클래스 메소드와 필드를 호출할 수 있지만 인스턴스 메소드와 필드는 인스턴스 없이 직접 메소드를 호출할 수 없다. 반면에 같은 클래스 안의 인스턴스 메소드에서 다른 인스턴스 메소드나 인스턴스 필드를 직접 호출할 수 있다. 인스턴스 메소드 실행 시점은 인스턴스가 생성된 이후 이므로 모든 인스턴스 메소드가 이미 생성되어 있어 있기 때문이다.

48라인의 `this.r`에서 `this.`은 자기 자신의 인스턴스 주소를 의미하여 현재 메소드가 실행되고 있는 인스턴스의 `r`를 의미한다. 인스턴스 `c1, c2`에서 `c1.r=10, c2.r=20`일때 `c1`에서의 `this.r`은 10을 의미하고 `c2`에서의 `this.r`은 20을 의미 한다.

Circle클래스를 설명을 해보자면 `totalCount`는 Circle클래스의 생성된 인스턴스 전체 개수를 저장하는 클래스 필드로 선언하였고 외부에서 값을 변경하면 생성된 인스턴스 개수에 왜곡이 일어날 수 있으므로 아무나 변경할 수 없도록 접근제한자를 `private`로 선언 하였다. 원의 반지름 표현하는 `r`필드는 원마다 다른 값을 가지고 있으므로 인스턴스 필드로 구현하였고 언제든지 값을 변경할수 있도록 `public`으로 선언하였다. `getTotalCount()` 메소드의 경우 `private`로 선언된 `totalCount`의 값을 직접 읽어 올수 없어서 생성한 `public`메소드이다. `createSN()`메소드의 경우 내부에서만 접근할 수 있도록 `private`로 선언하였다. `area`메소드는 원의 둘레를 구하는 메소드이다. 원마다 다른 값으로 저장된 인스턴스 필드 `r`에 저장되어 있어 인스턴스 메소드로 구현 하였다.

다음 Card클래스를 통해서 다시한번 확인해 보자.

```

2 public class Card {
3     public static int width=80; //카드의 width
4     public static int height=140; //카드의 height
5     public int number=1;
6     public String numberShape="A";
7     public String shape="하트";
8
9     static {Card.width=80; Card.height=140;}
10    public Card() {}
11    public static void displaySize() {
12        System.out.println("카드의 넓이는"+Card.width
13                +" 높이는"+Card.height);
14        //static메소드는 객체 생성전에 클래스 이름으로 접근 가능하므로
15        //this.shape,this.numberShape은 접근할 수 없다.
16    }
17    public void displayCard() {
18        System.out.println("카드의 넓이는"+Card.width
19                +" 높이는"+Card.height+"카드 모양은 "+this.shape
20                +" 카드 숫자는"+this.numberShape);
21    }
22 }

```

1. 클래스: 상위코드가 클래스이다. 현실 세계에 존재하는 객체를 프로그램에 사용할 수 있도록 형상화 한 것이다. `class`라는 키워드를 사용하여 구현하였다.

2. 인스턴스: 클래스와 `new`연산자를 사용해서 인스턴스를 생성한 것을 의미 한다. 하나의 클래스는 여러개의 인스턴스를 가질수 있다.

3. 필드: 3~7번라인에 있는 클래스에서 선언된 변수로 클래스 필드와 인스턴스 필드 이다.

4. 클래스필드: 정적필드:`static`필드: 같은 의미로 쓰이며 3,4번 라인에 선언된 필드들이 이에 해당한다. 클래스 필드들은 인스턴스를 생성하지 않아도 클래스이름으로 접근할 수 있고 인스턴스를 여러개 생성하더라도 하나만 생성되고 모든 인스턴스에서 값을 공유할 수 있고 전역변수 개념이여서 모든 지역에서 클래스 필드를 접근 할 수 있다. `Card`클래스는 한장의 카드를 저장하는 클래스이다. 카드의 `shape`와 `number`는 카드마다 다르지만 `width`와 `height`는 모든 카드가 값이 동일 하여 따로따로 선언 할 필요가 없다. 이렇게 카드마다 다른 값을 저장해야하는 데이터인 `shape`나 `number`등은 인스턴스 필드로 선언하고 모든 카드가 동일한 값을 가지는 `width`와 `height`는 클래스 필드로 선언한다. 상위 코드 처럼 `Card.width` `Card.height`로 클래스 필드에 접근 할 수 있고, “`Card.`” 은 식별 가능하면 생략 가능 하지만 되도록 생략하지 말고 사용 하자.

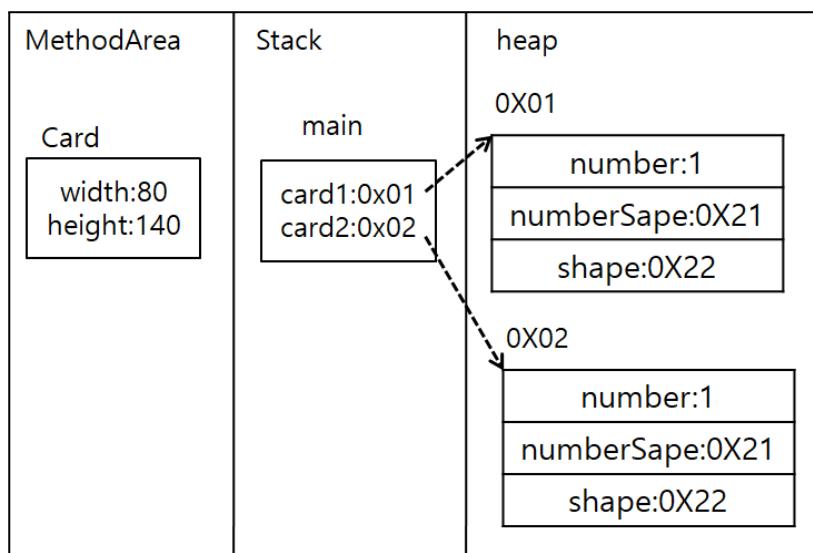
5. 인스턴스필드: 동적필드: 같은 의미로 쓰이며 `new` 연산자로 힙 메모리 영역에 할당할 때마다 저장 공간이 새로 만들어 진다. 5,6,7번 라인이 이에 해당 한다. 값에 접근하려면 해당 클래스 안에서 `this.shape` 처럼 기술하여 접근 하면 되고 앞에 “`this.`”는 식별이 가능하면 생략 할 수 있지만 되도록 생략하지 말고 사용 하자.

메모리에 잡힐 때 `static` 붙은 멤버들은 메소드영역에 생성되서 전역에서 사용가능하고 한번만 생성되지만 인스턴스 객체는 `new` 로 할당 할때 마다 힙에 생성된다. 힙의 경우

생성된 주소를 알고 있다면 어디서나 접근 가능하다.

다음은 main메소드에서 Card card1=new Card(); Card card2=new Card(); 코드가 실행

되었을 때 메모리에 잡힌 모양을  
그림으로 그린 것이다. 제일  
먼저 메소드 에어리어에  
Card클래스의 클래스필드  
width, height가 잡힌다.  
클래스 필드 여서 모든 지역에서  
접근 할 수 있다. card1,  
card2는 메소드에 선언된  
지역변수여서 stack에 잡히고  
실제 데이터는 heap에  
생성된다.



초기화 하는데 사용 한다. 실행 시점에 인스턴스가 존재하지 않아 인스턴스 필드나 메소드에 접근할 수 없다. 정적블럭은 메모리에 최초 올라갈때 1번 실행된다.

7. 생성자: 클래스의 인스턴스가 새로 생성될 때마다 실행되며 자기 자신의 인스턴스 필드를 초기화 하는데 사용 한다. 10번라인에 해당한다.

8. 클래스 메소드: 동적 메소드: static 메소드 11번 라인에 해당하며 객체 생성없이 클래스 이름으로 메소드를 실행할 수 있다. Card.displaySize(); 보통 Card클래스와 관련이 있지만 인스턴스 필드와는 관련이 없는 내용을 메소드로 만들 때 사용한다.

9. 인스턴스 메소드 17번 라인에 해당하며 인스턴스에 점을 찍어 인스턴스.displayCard() 형태로 메소드를 실행시킨다. 보통 해당 인스턴스와 관련 있어 인스턴스 필드와 메소드에 접근하는 일이 있을 때 사용한다.

다음을 보고 이해해 보자.

1. 다음 이미지를 보고 무엇이 잘못 되었는지 생각해 보자.

The screenshot shows the Eclipse IDE interface with the following details:

- Package Explorer:** Shows a Java project named "997JavaStart" with two source folders: "src" and "Myclass".
  - "src" contains a package named "coke\_01" which has a file "MyClass02.java".
  - "Myclass" contains a package named "sec01.exam01" which has files "myClass01.java" and "JavaStory.java".
- Code Editor:** Displays the content of "myClass01.java". The code is as follows:

```
1 package sec01.exam01;
2 import sec01.exam01.*;
3 import sec01.exam01.Car;
4 public class myClass01 {
5
6     public static void ma
7             // TODO Auto-gene
8 }
```

- sec01.exam01 패키지 안에 Car 클래스가 없다.

- 같은 패키지안에 클래스는 import하지 않아도 패키지 경로를 생략하고 사용할 수 있지만 다른 패키지에 선언된 클래스는 반드시 import한후 사용해야 한다.

2. 다음 이미지를 보고 무엇이 잘못 되었는지 생각해 보자.

```

1 package com.Human.dto;
2
3 public class Car {
4     public String color="";
5     public String company="";
6     public int maxSpeed=0;
7     public String model="";
8     public Tire tire=new Tire();
9
10
11 }

```

- Tire클래스가 없어서 에러가 난것이다. 내부에 선언한 클래스를 먼저 구현해야 문제가 없다.
- 주어진 코드를 잘보면 Tire 가 아니고 tire라고 기술하였다. tire클래스는 있어도 Tire클래스는 존재하지 않는다.

```

public class MyClass01 {

    public static void main(String[] args)
        // TODO Auto-generated method stub
        Car c1=new Car();
        c1.name="모닝";
        c1.price=500000;
        System.out.println(c1.name);

        Car c2=new Car("moning",70000);
        System.out.println(c2.name);
    }
}

```

3. 다음 이미지를 보고 무엇이 잘못 되었는지 생각해 보자.

- 생성자를 추가후 기본 생성자를 추가하지 않았다.

```

public Car(String name, int price, Tire tire) {
    super();
    this.name = name;
    this.price = price;
    this.tire = tire;
}

public Car(String name,
          this(name,price); //자기 자신의 생성자로 이동
          this.tire=tire;
}

```

4. 다음 이미지를 보고 무엇이 잘못 되었는지 생각해 보자.

- 매개변수의

형태가 동일한 생성자를 2개 생성 하였다. 생성자의 매개변수 같은 것이 없어야 식별해서 실행 시킬수 있다. 다음 문제들을 풀어보자.

5) 텔레비전을 켜고 끌 수 있고, 채널을 변경할 수 있고, 볼륨을 조절할 수 있는 형태로 TV클래스를 만들어 보고 운영해 보자.

6) 차의 속도에 따라 자동으로 기아가 조절되는 클래스를 만들어 보고 운영해 보자.

//속도는 0~100까지 가능하다.

//10km/h일때 기아 1단계 //20km/h일때 기아 2단계

//30km/h일때 기아 3단계 //40km/h일때 기아 4단계 이하 생략

//speedUp, speedDown 메소드를 이용해서 속력을 조절할 수 있다.

7) 블랙잭 프로그램을 클래스로 만들어 보자.

8) 뱅크 프로그램을 클래스로 만들어 보자.

## > 10. 클래스의 생성자와 메소드

클래스의 사용을 좀 더 편안하게 하기 위해서 여러 방법들을 제공 한다. 그중에서

1.생성자, 2.`toString`, 3.`equals` 4.`getter`, `setter` 를 간단히 확인해 보자.

1. 생성자에 대해서 살펴보자.

```
package com.human.ex;
import com.human.dto.*;
public class JavaStart010 {
    public static void main(String[] args) {
        Human h1=new Human();
        h1.name="홍길동";
        h1.age=25;
        h1.height=166.5;

        System.out.println("name:"+h1.name);
        System.out.println("age:"+h1.age);
        System.out.println("height:"+h1.height);
    }
}
```

왼쪽 이미지는 생성한  
객체에 값을 넣은 다음  
객체의 내용을 출력하는 코드  
이다.

만약 왼쪽 이미지의 색칠한  
부분을 다음 코드처럼 변경  
하여 같은 결과를 얻을 수  
있다면 좀더 쉽게 클래스  
인스턴스를 생성해서 초기화  
할 수 있을 것이다.

```
Human h1=new Human
("홍길동",25,166.5);
```

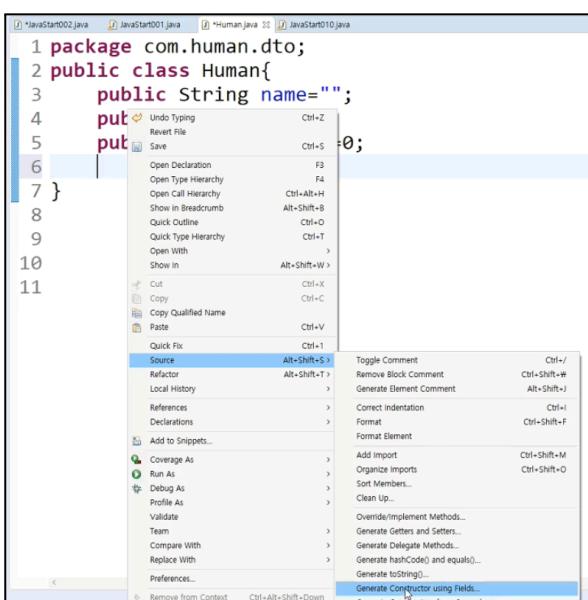
코드와 같은 매개변수가 있는 생성자를  
사용하면 생성과 함께 클래스 변수를 초기화  
할 수 있다.

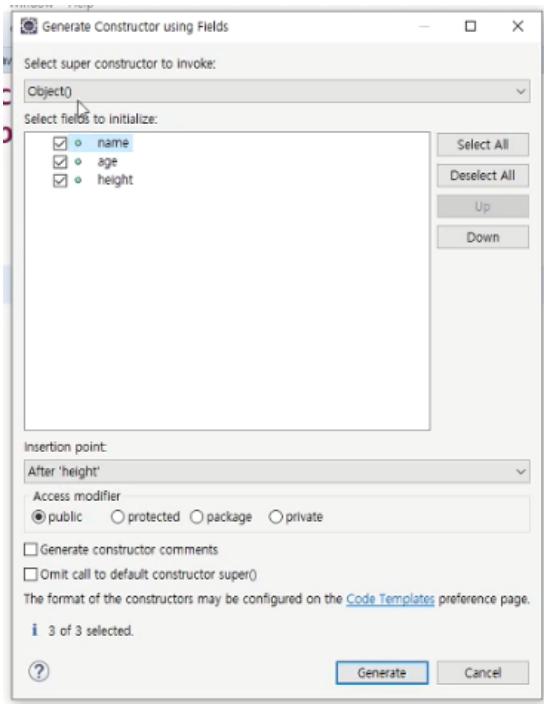
소괄호 안에 있는 데이터들을 매개 변수라  
한다.

일단은 그냥 따라서 생성자 함수를 만들어  
사용해 보자.

왼쪽 이미지에서 이전에 만든  
`com.human.dto` 패키지의 `Human.java` 파일을  
열어서 `Human` 클래스 코드 내부에 커서를 두고  
마우스 오른쪽 클릭한 다음 `source >>`  
`Generate Constructor using fields`를

클릭하면 다음 이미지처럼 팝업으로 창이 뜬다.





여기서 중간에 있는 체크 박스를 모두 클릭하고 오른쪽 하단 generate 버튼을 클릭하면 모든 작업이 끝난다. 여기서 클릭한 필드만이 생성자의 매개변수로 사용된다.

generate버튼을 누른 다음 Human 클래스 파일을 열어보면 다음 이미지 색칠된 부분이 추가되어 있는 것을 확인할 수 있다. 이것이 생성자인데 이것을 추가해야 Human h1=new Human("홍길동",25,166.5); 과 같은 매개 변수가 있는 생성자 형태로 값을 초기화 할수 있다. 추가된 생성자 코드를 잘 살펴 보면 new Human("홍길동",25,166.5); 부분의 매개변수와 일치하도록 정의 되어 있는 것을 확인 할 수 있다.

```
package com.human.dto;
public class Human{
    public String name="";
    public int age=0;
    public double height=0;
    public Human(String name, int age, double height) {
        super();
        this.name = name;
        this.age = age;
        this.height = height;
    }
}
```

여기서 super();는 부모의 생성자를 호출 하라는 이야기 이다. 이해가 안되면 무시하고 넘어가자.

여기서 this.name은 자기 자신의 클래스에 선언한 필드이고 this.이 없는 name은 생성자의 매개변수로 넘어온 name이다. 이름이 같아서 구분을 위해서 this를 사용 하였다.

this는 자기자신의 인스턴스 주소를 의미한다. super와 this는 나중에 자세히 이야기해볼 예정이다. 중요한 것은 자동으로 추가된 코드가 새로운 생성자로 생성시의 인스턴스의 값을 채울수 있도록 해준다는 것이다.

```
package com.human.dto;
public class Human{
    public String name="";
    public int age=0;
    public double height=0;
    public Human() {
    }
    public Human(String name, int age, double height) {
        super();
        this.name = name;
        this.age = age;
        this.height = height;
    }
}
```

생성자가 추가 되면 이전에 사용한 Human h1=new Human(); 코드가 에러가 발생 한다. 정상적으로 동작하게 하려면 왼쪽 이미지처럼 색칠된 부분을 추가해야 한다.

new Human();과 같은 형태로 인스턴스를 생성 하려면 반드시 기본 생성자인 public Human(){}이 있어야 한다.

사용자가 기술하지 않으면 자동으로 기본 생성자인 `public Human(){}`을 컴파일러가 만들어 주지만 사용자가 특정 생성자를 만들어 추가하면 기본 생성자는 만들어 지지 않는다.

그래서, 다른 생성자를 추가하면 사용자가 직접 기본 생성자를 만들어 주어야 한다. 기본 생성자는 `new Human();`를 통해서 사용 된다. 평소 사용하지 않더라도 나중에 사용할 수 있으니 추가해두는 습관을 기르자.

```
1 package com.human.ex;
2 import com.human.dto.*;
4 public class JavaStart010 {
5     public static void main(String[] args) {
6         Human h1=new Human("홍길동",25,166.5);
7
8         h1.age=26;
9
10        System.out.println("name:"+h1.name);
11        System.out.println("age:"+h1.age);
12        System.out.println("height:"+h1.height);
13
14        System.out.println(h1);
15    }
16}
17 }
```

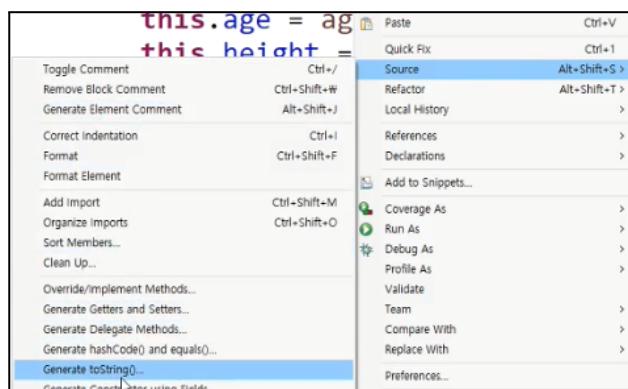
생성자가 완성되면 왼쪽 6번 라인 처럼 생성자를 통해서 객체를 만들수 있고 이전 처럼 `.을` 이용하여 값을 변경하는 작업도 할 수 있다.

다음으로 `toString`에 대해서 확인해 보자.

상위 코드 14번 라인의 출력 결과를 확인해 보면 다음과 같다.

```
1 package com.human.dto;
2 public class Human{
3     public String name="";
4     public int age=0;
5     public double height=0;
6     //생성자
7     public Human() {
8
9     }
10    public Human(String name, int age,
11                  super();
12                  this.name = name;
13                  this.age = age;
14                  this.height = height;
15    }
16    //toString
17 }
```

클래스 안에 선언된 변수를 필드라 한다.  
`com.human.dto.Human@cf7049a7` 컴퓨터마다 다르게 출력될 수 있으나 비슷 할 것이다.  
객체의 주소 해시 코드 값이 출력된 것이다.  
하지만, 주소 보다는 현재 객체가 가지고 있는 필드의 내용을 화면에 출력하고 싶은 경우가 실제 프로그램에서는 더 많다. 이때 `toString`이라는 메소드를 추가하여 주소 대신에 클래스안의 필드 내용과 같은 특정 문자열이 출력 되게 할 수 있다.



이전 코드 이미지의 `toString` 주석 부분을 클릭하고 마우스 오른쪽을 클릭한 다음 왼쪽 이미지 처럼 `source > generate` `toString` 를 차례대로 선택하면 아래 이미지와 같은 창이 뜬다. 해당 창에서 출력 하였을때 화면에 출력하고 싶은 필드를 클릭하고 하단의 `generate` 버튼을 클릭하면 아래 왼쪽 색칠된 부분처럼 자동으로 코드가 만들어 진다.

`toString`를 추가하면 객체를 찍을때 객체의 주소 대신에 다음 오른쪽 이미지 처럼 색칠된 부분 `return`이라고 써있는 뒷부분의 문자열이 출력 된다. 이전 `main`에 기술한 코드를 실행하면 화면에 14라인에서 주소 대신 `return`이라고 써있는 뒷부분의 문자열이 출력 된다.

```

public double height=0;
//생성자
public Human() {
}

public Human(String name, int a
super();
this.name = name;
this.age = age;
this.height = height;
}

//toString
@Override
public String toString() {
    return "Human [name=" + na
}

```

상위 `toString()`에서는 생성자와 다르게 `this`를 사용하지 않았다. `toString` 메소드에서 매개 변수가 없어서이다. 이렇게 `toString` 내부에서 `name`에 접근할 수 있는 변수가 없다면 자동으로 클래스의 필드에서 `name`을 찾아 사용한다. 결국 `this`가 생략된 것이다.

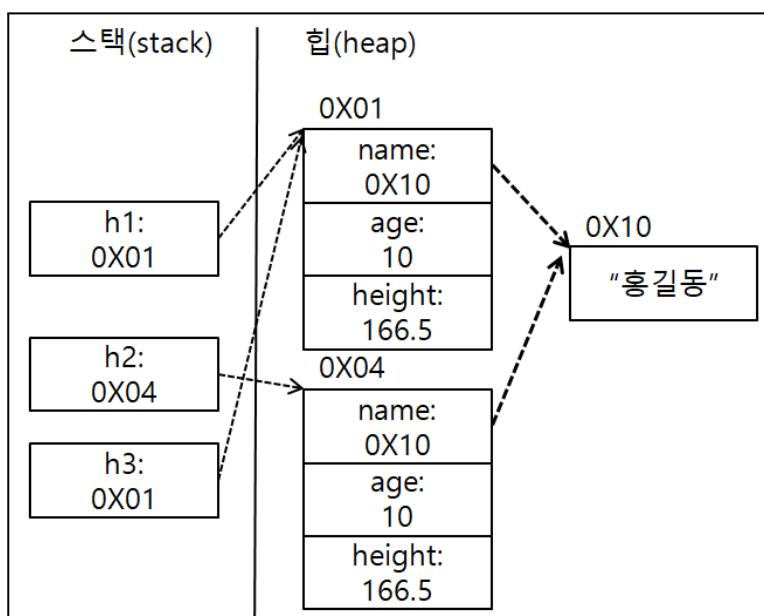
이전 사용한 `String` 클래스도 내부적으로 `toString` 메소드가 정의되어 주소 대신 문자열이 출력된 것이다. 이전 코드 14라인에서 주소 대신 객체정보가 문자열로 출력되는 것을 확인하자.

## > 11. 참조 자료형의 비교 ==과 equals

equals는 두 객체를 비교하는데 사용하는 메소드이다. ==는 변수에 들어 있는 값은 값을 비교하는 연산자이다. 이번 챕터에서 두개의 차이를 이해해 보자. toString이나 생성자처럼 equals라는 메소드를 Human클래스에 추가 하여 사용 할 수 있다. 다음 코드에서 equals관련 메소드 추가 여부에 따라 결과가 달라지는 것을 확인 할 수 있다. 아직 equals를 추가하지 않아 추가 전 값이 나올 것이다. 주석을 잘 확인해보자. 이후 추가하고 나면 추가 후 결과가 출력될것이다. 이전에 만든 Human클래스를 가지고 일단 다음 코드를 완성해서 실행 해서 equals 추가전의 결과가 나오는지 확인해 보자. 이후 equals를 추가하고 실행하면 추가후 결과가 나올것이다.

```
1 package com.human.ex;
2 import com.human.dto.Human;
3 public class JavaStart011 {
4     public static void main(String args[]) {
5         Human h1=new Human("홍길동",10,166.5);
6         Human h2=new Human("홍길동",10,166.5);
7         Human h3=h1; //equals //추가전 //추가후
8         System.out.println(h1==h2); //false//false
9         System.out.println(h1==h3); //true //true
10        System.out.println(h1.equals(h2)); //false//true
11        System.out.println(h1.equals(h3)); //true //true
12    }
13 }
14 }
```

아래 이미지는 상위 코드를 메모리로 그린 것이고, 상의 이미지 h1과 h2 클래스 필드는 둘다 new Human("홍길동", 25, 166.5); 값을 가지고 있다. 하지만, h1과 h2가 같은지



물어보는 == 연산자를 실행 시켜 보면 false라는 결과가 나온다. 이유는 아래 그림에서처럼 h1과 h2에 들어 있는 주소 데이터가 다르기 때문이다. h1, h2 변수 각각 new로 할당 하여서 주소가 다른 위치에 각각의 인스턴스가 생겼다. h1==h2의 결과는 h1, h2에 들어 있는 주소를 비교하는 것이지 들어 있는 주소가 가리키는 값을 비교하라는 이야기가 아니다. 따라서, 주소가 다르므로 8번 라인에 false가 출력되는 것을 확인 할 수 있다.

`h1==h3`는 주소가 같으므로 `true`이다.

`h1==h2`는 `0x01==0x04`를 비교해서 결과가 `false`가 된다. 하지만, "홍길동", 10, 166.5 처럼 실제 들어 있는 데이터가 같으면 `true`가 되게 만들고 싶을 경우가 있는데 `equals` 메소드를 이용하여 원하는 객체의 특정 필드들이 같은지 여부에 따라 `true`, `false`를 생성하게 만들 수 있다. 여기서 필드란 클래스안에 선언된 `name`, `age`, `height` 같은 변수들을 의미한다. 상위 코드를 실행해 보면 `equals`를 사용 하여도 `true`가 아닌 `false`가 리턴된다. `equals` 메소드를 추가 하지 않으면 `equals`는 `==` 이랑 동일하게 동작 하기 때문이다.

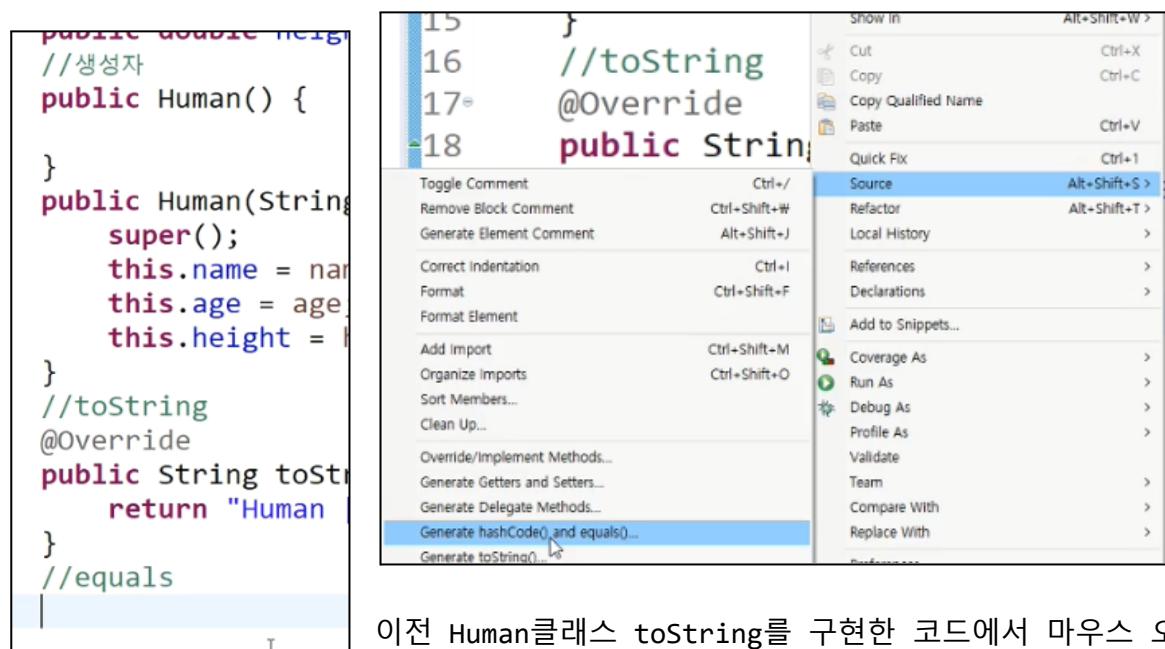
다시 한번 정리해 보자면 다음과 같다.

비교 연산자로 `==` 이 있으나 이것은 해당 변수에 들어 있는 값을 비교하는 것이다. 인스턴스를 담고 있는 클래스 변수의 경우 해당 인스턴스를 가리키는 주소 값이 들어 있어 주소를 비교하게 된다.

```
Human h1 = new Human("홍길동", 25, 166.5);
Human h2 = new Human("홍길동", 25, 166.5);
```

`h1`과 `h2`는 같은 데이터인지 생각해 보자. 인스턴스에 들어 있는 값은 같으나 다른 메모리 공간에 잡혀 있어서 인스턴스를 가리키는 주소는 다르다. 따라서, `h1 == h2`는 들어 있는 값이 같아서 `true`로 착각하기 쉽지만 다른 주소값을 가지고 있으므로 `false`이다. 하지만, 주소는 다르지만 내용이 같은 인스턴스를 같은 인스턴스로 보고 싶은 경우가 있다. 이때 `equals`를 사용하여 주소는 다르지만 내용이 같으면 `true`를 리턴하게 할 수 있다.

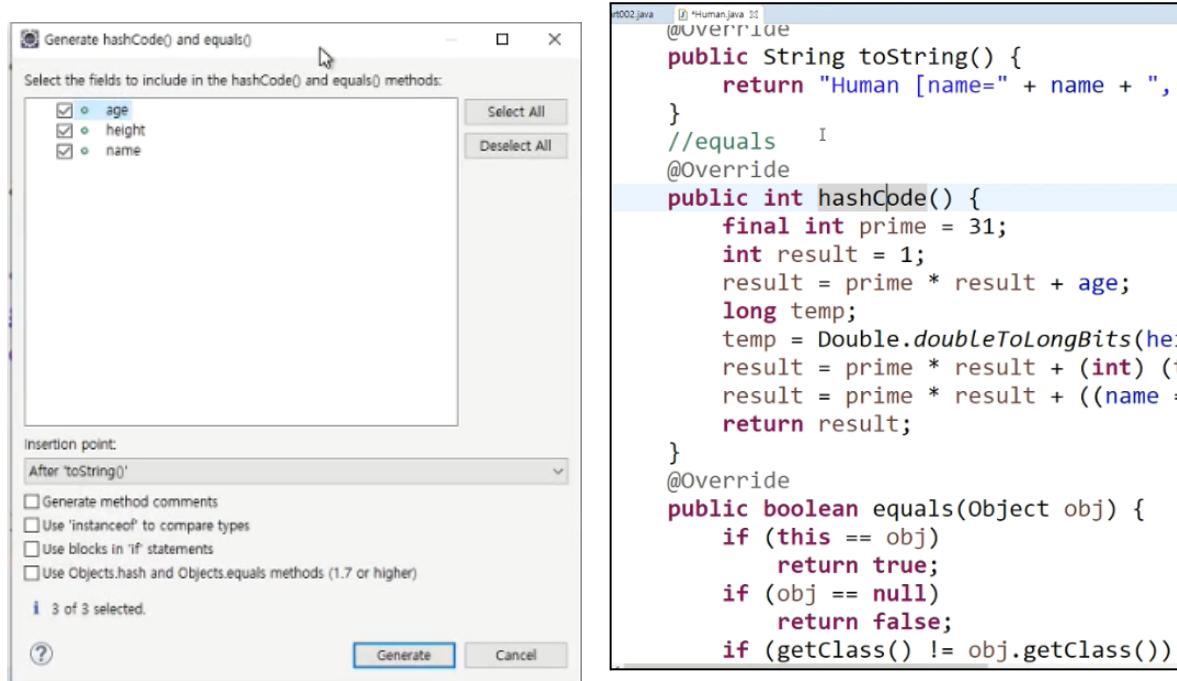
`Human` 클래스에 `equals`메소드를 추가하여 원하는 결과를 얻도록 구현하면 상위 코드에서 선언된 인스턴스 `h1.equals(h2)==true` 의 결과가 참인 것처럼 동작 하도록 만들 수 있다. 다음을 따라서 `equals`를 재정의 하여 원하는 필드를 비교할 수 있다.



이전 `Human` 클래스 `toString`를 구현한 코드에서 마우스 오른쪽 클릭을 해서 `source >> generate hashCode() and equals()` 항목을 클릭하면 아래 왼쪽 이미지 같은 팝업창이 나오고 중간에 필드들이 나오는데

인스턴스 비교시 사용할 필드를 클릭하고 generate 버튼을 누르면 선택된 필드의 값이 같은 인스턴스를 같은 인스턴스로 보는 equals 메소드를 재정의 할 수 있다. equals메소드가 재정의 되면 .equals로 비교하여 특정 필드가 같으면 true를 얻을 수 있게 할 수다.

인스턴스 비교시 이름만 같으면 같은 객체로 보고 싶다면 이전 장에서 모두 선택할 것이 아니라 name만 클릭하여 선택하면 된다.



우리는 모든 필드가 같으면 같은 객체로 판별 할 예정이므로 모두 체크하고 아래에 generate 버튼을 클릭 하면 상위 오른쪽 코드와 같은 hashCode와 equals 메소드가 만들어지는데 코드 내용은 생각하지 말고 결과적으로 이코드로 인해서 .equals를 통해서 두 인스턴스의 값을 비교할 수 있게 된다고 생각하자.

hashCode메소드도 생성된 것을 확인 할 수 있는데 equals 메소드를 사용하기 위해서 필요한 메소드라 생각하자. 복잡한 내용을 포함하고 있는데 equals를 재정의 하면 반드시 재정의 해주어야 한다.

equals 메소드 추가가 끝나면 .equals로 인스턴스의 값 비교를 할 수 있다.

처음에 두 인스턴스를 equals로 비교하던 첫번째 코드를 실행해 보면 10라인이 true로 바뀌어 있는것을 확인 할 수 있다. 결과를 정의해 보면 8번 라인은 h1과 h2가 다른 주소를 가지고 있으므로 false이고 9라인은 같은 주소를 가지고 있으므로 true이다. 10,11번 라인은 equals가 들어 있는 값을 비교하는 형태로 추가 되어 둘다 같은 값을 가지고 있어 모두 true가 된다. equals메소드를 추가해서 이전 equals가 주소비교 하던것이 인스턴스의 값비교 하는 형태로 변경되었다.

결과적으로 .equals 메소드를 클래스에 추가하면 인스턴스의 == 비교는 주소 비교가 되고 .equals비교는 값비교가 된다.

`String` 객체 비교시 `==` 과 `.equals` 의 차이를 이해해 보자.

`==` 연산자는 두 객체의 메모리 주소를 직접 비교합니다. `.equals` 메서드는 객체의 내용을 비교합니다.

`String` 클래스를 다음 2가지 방법으로 메모리 힙에 생성하고 해당 주소를 가지고 올릴 수 있다.

1. `String name= “홍길동”;`
2. `String name = new String(“홍길동”);`

상위 2개의 차이점은 1번은 힙영역 상수풀에 생성되고 2번은 힙영역의 상수풀이 아닌 일반 힙영역에 생성된다. 상수는 변경이 불가능하여 같은 값을 가지는 상수는 여러개 기술 하여도 상수풀에 한개만 생성되는 특징을 가지고 있다. 반면 `new` 연산자를 사용하여 `String` 클래스를 생성한 경우 상수풀이 아닌 일반 힙영역에 생성되어 `new`로 선언할 때마다 새로운 주소에 만들어 진다.

문자열 상수는 같은 값을 2번 쓴다고 하여 힙 메모리 상수풀에 2번 생성되지 않는다.

`String a1= “소나타”; String a2= “소나타”;` 이때 소나타의 메모리 영역은 힙영역의 상수풀에 1개만 생성된다. 이유는 상수는 변경이 불가능하고 읽기만 가능한 데이터이다. 여러개 생성해 봐야 의미가 없어 메모리상에 하나만 저장이 된다.

`String a1=new String(“소나타”); String a2=new String (“소나타”);`와 같이 `new`연산자를 이용해서 `String` 데이터를 힙에 만들 수 도 있다. 이 때는 힙 메모리에 주소가 다른 2개의 문자열 데이터가 생성 된다.

문자열 상수는 같은 값을 2번 쓴다고 하여 힙 메모리 상수풀에 2번 생성되지 않는다.

`String a1= “소나타”; String a2= “소나타”;` 이때 소나타의 메모리 영역은 힙영역의 상수풀에 1개만 생성 된다. 이유는 상수는 변경이 불가능하고 읽기만 가능한 데이터이다. 여러개 생성해 봐야 의미가 없어 메모리상에 하나만 저장되어 필요 할 때 마다 가져다 사용한다.

1. ‘안녕’을 입력하면 ‘너도 안녕’, ‘잘자’ 입력하면 ‘너도 잘자’, ‘잘가’ 가 입력되면 ‘너도 잘가’가 출력되도록 프로그램을 구현해 보자.

힌트) 문자열의 비교는 `==`으로 하지 말고 `.equals`를 사용해서 해보자.

```
String a1= new String("소나타");
```

a1== "소나타" 이런식으로 사용하면 안되고 a1.equals("소나타"); 이런식으로 사용해야 한다. a1.equals("소나타");의 실행결과는 a1이 소나타이면 true가 생성되고 아니면 false가 생성된다.

```
String str = a1.equals("그랜저")? "그랜저" : "소나타";
```

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

다음으로 String 클래스를 확인해 보자.

다음 참조 데이터 String 클래스의 equals 메소드와 == 연산자를 확인해 보자.

```
//참조자료형에서 string의 equals 메소드는 들어있는 문자열데이터를 비교한다.  
//string의 equals 메소드는 들어있는 문자열데이터를 비교한다.  
String str1="홍길동";  
String str2="홍길동";  
String str3=new String("홍길동");  
String str4=new String("홍길동");  
if(str1==str2) {  
    System.out.println("str1==2는 같다.");  
}  
if(str1==str3) {  
    System.out.println("str1==3는 같다.");  
}  
if(str3==str4) {  
    System.out.println("str3==4는 같다.");  
}  
if(str1.equals(str2)) {  
    System.out.println("str1 equal 2는 같다.");  
}  
if(str1.equals(str3)) {  
    System.out.println("str1 equal 3는 같다.");  
}  
if(str3.equals(str4)) {  
    System.out.println("str3 equal 4는 같다.");  
}
```

String 클래스 참조 데이터는 전문 프로그래머가 미리 문자열이 같으면 true가 리턴 되도록

equals를 재정의 해 두어서 문자열 비교시 바로 사용하면 된다. 왼쪽 이미지는 상위 코드를 메모리로 그린 결과이다. 조심해야 할 것은 new로 생성하지 않는 상수 문자열은 내용이 같으면 메모리 힙영역의 상수풀에 한번만 생성되므로 내용이 같으면 주소도 같다.

==연산자는 다음과 같은 결과를 가진다.

str1==str2는 0x10==0x10이므로 true

str1==str3는 0x10==0x01이므로 false

str3==str4는 0x01==0x02이므로 false

.equals같은 경우 들어 있는 값이 모두 “홍길동”이므로 모두 true가 리턴된다.

결과적으로 값 비교를 하고 싶다면 기본 자료형은 == 으로 비교하면 되지만 참조 자료형은 equals로 비교하여야 한다. ==의 경우 변수에 들어 있는 값을 비교하고 equals같은 경우 처음에는 ==과 동일 하지만, equals 메소드의 재정의를 통해 비교 할 필드를 결정 하여 결정된 필드가 같으면 같은 인스턴스로 보는 작업이 가능 해진다.

다음 문제를 확인해 보자.

1. 사용자가 바위를 입력하여도 무승부가 출력되지 않는다. 무엇이 문제인가?

```
String a=sc.nextLine();

if("바위"==a){

    System.out.println("무승부");

}
```

2. 사용자 입력 true를 받아서 ‘true’가 입력되었습니다’. ‘true’가 입력되지 않았습니다.’ 가 출력 되도록 만들어 보자.

3. 사용자에게 숫자를 입력받아 1004이면 ‘암호가 맞음’ 아니면 ‘암호가 틀림’이 출력되도록 프로그램을 구현해 보자.

답안

String s1 = scanner.nextLine();에서 nextLine()메소드는 사용자 입력을 받아서 힙에 동적으로 문자열을 생성해서 주소를 리턴해주는 메소드이다. s1은 new로 사용자 입력

문자열을 힙에 생성한 주소를 가지고 있는 상태라 생각하면된다.

scanner로 사용자 입력 ‘안녕’를 받으면 프로그램에 있는 상수 ‘안녕’과 주소가 다르다. == 을 사용하면 false가 리턴되니 equals를 사용하면 true가 리턴된다.

## 접근 제한자 private와 public

private와 public은 접근 제한자이다. 클래스안에 public으로 선언한 필드들은 모든 곳에서 사용 할 수 있고, private로 선언한 필드들은 같은 클래스 안에서만 사용 할 수 있다.

### 1. private를 사용한 클래스

```
public class Rectangle {  
    private int height;  
    private int width;  
    // ... 다른 멤버들 및 메소드들 ...  
}  
  
//다음 코드를 메인에 기술하고 실행시켜 보자  
public class Main {  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle();  
        //Rectangle 클래스가 아닌 다른 클래스(Main클래스)이므로 접근 불가 하다.  
        r1.height = 30; // 에러 발생: private 멤버에 접근 불가  
        r1.width = 40; // 에러 발생: private 멤버에 접근 불가  
    }  
}
```

Rectancle 클래스에 toString를 추가해보면 private로 선언 한 필드들을 사용하고 있다. private로 선언한 필드는 같은 클래스에서 접근 할 수 있으나 다른 클래스에서는 접근할 수 없다.

### 2. public를 사용한 클래스

```
public class Rectangle {  
    public int height;  
    public int width;  
    // ... 다른 멤버들 및 메소드들 ...  
}  
  
//다음 코드를 메인에 기술하고 실행시켜 보자  
public class Main {  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle();  
        //public으로 선언해서 다른 클래스여도 접근 가능하다.  
        r1.height = 30; // 정상 동작  
        r1.width = 40; // 정상 동작  
    }  
}
```

상위 두 코드의 차이점은 접근 제한자가 다르다는 것이다. 하나는 `private`이고 하나는 `public`이다.

그동안 `public`만 사용해 왔는데 `public`이 제약 없이 모든 곳에서 접근 가능하기 때문에 사용하기 쉬워서 였다. `private` 접근자의 경우 같은 클래스 안에서만 접근 할 수 있도록 사용 범위 제약이 있다.

상위 1번째 코드로 선언 하였다면 `private` 멤버를 다른 클래스에서 접근하려 하니 에러가 나오고 2번째 코드로 선언 하였다면 `public` 멤버를 다른 클래스에서 접근 가능하므로 문제 없이 동작한다.

다음과 같이 정리 하자. `private`로 선언한 필드는 해당 클래스 안에서만 사용할 수 있고 `public`으로 선언한 필드는 모든 곳에서 접근 할 수 있다.

`getter setter`에 대해서 간단하게 설명하면 다음과 같다.

`getter setter`는 `private`로 선언된 필드에 값을 읽어오고 저장하는 용도로 사용 된다.

`private`로 선언한 필드는 다른 클래스에서 접근할 수 있다. 다른 클래스에서 `private`로 선언한 필드에 접근 하려면 `getter, setter` 메소드를 이용해서 할 수 있다.

이후 내용이 이해 안되면 복잡하게 생각하지 말고 `getter, setter`를 자동 완성해서 만들고 필드에 접근하려면 인스턴스.`getter()`, 인스턴스.`setter(넣을값)`를 사용하면 된다고 생각하자.

다음 예제를 확인해 보자.

```
public class Person {  
    private String name;  
    private int age;  
  
    // Getter 메소드  
    public String getName() {  
        return name;  
    }  
  
    // Setter 메소드  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    // Getter 메소드  
    public int getAge() {
```

```

        return age;
    }

    // Setter 메소드
    public void setAge(int age) {
        this.age = age;
    }
}

public class Main {
    public static void main(String[] args) {
        // Person 클래스의 객체 생성
        Person person = new Person();
        // Setter를 이용한 값 설정
        person.setName("홍길동");
        person.setAge(30);
        // Getter를 이용한 값 조회
        System.out.println("이름: " + person.getName());
        System.out.println("나이: " + person.getAge());
        // 다음과 같은 형태는 private 필드여서 접근할 수 없다.
        //person.name="홍길동";
        //person.age=30;
    }
}

```

내용이 이해 안되면 getter,setter를 자동 완성해서 만들고 필드에 접근하려면 인스턴스.getter(), 인스턴스.setter(넣을값)를 사용하면 된다고 생각하자.

**Getter 메소드 생성:**

필드의 이름 앞에 "get"을 붙이고, 첫 글자를 대문자로 변환합니다.

메소드의 반환 타입은 해당 필드의 타입과 일치해야 합니다.

메소드 내부에서는 해당 필드의 값을 반환합니다.

다음을 추가하면 인스턴스.getHeight()를 이용해서 필드 height의 값을 읽어 올수 있다.

```

public int getHeight() {

    return height;
}

```

**Setter** 메소드 생성:

필드의 이름 앞에 "set"을 붙이고, 첫 글자를 대문자로 변환합니다.

메소드의 매개변수로 해당 필드의 타입과 일치하는 매개변수를 받습니다.

메소드 내부에서는 해당 필드에 전달받은 매개변수 값을 할당합니다.

다음을 추가하면 인스턴스.setHeight(30)를 이용해서 필드 height의 값에 30를 넣을 수 있다.

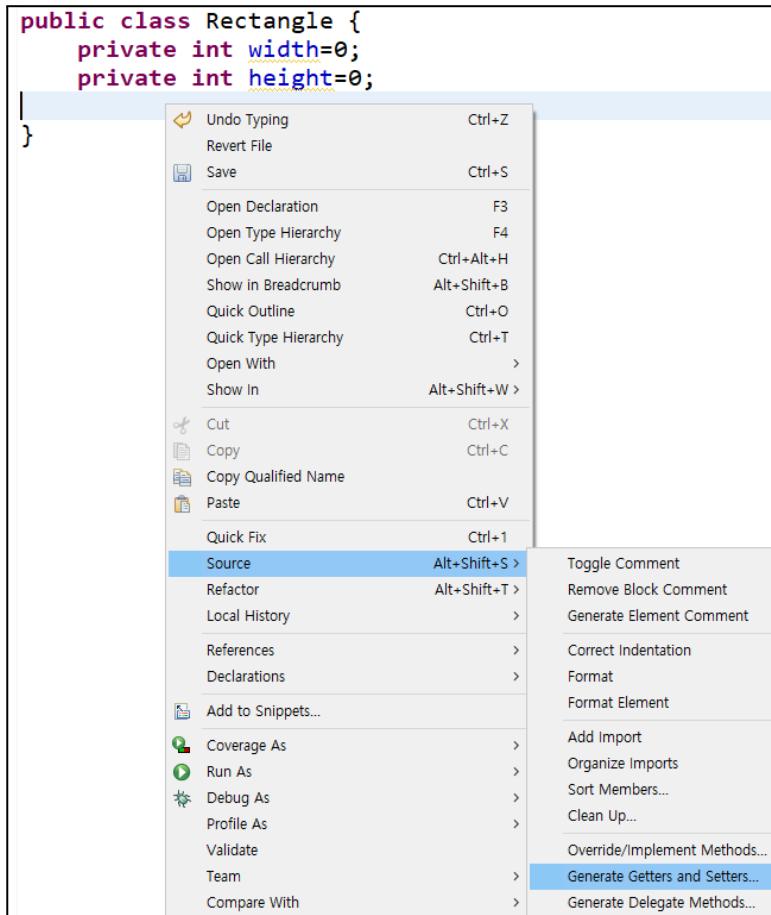
```
public void setHeight(int height) {  
    this.height = height;  
}
```

다음 예제의 Person 클래스의 person인스턴스에 private으로 선언된 name 필드에 값을 넣는 방법은 person.setName("홍길동"); 이고 읽어오는 방법은 person.getName()이다.

**Getter, Setter** 사용:

```
Rectangle r1 = new Rectangle();  
  
r1.setHeight(30);  
  
System.out.println("변경 후 높이: " + r1.getHeight());
```

위의 코드에서 setHeight(30)은 Rectangle 객체의 높이를 30으로 설정하는 Setter 메소드이다. getHeight()메소드는 현재 Rectangle 객체의 높이를 가져오는 Getter 메소드이다.



다음 예제를 따라해서  
hashcode, equals, 생성자,  
toString 메소드, getter,  
setter 등을 자동으로 추가해  
보자.

### getter와 setter 만들기

왼쪽 이미지 처럼 Rectangle  
클래스에 private로 선언된  
필드를 생성한다.

eclipse 툴에서 자동으로 만들어  
주는 방법을 지원해준다. 상위  
이미지 처럼 필드들을 기술한  
다음에 private 필드가 있는  
Rectangle 클래스안에서  
오른쪽마우스클릭>>  
source선택>>generate Getters  
and Setter 항목을 선택 하자.  
그러면 아래와 같은 이미지가  
나올 것이다. select All버튼을

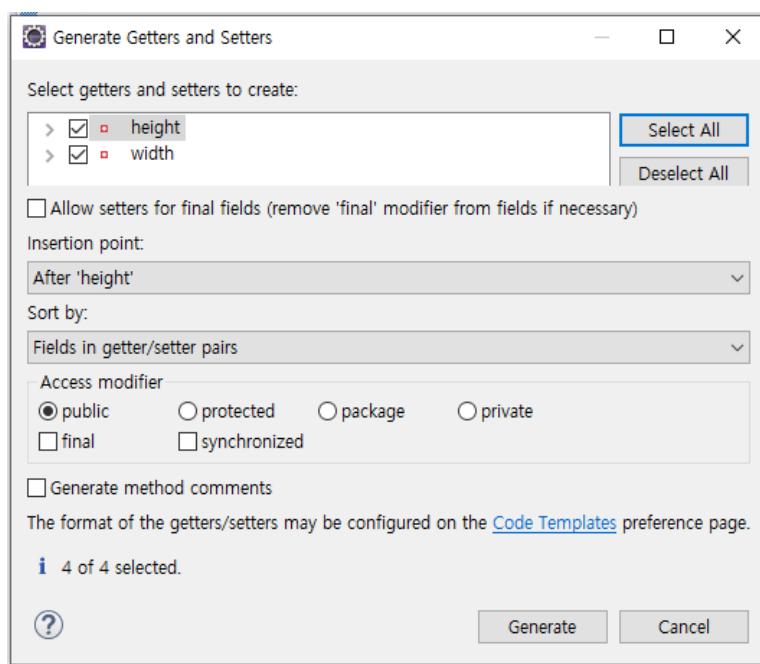
클릭하면 왼쪽 상단에 보이는 체크박스를 사용해 보여주고 있는 필드 목록에 모든 필드들이  
선택되는 것을 볼수 있고 하단의 Generate 를 클릭하면 자동으로 추가된 코드를 확인 할 수  
있다. 과거에 r1.height=30; r1.width=40; 형태로 필드를 변경 하였다면 이제는  
읽어올때는 r1.getWidth() r1.getHeight() 로 읽어오고 값을 변경할 때에는  
r1.setWidth(3)r1.setHeight(4) 를 사용해서 변경 할 수 있다.

```

public class Rectangle {
    private int width=0;
    private int height=0;
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = width;
    }
    public int getHeight() {
        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
}

```

그동안 만든것을 종합해서  
Rectangle클래스를 만들어 보자.



기존 만든 Human클래스에 getter, setter를 추가해 보자.

---

## > 08. Wrapper클래스

---

다음은 Wrapper클래스 관련 설명이다. 기본 자료형을 클래스로 만들어 놓은 것이다.

용도는 기본자료형과 관련된 전역 메소드

부득이하게 기본 데이터를 참조 데이터로 바꿔서 사용해야 할 경우 사용하는 클래스이다. 나중에 배열 컬렉션의 경우에는 참조 데이터만 사용할 수 있다. 이럴 때 기본 데이터를 참조 데이터로 만들어 사용하는데 이때 사용하는 클래스를 wrapper 클래스라 한다. wrapper 클래스 종류는 다음과 같다.

```
int : Integer      char : Character      float : Float      short : Short  
long : Long        double : Double       byte : Byte
```

기본 자료형과 관련된 메소드를 기본 자료형에 추가 할 수 없기 때문에 래퍼클래스에 추가한다. 대표적인 Wrapper클래스의 메소드에는 .parseInt .parseDouble 같은 메소드가 있다. 문자열을 해당 자료형으로 변경하는 메소드이다.

int i1=1; Integer i2=i1; 코드에서 int형 기본 자료형을 Integer 클래스에 넣었다. i1은 자동 형변환 되어 Integer 클래스에 들어 간다.

int i3=i2+5; 다음 같은 경우 Integer클래스가 int형으로 형변환되어 i3에 들어 가게 된다 언박싱이라고 한다. 기본 자료형과 wrapper클래스는 서로 자동 형변환 된다.

나중에 컬렉션을 배우면 제네릭은 데이터로 클래스만 넣을 수 있는데 데이터로 기본 자료형을 넣으려면 wrapper클래스를 사용하면 된다.

---

## > 13. ArrayList

---

ArrayList는 자바 프로그래밍 언어에서 제공되는 컬렉션 클래스입니다. 컬렉션은 자료구조를 자바로 구현한 클래스이다. 자료구조는 변수, 배열 처럼 컴퓨터에서 자료를 다루는 여러 방법들을 의미한다. ArrayList는 list라는 자료구조중 하나로 배열과 유사하지만 인덱스를 사용하지 않고 객체를 생성해 메소드로 데이터 저장 공간을 조작한다. 리스트가 배열과 가장 큰 차이점은 연속된 데이터이다. 배열은 데이터가 없어도 저장 공간을 가지고 있지만 리스트는 중간의 빈공간이 생기면 뒤에 있는 데이터를 끌어당겨 연속된 데이터로 만든다. 배열은 빈공간이 존재 하지만 리스트에는 빈공간이 존재 하지 않는다.

다음 예제를 통해서 ArrayList를 이해해 보자.

ArrayList 생성 및 데이터 추가:

`ArrayList<Integer> arrList = new ArrayList<Integer>();`으로 정수형 데이터를 저장할 ArrayList를 생성합니다.

`arrList.add(1);, arrList.add(2);` 등을 통해 데이터를 추가합니다.

데이터 읽기 및 출력:

for문을 사용하여 arrList의 데이터를 인덱스를 통해 읽어오고 출력합니다.

데이터 삽입:

`arrList.add(2, 9);`로 특정 인덱스에 데이터를 삽입합니다.

데이터 변경:

`arrList.set(3, 10);` 3번 인덱스에 데이터 10 변경

데이터 삭제:

`arrList.remove(2);`와 `arrList.remove((Integer) 5);`로 인덱스와 값으로 데이터를 삭제합니다.

데이터 검색:

`arrList.indexOf((Integer) 4);`로 데이터의 인덱스를 찾습니다.

포함 여부 확인:

`arrList.contains((Integer) 8);`와 `arrList.contains((Integer) 2);`로 데이터의 존재

여부를 확인합니다.

비어있는지 확인:

`arrList.isEmpty();`로 리스트가 비어있는지 여부를 확인합니다.

`Iterator`을 이용한 순회:

`Iterator<Integer> iter = arrList.iterator();`로 `Iterator`을 생성하고, `iter.hasNext()`와 `iter.next()`를 통해 전체 데이터를 순회합니다.

리스트 초기화:

`arrList.clear();`로 리스트의 모든 데이터를 제거하여 초기화합니다.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Collections;

public class MyArrayList {

    public static void main(String[] args) {
        // ArrayList 생성

        // 다음은 arrayList 생성 방법이다. <Integer> 부분은 제네릭이라고 하는데
        // arrayList에 정수만 담을 수 있다는 이야기이다.
        ArrayList<Integer> arrList = new ArrayList<Integer>();

        // 데이터 추가
        // add 메소드와 넣고자하는 데이터를 이용해서 데이터를 리스트의 마지막에
        // 추가할 수 있다.
        arrList.add(1);           arrList.add(2);
        arrList.add(3);           arrList.add(4);
        arrList.add(5);           arrList.add(6);

        // 데이터 읽기
        // 리스트에서 get 메소드와 인덱스를 이용해서 원하는 데이터를 읽어오는 방법과
        // size() 메소드를 이용해서 리스트의 전체 데이터 개수를 얻어오는 방법을 확인해 보자.
        for (int i = 0; i < arrList.size(); i++) {
            System.out.print(arrList.get(i) + " ");
        }
        System.out.println();
```

```
// 데이터 삽입 add 메소드를 이용해서 리스트의 특정 위치에 데이터를 저장할 수 있다.
```

배열:

특정 위치에 데이터를 삽입하면, 해당 위치의 기존 데이터는 사라지고 새로운 데이터로 덮어 씌워진. 다른 데이터는 자리 이동이 없이 그대로 위치 유지됨.

리스트:

특정 위치에 데이터를 삽입하면, 기존 데이터부터 그 이후의 모든 데이터가 한 칸씩 뒤로 이동함. 데이터의 삽입으로 인해 리스트의 크기가 동적으로 조절되며, 유연한 데이터 관리가 가능함.

```
arrList.add(2, 9); // 2번 인덱스에 9 추가
```

```
arrList.set(3, 10); // 3번 인덱스에 10 변경
```

```
System.out.println("---");
for (int i : arrList) {
    System.out.print(i + " ");
}
System.out.println();
```

// 데이터 삭제

arrList.remove(2); // 인덱스로 삭제 삭제후 삭제된 인덱스 뒷부분 모든 데이터가 한 칸씩 앞으로 이동함.

arrList.remove((Integer) 5); // 값으로 삭제 삭제후 삭제된 인덱스 뒷부분 모든 데이터가 한 칸씩 앞으로 이동함. 들어 있는 데이터가 중복되어 있으면 앞쪽에 위치한 데이터로 한계만 삭제된다.

// 중복 데이터가 있으면 1개만 삭제된다.

// 반복문으로 반복적으로 삭제 할때 인덱스로 검색해서 삭제시 뒤부터 삭제 해야 시프트 현상에 영향을 받지 않는다.

// 다음과 같은 방법으로 리스트의 데이터를 순회 할 수 있다.

```
for (int i : arrList) {
    System.out.print(i + " ");
}
```

```

System.out.println();

// 데이터 검색 indexOf와 데이터로 리스트안에 해당 데이터가 존재하는
위치의 인덱스를 얻을 수 있다.
int index = arrList.indexOf((Integer) 4); // 해당 데이터의 인덱스 리턴
System.out.println("Index of 4: " + index); // 없으면 -1

// 포함 여부 확인 contains와 들어있는 데이터를 통해서 현재 리스트에 해당
데이터가 존재하는지 여부를 확인 할 수 있다.
System.out.println("Contains 8: " + arrList.contains((Integer) 8));
System.out.println("Contains 2: " + arrList.contains((Integer) 2));

// 비어있는지 확인 isEmpty메소드로 비어 있으면 true, 데이터를 가지고
있으면 false를 얻을 수 있다.
System.out.println("Is empty: " + arrList.isEmpty());

// Iterator를 이용한 순회 데이터를 순회하는 iterator 객체를 통해
데이터를 순회한다.
Iterator<Integer> iter = arrList.iterator();
while (iter.hasNext()) {//다음 데이터를 가지고 있는지 확인
    System.out.print(iter.next() + " "); //다음 데이터로 이동후값을얻어옴
}
System.out.println();

// 리스트 초기화
arrList.clear(); //리스트에 담긴 모든 데이터를 지움
System.out.println("Cleared list: " + arrList);
}

}

}

```

다음 예제들을 확인해 보자.

```

import java.util.ArrayList;
import java.util.Iterator;
public class ArrayListExample {
    public static void main(String[] args) {
        // 문자열을 저장하는 ArrayList 생성
        ArrayList<String> stringList = new ArrayList<>();
        // 문자열 추가
        stringList.add("Apple");           stringList.add("Banana");
        stringList.add("Orange");          stringList.add("Grapes");
        stringList.add("Mango");
        // ArrayList의 크기 출력
        System.out.println("ArrayList의 크기: " + stringList.size());
    }
}

```

```

// 특정 인덱스의 요소 가져오기
String fruit = stringList.get(2);
System.out.println("인덱스 2의 과일: " + fruit);
// 전체 목록 출력 - 방법 1: 향상된 for문 사용
System.out.println("ArrayList 목록 - 방법 1: 향상된 for문");
for (String item : stringList) {
    System.out.println(item);
}
// 요소 존재 여부 확인
boolean containsBanana = stringList.contains("Banana");
System.out.println("ArrayList에 Banana가 포함되어
있는가? " + containsBanana);
// 특정 요소 삭제
stringList.remove(1); //인덱스 1 삭제
stringList.remove("Grapes"); //리스트에 들어있는 Grapes삭제
// 특정 요소 변경
stringList.set(0, "Grapes"); //0번인덱스 Grapes로 변경
// 전체 목록 출력 - 방법 2: Iterator 사용
System.out.println("ArrayList 목록 - 방법 2: Iterator");
Iterator<String> iterator = stringList.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
// ArrayList 비우기
stringList.clear();
// ArrayList가 비어있는지 확인
boolean isEmpty = stringList.isEmpty();
System.out.println("ArrayList가 비어있는가? " + isEmpty);
}

import java.util.ArrayList;
import java.util.Iterator;
public class ArrayListDoubleIteratorExample {
    public static void main(String[] args) {
        // double 값을 저장하는 ArrayList 생성
        ArrayList<Double> doubleList = new ArrayList<>();
        // double 값 추가
        doubleList.add(3.14);           doubleList.add(2.718);
        doubleList.add(1.618);          doubleList.add(0.707);
        doubleList.add(4.669);
        // 전체 목록 출력 - 방법 1: 향상된 for문 사용
        System.out.println("ArrayList 목록 - 방법 1: 향상된 for문");
        for (double num : doubleList) {
            System.out.println(num);
        }
        // 전체 목록 출력 - 방법 2: Iterator 사용
        System.out.println("ArrayList 목록 - 방법 2: Iterator");
    }
}

```

```
Iterator<Double> iterator = doubleList.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
// 특정 값이 포함된 요소 제거
doubleList.remove();

// 요소 제거 후 목록 출력
System.out.println("요소 제거 후 목록:");
for (double num : doubleList) {
    System.out.println(num);
}
}
```

---

## > 14. 사용자 정의 class를 사용한 ArrayList

---

이 코드는 RectArrayList 클래스에서 ArrayList를 사용하여 Rect 객체들을 다루고 있습니다. 코드 중간 중간에 주석을 확인해서 사용자가 정의한 객체를 arrayList에서 어떻게 조작하는지 확인해 보자.

ArrayList 생성 및 초기화:

`ArrayList<Rect> arrList = new ArrayList<Rect>();`으로 Rect 객체들을 저장할 ArrayList를 생성합니다.

데이터 추가 및 출력:

`arrList.add(new Rect(11,11));`와 같이 Rect 객체들을 ArrayList에 추가합니다.

for문을 사용하여 ArrayList의 데이터를 출력합니다.

데이터 삽입:

`arrList.add(2, new Rect(99,99));`로 특정 인덱스에 데이터를 추가합니다.

데이터 삭제:

`arrList.remove(2);`과 `arrList.remove(new Rect(31,11));`로 인덱스와 객체를 통해 데이터를 삭제합니다.

데이터 검색 및 수정:

`arrList.indexOf(new Rect(11,21));`로 데이터의 인덱스를 찾고, `arrList.set(2, new Rect(44,55));`로 데이터를 수정합니다.

Iterator를 통한 전체 데이터 순회:

`Iterator<Rect> iter = arrList.iterator();`로 Iterator를 생성하고, `iter.hasNext()`와 `iter.next()`를 통해 전체 데이터를 순회하면서 출력합니다.

```
//com.human.dto.Rect.java
package com.human.dto;
import java.util.Objects;
public class Rect {
    public int width=0;
    public int height=0;
    public Rect() {}
    public Rect(int width, int height) {
        this.width = width;      this.height = height;
```

```

    }

    @Override
    public int hashCode() {
        return Objects.hash(height, width);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Rect other = (Rect) obj;
        return height == other.height && width == other.width;
    }

    @Override
    public String toString() {
        return "Rect [width=" + width + ", height=" + height + "]";
    }
}

//com.human.ex.RectArrayList.java

package com.human.ex;
import java.util.ArrayList;
import java.util.Iterator;
import com.human.dto.Rect;
public class RectArrayList {
    public static void main(String[] args) {
        ArrayList<Rect> arrList=new ArrayList<Rect>();
        //add메소드를 이용해서 사각형 객체 6개를 리스트 마지막에 추가
        arrList.add(new Rect(11,11));arrList.add(new Rect(21,11));
        arrList.add(new Rect(11,41));arrList.add(new Rect(11,21));
        arrList.add(new Rect(31,11));arrList.add(new Rect(11,31));
        for(int i=0;i<arrList.size();i++) {//size를 통해서 배열의 개수 생성
            //toString이 있어야 찍힌다.
            System.out.println(arrList.get(i));
        }
        System.out.println("-----");
    }
}

//add메소드를 이용해서 특정위치에 데이터를 추가하는 방법이다.
//특정 위치에 추가되면 기존 특정 위치 데이터가 사라지는 것이 아니라 특정
위치 이후에 들어 있는 모든 데이터가 뒤로 하나씩 밀린다.
arrList.add(2, new Rect(99,99));//2번인덱스에 3추가 (인덱스, 넣을값)
for(Rect i :arrList) {
    System.out.println(i);
}
System.out.println("-----");
//특정위치의 데이터를 삭제하는 방법 특정위치의 데이터가 삭제되면

```

특정위치 이후에 들어 있는 모든 데이터가 앞으로 하나씩 당겨진다.

```
arrList.remove(2); //index 삭제  
//Rect 클래스에 equals가 있어야 정상 동작한다.  
arrList.remove(new Rect(31,11)); //데이터를 이용한 삭제  
for(Rect i : arrList) {  
    System.out.println(i);  
}  
System.out.println("-----");  
//indexOf 메소드로 해당 데이터의 인덱스 위치 확인 리턴 없으면 -1  
int index=arrList.indexOf(new Rect(11,21)); //equals가 있어야 동작  
System.out.println(index);  
//.contains 메소드는 해당 데이터가 존재하는지 여부 확인 true, false  
System.out.println(arrList.contains(new Rect(11,21))); //equals  
System.out.println(arrList.contains(new Rect(91,21))); //equals  
System.out.println(arrList.isEmpty()); //비어 있는지 true false 리턴  
  
//set 메소드를 이용해서 데이터를 수정할 수 있다.  
arrList.set(2, new Rect(44,55)); //수정 시프트가 일어나지 않는다.  
  
Iterator<Rect> iter=arrList.iterator(); //전체 데이터 순회  
while(iter.hasNext()) { //인덱스 다음에 데이터가 있는가?  
    // 인덱스를 하나 증가하고 들어 있는 데이터를 읽어옴  
    System.out.print(iter.next());  
}  
}  
}
```

다음 예제를 확인해 보자.

```
// com.human.ex.PersonArrayList.java  
package com.human.ex;  
import java.util.ArrayList;  
import java.util.Iterator;  
import com.human.dto.Person;  
public class PersonArrayList {  
    public static void main(String[] args) {  
        ArrayList<Person> personList = new ArrayList<>();  
        // Person 객체 추가  
        personList.add(new Person("Alice", 25));  
        personList.add(new Person("Bob", 30));  
        personList.add(new Person("Charlie", 22));  
        // 전체 목록 출력  
        System.out.println("ArrayList 목록:");  
        for (Person person : personList) {  
            System.out.println(person);  
        }  
  
        System.out.println("-----");
```

```
// 특정 위치에 데이터 추가
personList.add(1, new Person("David", 28));
// 수정된 목록 출력
System.out.println("수정된 목록:");
for (Person person : personList) {
    System.out.println(person);
}
System.out.println("-----");
// 특정 위치의 데이터 삭제
personList.remove(2);
// 삭제된 목록 출력
System.out.println("삭제된 목록:");
for (Person person : personList) {
    System.out.println(person);
}
System.out.println("-----");
// 데이터 수정
personList.set(0, new Person("Eva", 35));
// 수정된 목록 출력
System.out.println("수정된 목록:");
for (Person person : personList) {
    System.out.println(person);
}
System.out.println("-----");
// Iterator를 사용하여 전체 데이터 출력
Iterator<Person> iter = personList.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next());
}
}
```

---

## > 15. ArrayList를 이용한 은행프로그램

---

개선된 코드에 대한 설명은 다음과 같습니다.

변수 정리:

List를 사용하면 더 이상 필요하지 않은 totalUserCount 변수를 제거했습니다.

List 초기화:

BankUser 인스턴스를 저장할 ArrayList<BankUser> bankUserList = new ArrayList<BankUser>();를 초기화했습니다. 이 리스트는 사용자 데이터를 동적으로 처리할 것입니다. 동적이란? 배열처럼 크기가 고정되어 있지 않음을 의미한다.

사용자 초기화:

초기 사용자를 나타내기 위해 bankUserList에 샘플 BankUser 인스턴스를 추가했습니다.

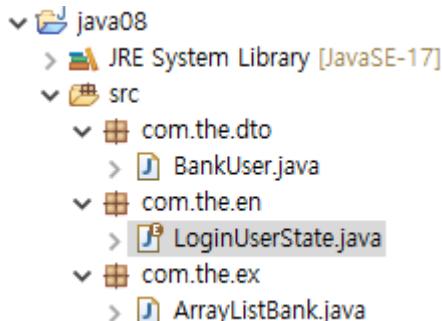
로그인 및 사용자 관리:

사용자 로그인, 추가, 삭제 및 수정에 대한 코드에서 배열 인덱싱을 List 메서드인 get(), size(), 및 add()로 대체했습니다.

totalUserCount를 수동으로 추적하는 대신 List의 size() 메서드를 사용했습니다.

다음은 결과코드인데 확인전에 스스로 구현해 보고 구현이 어려우면 보고해 보자.

BankUser 클래스는 이전에 만든 클래스와 동일해서 제외하였다.



```
// com.the.en.LoginUserState.java
```

```
package com.the.en;
//프로그램 진행 상태 enum
public enum LoginUserState {
```

```

LOGOUT,           //로그아웃 상태
USER_LOGIN, //유저로그인
ADMIN_LOGIN, //관리자로그인
EXIT      //프로그램 종료
}

//해당 프로그램은 4개중 반드시 하나의 상태를 가진다.

//com.the.dto.BankUser.java

package com.the.dto;

import java.util.Objects;
//클래스 필드 , static 필드 , 전역 필드
//static 유무에 따라 클래스 필드와 인스턴스 필드로 구분된다.
//클래스 필드 : 1개만 생성되고 전역변수 클래스이름으로 접근 new와 관련이 없다.
//인스턴스 필드 : new를 이용해서 원하는 만큼 생성, 인스턴스를 통해서 접근 가능
public class BankUser {
    //클래스 필드 예 : 전체 은행고객수, 해당 은행이름
    private String id="noData";
    private String pw="noData";
    private double account=0;
    //4가지 만들어보자. 생성자,getter setter,toString>equals

    //기본생성자 개발자가 생성자에 손대면 컴파일러가 자동으로 등록하지 않는다.
    public BankUser() {
        this("u1", "u1", 0); //자기 자신의 다른 생성자 호출
    }
    public BankUser(String id, String pw, double account) {
        super(); //부모의 기본 생성자를 호출한다.
        this.id = id;
        this.pw = pw;
        this.account = account;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getPw() {
        return pw;
    }
}

```

```
public void setPw(String pw) {
    this.pw = pw;
}
public double getAccount() {
    return account;
}
public void setAccount(double account) {
    this.account = account;
}
@Override
public String toString() {
    return "UserBank [id=" + id + ", pw=" + pw + ", account=" + account + "]";
}
@Override
public int hashCode() {
    return Objects.hash(account, id, pw);
}
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    BankUser other = (BankUser) obj;
    return Objects.equals(id, other.id) && Objects.equals(pw, other.pw);
}
}
```

*com.human.ex.ArrayListBank.java*

```
package com.the.ex;

import java.util.ArrayList;
import java.util.Scanner;
import com.the.dto.BankUser;
import com.the.en.LoginUserState;
public class ArrayListBank {
```

```

// arrayList를 이용한 은행프로그램
// 1. 상태값 enum : 프로그램 진행 상태값저장
// LoginUserState
public static LoginUserState LoginUserState =LoginUserState.LOGOUT;

//ArrayList에서는 해당 변수 arrayList에 들어있는 size가 전체 개수가 된다.
//public static int totalUserCount=0;

//public static int loginUserID=-1;
//여러개의 배열의 값을 접근할때는 index로 접근하는 것이 유리했지만
//arrayList에서는 찾은 객체를 담아서 처리하는 것이 유리하다.
//UserBank 은행 사용자 계정 정보
public static BankUser LoginUser=null;
//admin 사용자 계정정보
public static BankUser adminUser=new BankUser("admin","1111",0);

//은행 사용자 유저 관리 arrayList
public static ArrayList<BankUser> bankUserList=new ArrayList<BankUser>();

//사용자 입력 처리를 위한 변수들
public static Scanner sc =new Scanner(System.in);
public static String inputId=null;
public static String inputPw=null;
public static double inputAccount=0;

public static void main(String[] args) {
    //3명의 은행사용자 등록
    //ArrayList<BankUser> bankUserList=new ArrayList<BankUser>();
    bankUserList.add(new BankUser("user1","user1",0));
    bankUserList.add(new BankUser("user2","user2",0));
    bankUserList.add(new BankUser("user3","user3",0));
    bankUserList.size(); //전체 유저 개수 3

//은행 프로그램 메인 메뉴 구현
//은행 프로그램은 무한 반복하는데 사용자가 종료를 원하면 종료한다.
//LoginUserState값이 LoginUserState.EXIT로그아웃을 원하는 상태가 아닌동안 반복
while(!LoginUserState.equals(LoginUserState.EXIT)) {
    LoginUserState=LoginUserState.LOGOUT;
    //사용자로부터 id,pw 입력
    System.out.println("id 입력 종료를 원하면 exit입력 >");
    ArrayListBank.inputId=sc.nextLine();

```

```

System.out.println("pw 입력 >>");
ArrayListBank.inputPw=sc.nextLine();
//사용자로 부터 정보를 입력한다음 할일
//1.EXIT 종료를 원하는지 확인
//2.USER_LOGIN 인지 확인
//3.ADMIN_LOGIN 인지 확인
//4.사용자가 선택한 현재 프로그램 상태에 따른 하위메뉴 실행

//1.종료를 원하는지 확인 종료를 원하면 상태값을 EXIT로 변경
if(inputId.equals("exit")) {
    loginUserState=LoginUserState.EXIT;
    break;
}

//2.USER_LOGIN 인지 확인
// for (int i = 0; i < bankUserList.size(); i++) {
//     if (bankUserList.get(i).getId().equals(inputId)) {
//         if (bankUserList.get(i).getPw().equals(inputPw)) {
//             //forEach는 읽기모드에서만 사용가능
//             //로그인 실패 메시지 작성을 위한 flag
// boolean isFlag=true;
// for(BankUser bankUser:bankUserList) {
//     //로그인 성공시 실행되는 if문
//     if(bankUser.equals(new BankUser(inputId,inputPw,0))) {
//         System.out.println(bankUser+"님 로그인 하셨습니다.");
//         loginUserState=LoginUserState.USER_LOGIN;
//         ArrayListBank.LoginUser=bankUser;
//         isFlag=false;
//         break;
//     }
// }
// if(isFlag) {
//     System.out.println("없는 유저입니다. 관리자로 확인해 보겠습니다.");
// }

//3.ADMIN_LOGIN 인지 확인
//유저 로그인 실패했을 경우에만 실행되는 if문
if(!LoginUserState.equals(LoginUserState.USER_LOGIN)) {
    if(ArrayListBank.adminUser.equals(new BankUser(inputId,inputPw,0))){
        System.out.println("관리자로그인 성공");
        loginUserState=LoginUserState.ADMIN_LOGIN;
    }
}

```

```

}else {
    System.out.println("관리자 로그인 실패 다시 입력 해주세요>>");
}
}

//4. 사용자가 선택한 현재 프로그램 상태에 따른 하위메뉴 실행
switch(LoginUserState) {
case ADMIN_LOGIN:
    //////////////아래 부분에 관리자로 로그인 했을때 실행되는 코드가 있다.///////////
    break;
case USER_LOGIN:
    //////////////아래 부분에 사용자로 로그인 했을때 실행되는 코드가 있다.///////////
    break;
case LOGOUT:
    System.out.println("로그인 작업 실패 다시 시도해 주세요");
    break;
case EXIT:
    System.out.println("프로그램 종료");
    break;
default:
    System.out.println("알수 없는 프로그램 상태");
    LoginUserState=LoginUserState.EXIT;
    break;
}
}

}//while종료

```

```

        System.out.println("프로그램 종료");
    }
}

////// "관리자 메뉴 작업" 관련 소스
boolean isAdminMenu=true;
while(isAdminMenu) {
    System.out.println("관리자 메뉴 작업");
    System.out.println("1.계정추가 2.계정삭제 3.id,pw변경 "
        + "4.모든 사용자 출력 5.종료 >>");
    switch(sc.nextLine()) {
    case "1":
        System.out.println("추가할 id 입력 >>");
        ArrayListBank.inputId=sc.nextLine();
        System.out.println("추가할 pw 입력 >>");
        ArrayListBank.inputPw=sc.nextLine();
    }
}

```

```

        ArrayListBank.bankUserList.add(new BankUser(inputId, inputPw, 0));
break;
case "2":
    System.out.println("삭제할 id 입력 >>");
    ArrayListBank.inputId=sc.nextLine();
    System.out.println("삭제할 pw 입력 >>");
    ArrayListBank.inputPw=sc.nextLine();
ArrayListBank.bankUserList.remove(new BankUser(inputId, inputPw, 0));
break;
case "3":
    BankUser preBankUser;//변경전 유저 객체
    BankUser nextBankUser;//변경후 유저 객체

    System.out.println("변경전 id 입력 >>");
    ArrayListBank.inputId=sc.nextLine();
    System.out.println("변경전 pw 입력 >>");
    ArrayListBank.inputPw=sc.nextLine();
    preBankUser=new BankUser(inputId, inputPw, 0);

    System.out.println("변경후 id 입력 >>");
    ArrayListBank.inputId=sc.nextLine();
    System.out.println("변경후 pw 입력 >>");
    ArrayListBank.inputPw=sc.nextLine();
nextBankUser=new BankUser(inputId, inputPw, preBankUser.getAccount());

//set(변경전 index, 변경할내용)
if(ArrayListBank.bankUserList.contains(preBankUser)) {
    //변경전 index
    int preUserIndex=ArrayListBank.bankUserList.indexOf(preBankUser);
    ArrayListBank.bankUserList.set(preUserIndex, nextBankUser);
} else {
    System.out.println("변경전 유저 정보를 잘못 입력했습니다.");
}
break;
case "4":
    System.out.println("모든 사용자 출력");
    for(BankUser item:ArrayListBank.bankUserList) {
        System.out.println(item);
    }
break;
case "5":

```

```
    isAdmMenu=false;
    System.out.println("관리자 메뉴를 종료 합니다.");
break;
default:
    isAdmMenu=false;
    System.out.println("알수 없는 선택으로 메뉴를 종료합니다.");
break;
}
}//while문 종료
```

```
////// "사용자 메뉴 작업" 관련 소스
boolean isUseMenu=true;
while(isUseMenu) {
    System.out.println("사용자 메뉴 작업");
    System.out.println("1.입금 2.출금 3.잔액조회 4.종료 >>");
    switch(sc.nextLine()) {
        case "1":
            System.out.println("입금액 입력>>");
            ArrayListBank.inputAccount=Double.parseDouble(sc.nextLine());
            //입력받은 금액을 로그인한 계정에 추가한다.
            ArrayListBank.LoginUser.setAccount(
                ArrayListBank.LoginUser.getAccount()+ArrayListBank.inputAccount);
            System.out.println(ArrayListBank.LoginUser);
            break;
        case "2":
            System.out.println("출금액 입력>>");
            ArrayListBank.inputAccount=Double.parseDouble(sc.nextLine());
            //입력받은 금액을 로그인한 계정에 추가한다.
            ArrayListBank.LoginUser.setAccount(
                ArrayListBank.LoginUser.getAccount()-ArrayListBank.inputAccount);
            System.out.println(ArrayListBank.LoginUser);
            //ArrayListBank.loginUser.getId()
            //ArrayListBank.loginUser.getPw()
            //ArrayListBank.loginUser.getAccount()
            break;
        case "3":
            System.out.println("사용자 잔액정보");
            System.out.println(ArrayListBank.LoginUser);
            break;
        case "4":
```

```
    System.out.println("사용자 메뉴를 종료");
    isUseMenu=false;
    break;
default:
}
}//while문 종료
```

## > 03. 은행 프로그램 메소드로 구현하기

```
//은행초기데이터 3명의 계좌 정보입력
public static void init() {
//사용자나 관리자로 로그인 하기
public static void login() {
//사용자가 사용할 메뉴
public static void userMenu() {
//관리자가 사용할 메뉴
public static void adminMenu() {
//은행프로그램 시작메소드
public static void playBank() {
public static void main(String[] args) {
    init();
    playBank();
    System.out.println("프로그램 종료");
}
```

왼쪽은 필요한 메소드를 구현한 것이다. 주석을 확인해서 무엇을 하는 메소드인지 확인해 보자.

메인에 있는 메소드를 순서대로 실행하며 프로그램이 동작하니 메인에 있는 메소드를 순서대로 확인해 보자.

```
public class BankArrayList {
    public static State state =State.LOGIN_LOG_OF;
    public static int totalUserCount=3;
    public static String inputId="";
    public static String inputPw="";
    public static double inputAccount=0;
    public static int loginIndex =-1;
    public static String adminId="admin";
    public static String adminPw="1111";
    public static ArrayList<User> user =new ArrayList<User>();
    public static Scanner sc= new Scanner(System.in);
    public static void main(String[] args) {
        init();
        playBank();
        System.out.println("프로그램 종료");
    }
    public static void init() {
        user.add(new User("user1","user1",1000));
        user.add(new User("user2","user2",2000));
        user.add(new User("user3","user3",3000));
    }
    public static void login() {
        while (state == State.LOGIN_LOG_OF) {
            System.out.println("Id>>");
            inputId = sc.nextLine();
            System.out.println("Pw>>");
            inputPw = sc.nextLine();
```

```

        boolean isFindId = false;
        for (int i = 0; i < totalUserCount; i++) {
            if (user.get(i).getId().equals(inputId) &&
user.get(i).getPw().equals(inputPw)) {
                System.out.println(user.get(i).toString() + "님 로그인
하셨습니다.");
                state = State.LOGIN_USER;
                loginIndex = i;
                isFindId = true;
                break;
            }
        }

        if (!isFindId) {
            if (adminId.equals(inputId) && adminPw.equals(inputPw)) {
                System.out.println("관리자로 로그인 하였습니다.");
                state = State.LOGIN_ADMIN;
                break;
            } else {
                System.out.println("일치하지 않는 정보입니다.");
            }
        }
    }
}

public static void userMenu() {
    boolean isUseMenu=true;
    double userAccount=user.get(loginIndex).getAccount();
    while(isUseMenu) {
        System.out.println("1.입금 2.출금 3.정보확인 4.종료 /n
입력>>");

        switch(sc.nextLine()) {
        case "1":
            System.out.println("입금액 입력>>");
            inputAccount= Double.parseDouble(sc.nextLine());
            userAccount +=inputAccount;
            user.get(loginIndex).setAccount(userAccount);
            System.out.println(user.get(loginIndex).toString());
            break;
        case "2":
            System.out.println("출금액 입력>>");
            inputAccount= Double.parseDouble(sc.nextLine());
            if(user.get(loginIndex).getAccount()>=inputAccount) {
                userAccount -=inputAccount;
                user.get(loginIndex).setAccount(userAccount);

System.out.println(user.get(loginIndex).toString());
                break;
            }
        }
    }
}

```

```

        }else {
            System.out.println("잔액이 부족합니다.");
        }
        break;
    case "3":
        System.out.println("정보확인");
        System.out.println(user.getLoginIndex().toString());
        break;
    case "4":
        System.out.println("사용자 메뉴 종료");
        state=State.LOGIN_LOG_OF;
        isUseMenu=false;
        break;
    }
}
}

public static void adminMenu() {
    System.out.println("관리자 메뉴");
    boolean isUseMenu=true;
    while(isUseMenu) {
        System.out.println("1.계정추가 2.계정삭제 3.id변경
4.모든사용자출력 5.종료");
        switch(sc.nextLine()) {
            case "1":
                System.out.println("추가할 Id>>");
                inputId=sc.nextLine();
                System.out.println("추가할 Pw>>");
                inputPw=sc.nextLine();
                System.out.println("초기잔액 account>>");
                inputAccount=Double.parseDouble(sc.nextLine());
                user.add(new User(inputId,inputPw,inputAccount));

                System.out.println(user.getTotalUserCount().toString());
                totalUserCount++;
                break;
            case "2":
                System.out.println("삭제할 Id>>");
                inputId=sc.nextLine();
                int findIndex =totalUserCount;
                for(int i=0; i<totalUserCount; i++) {
                    if(user.get(i).getId().equals(inputId)) {
                        System.out.println("비밀번호를
입력하시오");

                        inputPw=sc.nextLine();
                        if(user.get(i).getPw().equals(inputPw)) {
                            findIndex=i;
                        }else {

```

```

        System.out.println("일치하지
않습니다.");
    }
}
}

System.out.println(user.get(findIndex).toString()+"해당 아이디를
삭제하였습니다.");
    user.remove(findIndex);
    totalUserCount--;
    break;
case "3":
    System.out.println("변경할 Id>>");
    inputId=sc.nextLine();
    for(int i=0; i<totalUserCount; i++) {
        if(user.get(i).getId().equals(inputId)) {
            System.out.println("Id>>");
            inputId=sc.nextLine();
            System.out.println("Pw>>");
            inputPw=sc.nextLine();
            user.set(i,new
User(inputId,inputPw,user.get(i).getAccount()));
        }
        System.out.println(user.get(i).toString()+"로 변경되었습니다.");
        break;
    }
    break;
case "4":
    System.out.println("모든사용자출력");
    for(int i=0; i<totalUserCount; i++) {
        System.out.println((i+1)+" 번째 고객:
"+user.get(i).toString());
    }
    break;
case "5":
    System.out.println("관리자 메뉴 종료");
    state=State.LOGIN_LOG_OF;
    isUseMenu=false;
    break;
}
}
}

public static void playBank() {
    boolean isUseMain=true;
    while(isUseMain) {
        login();

```

```

        switch(state) {
            case LOGIN_LOG_OF:
                break;
            case LOGIN_USER:
                userMenu();
                break;
            case LOGIN_ADMIN:
                adminMenu();
                break;
            default:

        }
        System.out.println("종료하시겠습니까? yes or no");
        if(sc.nextLine().equals("yes")) {
            isUseMain=false;
        }
    }
}

package com.human.ex;
public enum State {
    LOGIN_LOG_OF,
    LOGIN_USER,
    LOGIN_ADMIN
}

package com.human.ex;
import java.util.Objects;
public class User {
    private String id="";
    private String pw="";
    private double account=0;
    public User(){
    }
    public User(String id, String pw, double account) {
        this.id = id;
        this.pw = pw;
        this.account = account;
    }
    public User(String id, String pw) {
        this.id = id;
        this.pw = pw;
    }
    @Override
    public String toString() {
        return "User [id="+id+", pw="+pw + ", account=" + account + "]";
    }
}

```

```
}

@Override
public int hashCode() {
    return Objects.hash(account, id, pw);
}
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    User other = (User) obj;
    return Double.doubleToLongBits(account) ==
Double.doubleToLongBits(other.account)
        && Objects.equals(id, other.id) && Objects.equals(pw,
other.pw);
}
public String getId() {
    return id;
}
public void setId(String id) {
    this.id = id;
}
public String getPw() {
    return pw;
}
public void setPw(String pw) {
    this.pw = pw;
}
public double getAccount() {
    return account;
}
public void setAccount(double account) {
    this.account = account;
}
}
```

지역변수에 대해서 생각해 보자.

{ }(중괄호) 블록 내부에 선언되어 특정 블록 안에서만 사용할 수 있는 변수를 지역 변수라 한다. 그 동안 우리가 사용하고 있던 변수들이 지역변수에 해당한다.

즉, 바깥 블록에서 선언된 변수는 안쪽에서 접근할 수 있지만 안쪽에서 선언된 변수는 바깥쪽에서 사용할 수 없다.

다음 코드를 확인해 보자.

```
{  
    int a=1;  
  
    {  
        int b=2;  
        a=10; //안쪽 블록에서는 바깥쪽 블록에 접근 가능  
    }  
    b=10; //바깥쪽 블록에서는 안쪽 블록에 접근 불가능  
}
```

지역변수는 선언된 지역에 중괄호가 닫히면 메모리에서 사라져서 더 이상 접근할 수 없게

```
public class TestClass { // 1번 블록  
    public static void main(String[] args) { // 2번 블록  
        int a = 0;  
        if (true) { // 3번 블록  
            // 지역변수는 같은 메소드 안에서는 같은 이름으로 선언할 수 없다.  
            // 다른 메소드 안에서는 같은 이름으로 선언할 수 있다.  
            // int a = 10;  
            int b = 10;  
            if (true) { // 4번 블록  
                // 역변수는 자기보다 상위 블록에 접근할 수 있다.  
                int c = 0;  
                a++; b++; c++;  
            }  
            if (true) { // 4번 블록  
                // 지역변수는 자기보다 상위 블록에 접근할 수 있다.  
                // 변수 c와 같이 동등 레밸의 다른 블록에 선언된 변수에는 접근할 수 없다.  
                a++; b++; // c++;  
            }  
            System.out.println(a);  
            System.out.println(b);  
            // System.out.println(c);  
        }  
        System.out.println(a);  
        // 안쪽 블록에서 선언된 변수는 내부에서 사용할 수 없다.  
        // System.out.println(b);  
        if (true) { // 5번 블록  
            a++; // b++; c++ // 현재 블록이나 상위 블록에 b, c가 없어서 접근할 수 없다.  
        }  
    }  
}
```

1번 블록은 선언된 클래스를 묶는 블록이다.

2번 블록은 메소드를 묶은 블록이다. 이 블록 안에 선언된 변수는 해당 블록에서 사용할 수 있는 지역 변수이다. 함수의 매개 변수도 지역변수에 해당한다.

3번 블록, 5번 블록 처럼 if문 블록 안에 선언된 변수도 해당 블록의 지역 변수가 되고 for문 블록 안에 선언된 변수도 지역변수가 된다. 각각 해당 블록에서만 사용할 수 있다.

2개의 4번 블록 처럼 블록 안에 블록을 지속적으로 만들 수 있다.

args, a , b, c 가 지역변수에 해당한다. 이중에서도 args와 a는 블록의 가장 바깥쪽에서 선언되어 있어서 블록 안의 모든 종괄호 블록 안에서 사용할 수 있지만 지역 변수이므로 2번 블록 바깥쪽에서는 사용할 수 없다. 안쪽 블록 3,4,5에서 선언된 변수는 바깥쪽 2번 블록에서 사용할 수 없다.

```
2 public class TestClass2 {  
3     public static int add(int a, int b) {  
4         int sum = a + b;  
5         return sum;  
6     }  
7     public static void main(String[] args) {  
8         int result = 0;  
9         int num1 = 10;  
10        result = add(num1, 30);  
11        System.out.println(result);  
12    }  
13 }
```

다음 코드에 있는 모든 지역변수를 기술 해 보자. a, b, sum, args, result, num1

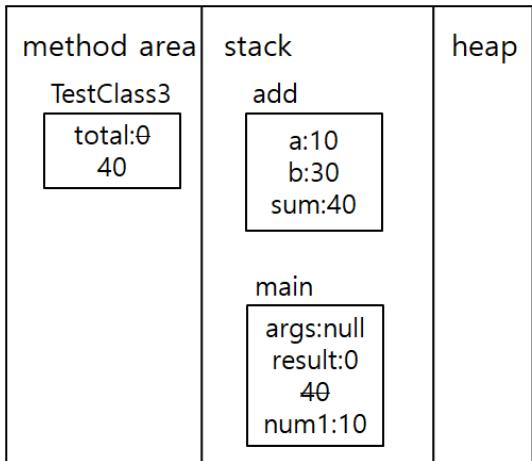
클래스를 구성하는 멤버로 클래스 필드, 클래스 메소드, 인스턴스 필드, 인스턴스 메소드가 있다.

```
1 public class TestClass3 {  
2     public static int total = 0; //클래스변수  
3     public static int add(int a, int b) {  
4         int sum = a + b;  
5         total = total + sum;//클래스변수  
6         return sum;  
7     }  
8     public static void main(String[] args) {  
9         int result = 0;  
10        int num1 = 10;  
11        result = add(num1, 30);  
12        System.out.println(result);  
13        System.out.println(total);//클래스변수  
14        System.out.println(TestClass3.total);//클래스변수  
15    }  
16 }
```

클래스에 선언된 변수를 필드, 함수를 메소드라 한다.  
클래스 멤버 중 클래스 필드와 클래스 메소드를 확인해 보자. 왼쪽에 public static이 붙은 필드와 메소드가 클래스 필드와 클래스 메소드이다. total이 클래스 필드이고

`add`가 클래스 메소드이다. 클래스 멤버는 지역변수와 달리 모든 지역에서 접근 가능하다.

클래스 필드는 힙, 스택이 아닌 메모리 영역에 저장 된다.



지역 변수는 해당 블록이 닫혀 실행을 종료할 때까지 생명주기를 유지 하지만 클래스 멤버는 프로그램이 실행될 때 메모리에 올라가 프로그램이 종료 될 때까지 메모리에 남아 있고 지역변수와 달리 모든 지역에서 접근 가능 하다.

클래스 필드는 선언된 ‘클래스명.변수명’으로 접근할 수 있다. 하지만, 같은 클래스 안에서는 `클래스명.변수명`을 생략할 수 있다. 상위 코드에서 `TestClass3.total`를 `total`로 줄여쓴 것이다. 사용 방법은 일반 변수 처럼 사용하면 된다.

클래스 메소드도 마찬가지로 클래스이름.을

생략할 수 있다. `static`이 붙은 클래스 멤버는 모든 지역에서 사용할 수 있어 전역 필드, 전역 메소드라 부르기도 한다. `static` 필드, `static` 메소드라 부르기도 한다. 되도록 클래스명을 생략하지 말고 ‘클래스명.변수명’으로 기술하여 사용하자.

메모리영역의 메소드영역은 `static`으로 선언된 전역 변수와 클래스 관련 코드를 관리하는 메모리 영역이다. 스택 영역은 메소드안에 선언된 지역변수를 관리하는 메모리 영역이다. 힙은 `new`로 할당된 인스턴스와 상수풀에서 관리되는 상수들을 관리하는 메모리 영역이다.

메소드의 용도는 다음과 같다. 클래스 메소드는 보통 클래스 필드를 조작하는 용도로 사용하고 클래스 메소드 사용 시점에 인스턴스 필드는 생성되어 있지 않아서 접근할 수 없다. 인스턴스 메소드는 보통 인스턴스 필드를 조작하기 위해서 사용하고 클래스 필드는 모든 지역에서 사용할 수 있어서 인스턴스 필드에서도 사용할 수 있다.

다음 예제를 확인해 보자.

```
// MyClass.java 파일
public class MyClass {
    public static int classField = 0; // 클래스 필드
    public int instanceField; // 인스턴스 필드
    // 클래스 메소드
    public static void classMethod() {
        classField++; // 클래스 필드 조작, 인스턴스 필드는 접근할 수 없다.
        System.out.println("클래스 변수 값: " + classField);
    }
    // 인스턴스 메소드
    public void instanceMethod() {
        instanceField++; // 인스턴스 필드 조작, 클래스 필드를 접근할 수 있다.
        System.out.println("인스턴스 변수 값: " + instanceField);
    }
    // 생성자
    public MyClass(int instanceField) {
        this.instanceField = instanceField;
    }
}
// MainClass.java 파일
public class MainClass {
    public static void main(String[] args) {

        MyClass.classField = 50;           // 클래스 필드 접근 방법
        MyClass.classMethod();           // MyClass의 클래스 메소드 호출

        MyClass myInstance1 = new MyClass(5);
        myInstance1.instanceField=50; // 인스턴스 필드 접근 방법

        // MyClass의 인스턴스 생성 및 인스턴스 메소드 호출
        myInstance1.instanceMethod();
        /*인스턴스 필드는 new로 인스턴스가 생성될 때마다 고유한 메모리 공간을
        할당받아 값을 저장합니다. 이에 따라 "인스턴스.인스턴스필드"를 통해 생성된 여러
        인스턴스 중 원하는 인스턴스의 위치에 값을 저장할 수 있습니다.

        인스턴스 메소드는 "인스턴스.메소드()" 형식으로 호출되며, 실행 시 해당 메소드가
        속한 인스턴스의 인스턴스 필드 값을 참조합니다. 따라서 호출된 인스턴스가 가지고 있는
        필드에 따라 메소드에서 접근하는 인스턴스 필드가 달라지게 됩니다.
    */
        MyClass myInstance2 = new MyClass(10);
```

```
myInstance2.instanceMethod();  
  
    MyClass.classMethod();  
}  
}
```

인스턴스 메소드는 인스턴스 필드를 조작하는 용도로 사용된다.

인스턴스 필드는 객체가 생성될 때마다 새로운 메모리 공간에 할당되기 때문에 각 인스턴스마다 고유한 값을 가질 수 있습니다. 인스턴스 메소드를 호출할 때는 해당 메소드를 호출하는 인스턴스를 기준으로 합니다. 즉, "인스턴스.메소드()" 형태로 실행되며, 이때 인스턴스 메소드에서 실행되는 인스턴스 필드는 해당 메소드를 호출한 인스턴스의 필드에 접근합니다. 이를 통해 인스턴스 메소드가 각각의 서로 다른 인스턴스 필드에 접근 할 수 있습니다.

클래스 메소드는 클래스 필드를 조작하는 용도로 사용된다. 클래스 필드는 보통 클래스를 대표한 저장 공간 하나를 가진다. 클래스 메소드도 인스턴스와 관련없이 클래스를 대표하는 메소드가 필요할 때 사용한다.

해당 인스턴스 개수를 생성하는 메소드는 해당 클래스의 인스턴스 개수와 관련이 있지만, 인스턴스 각각에 대해서는 관계가 없어 클래스 메소드로 만든다.

클래스 메소드는 클래스이름으로 모든 지역에서 접근할 수 있어 보통 전역 메소드라 하고, 인스턴스 메소드는 전역, 지역 관계없이 인스턴스에 접근 할 수 있을 때만 접근할 수 있다.

,

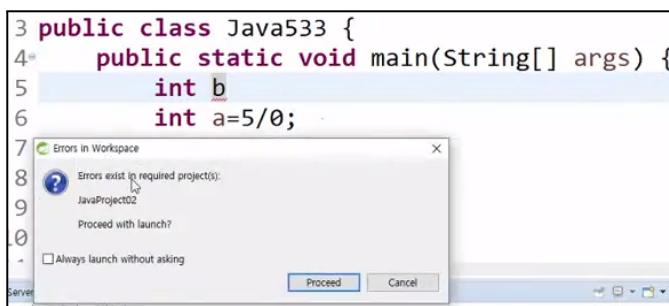
## > 16. 예외처리

오류란? 무엇인가 잘못되었다는 이야기이다. 자바에서 대표적인 오류에는 컴파일 오류와 런타임 오류가 있다.

컴파일 오류는 대부분 문법 오류여서 오류가 발생하면 프로그램을 실행시킬 수 없고 코드를 수정하여 프로그램을 실행시킬 수 있다.

런타임 오류는 프로그램이 실행 도중에 프로그램에 문제가 발생해 프로그램이 중지되는 오류이다.

예외처리란? 프로그램 동작중 문제가 발생하면 프로그램이 종료되는데 프로그램이 종료되지 않고 작업을 이어 나갈 수 있도록 하는 작업을 의미한다.



코드에 문법 에러가 있는 경우 왼쪽 이미지 박스처럼 에러가 나서 메시지 박스가 출력 된다. 컴파일 오류이다.

다음 이미지 코드처럼 문법적 오류가 없는 경우 프로그램을 실행 시키면 에러 메시지 박스없이 실행된다. 문법 적인 오류가 없기 때문이다.

```
public static void main(String[] args) {  
  
    int a=5/0;  
  
    System.out.println("프로그램 종료");  
  
}
```

A screenshot of the Eclipse IDE interface. On the left, there's a code editor window with Java code. On the right, a 'Console' window is open, showing the output of the program's execution. The output text is: 'ion in thread "main" java.lang.ArithmeticException at com.human.ex3.Java533.main(Java533.java:'. This indicates that an exception was caught and printed to the console.

하지만 콘솔창을 확인해보면 “프로그램 종료”라는 메시지는 없고 arithmeticException이라는 빨간색 예외 메시지가 나온 것을 확인할 수 있다. 문법적 에러가 없어 실행 되었으나 실행중 예외가 발생하여 프로그램이 종료된 것이다.

수학적으로 0으로 다른 수를 나눌 수 없다. 5/0 부분에서 프로그램 문법 보다는 수학적 예외 발생으로 프로그램이

멈춰서 “프로그램 종료”라는 메시지가 출력되지 않았다.

```
try {  
    int a=5/0;  
}catch(Exception e) {  
    e.printStackTrace();  
}  
System.out.println("프로그램 종료");
```

왼쪽은 예외처리를 한 코드이다.

기술하고 실행해 보면 코드가 실행되어 “프로그램 종료”라는 메시지가 출력된 것을 확인할 수 있다.

이렇게 프로그램이 문법적 에러가 없지만 예외적인 상황에 위해서 프로그램이 종료되면 사용자가 프로그램을 사용 하는데 불편해 질 것이다. 이런 문제를 해결하기 위해서

프로그램이 멈추지 않도록 예외처리를 할 수 있게 자바에서는 `try catch finally`문을 제공 한다.

사용자 입력을 받는 프로그램에서 사용자가 숫자를 입력해야 할 곳에 문자를 입력하게 되면 프로그램에 문제가 발생하고 프로그램이 멈춘다. 예외처리를 통해서 이런 상황에서 프로그램이 멈추지 않고 지속적으로 동작하게 할 수 있다.

런타임 오류는 실행 중에 프로그램이 오류를 만나 프로그램이 종료되는 것을 의미한다. 런타임 오류는 문법적으로는 문제가 없어서 프로그램이 실행되지만 사용자의 잘못된 입력 예를 들자면 숫자를 입력해야 되는데 문자를 입력한 다든지 배열의 인덱스를 벗어난 다든지 대부분 사용자 입력이 잘못 되었을 때 실행 도중 프로그램이 종료되는 경우를 이야기 한다. 이런 경우에 예외 처리(exception)를 이용해서 잘못된 부분을 올바르게 고쳐 프로그램이 종료되지 않고 계속 동작하게 만드는 것이 목적이다. 예외처리 사용되는 방법은 `try ~ catch ~ finally` 문을 이용해서 한다. `try ~ catch ~ finally` 문의 모양은 다음과 같다.

```
try{  
    //예외 발생 가능한 지점  
}catch(처리할예외클래스 변수명){  
    // 예외처리할 내용  
}catch (처리할예외클래스 변수명){  
    //처리할 내용  
}finally{  
    // 예외 발생 유무와 관계없이 항상 실행하는 코드 블럭  
}
```

`catch`문은 `switch`문 `case` 처럼 여러개 정의 할 수 있다.

`try`문의 중괄호 안에 예외가 발생할 수 있는 코드를 작성하고 예외가 발생하면 해당 `catch`문에서 예외를 처리한다.

발생한 예외 클래스와 동일한 `catch`문의 클래스를 찾아 해당 `catch`문의 매개 변수에 예외 클래스 인스턴스가 생성되고 이 인스턴스를 사용하여 해당 `catch`문에서 여러가지 작업을 진행한다.

`finally`부분에는 예외가 발생하든 발생하지 않든 반드시 실행해야 할 코드를 기술한다. 예외가 발생하면 예외 발생 아래 부분 코드나 특정 `catch`문에 기술한 코드는 실행되지 않을 수 있다. 어떠한 경우에라도 반드시 실행 해야 할 코드가 있다면 이곳에 기술하면 된다.

`finally`문은 코드가 없으면 생략할 수 있다.

다음 이미지의 4번 라인을 보면 사용자가 안녕을 입력하여 안녕이라는 문자를 숫자로 변경하고 있다. 만약 사용자 입력이라면 사용자가 숫자를 입력할지 문자를 입력할지 알 수 없는 일이여서 실행 시점에는 문법적으로 문제가 없어 컴파일되어 실행 되지만 사용자가 입력을 잘못 하게 되면 오류가 발생 될 수 있는 부분이다. 컴파일 시점에 `Integer.parseInt`의 매개 변수가 문자열이면 문제없이 실행되지만 실행중 해당 문자열을 숫자로 변경할 수 없으면 예외 메시지와 함께 프로그램이 중단된다. 한마디로 컴파일 시점에는 문제가 없지만

실행도중 문제가 발생한다는 이야기이다.

이런 부분에 예외 처리를 해서 사용자의 잘못된 입력에 대해서 프로그램이 종료 되지 않도록 만들어야 한다. 예외가 발생할 수 있는 부분을 try{} 안에 넣고 예외가 발생하면 switch 문처럼 해당 클래스가 정의된 catch 문만 실행 된다.

The screenshot shows the Eclipse IDE interface with the code editor and the console output. The code in the editor is:

```
1 public class TestClass8 {
2     public static void main(String[] args) {
3         try {
4             int num = Integer.parseInt("안녕");
5             System.out.println("정상 동작시만 출력");
6         } catch (NumberFormatException e) {
7             System.out.println(e);
8             e.printStackTrace();
9         } catch (Exception e) {
10            System.out.println(e);
11            e.printStackTrace();
12        }finally {
13            System.out.println("예외 관계없이 항상 출력");
14        }
15    }
16 }
```

The console output shows the following error message:

```
<terminated> TestClass8 [Java Application] D:\#2020eclipseSpring\#eclipseSpring\#common\#java\#jre1.8.0_201\#bin\#javaw.exe (2020. 1. 11. 오전
java.lang.NumberFormatException: For input string: "안녕"
java.lang.NumberFormatException: For input string: "안녕"
    at java.lang.NumberFormatException.forInputString(Num
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at TestClass8.main(TestClass8.java:4)
```

4번 라인에서 예외가 발생해서 5번 라인의 코드는 접근 할 수 없는 코드가 된다.

4번라인에서 NumberFormat... 예외가 발생해서 6번라인으로 점포한다.

13번라인은 항상 출력되는 코드이다.

어떤 종류의 예외가 존재하는지 확인해

보면 다음과 같다. 문자를 숫자로 바꿀 수 없을 때 발생하는 예외 클래스는 NumberFormatException 클래스 여서 try문 안에서 예외가 발생하면 catch (NumberFormatException e) { 의 catch 블록으로 이동해 실행을 이어 나간다. 만약 try 문 안에서 배열 인덱스의 범위를 벗어나는 인덱스를 배열에 사용하였을 경우 int []a=new int[5]; a[6]=1;과 같은 경우 ArrayIndexOutOfBoundsException 이 발생하고 관련 예외를 처리 하고 싶다면 catch문에 다음과 같은 catch(ArrayIndexOutOfBoundsException e) { catch 문을 추가 하여 처리하면 된다. 없는 클래스에 접근 하려 할 때는 ClassNotFoundException 예외가 발생하고, 10/0과 같이 정수를 0으로 나누면 ArithmeticException 예외가 발생한다. switch문에서 default처럼 Exception클래스는 모든 예외를 받아서 처리하는 클래스이다. Exception 클래스를 사용하면 모든 예외를 받아 처리할 수 있다. 더 많은 예외 클래스는 인터넷에서 찾아서 확인해 보자. e는 예외가 발생하면 생긴 예외 클래스의 인스턴스가 들어간다.

System.out.println(e); 의 경우는 예외 내용을 문자열로 찍으라는 이야기이고 e.printStackTrace();의 경우는 예외 내용뿐 아니라 예외가 발생한 함수들의 호출 관계를 모두 찍는 메소드이다. 예외 발생 지역 및 예외 내용을 확인 할 수 있다.

try문은 일반적으로 코드가 위에서 아래로 실행 되다가 예외가 발생함과 동시에 해당 catch 문으로 실행 순서가 이동한다. 따라서 5번 라인 같은 경우에는 4번 라인에서 예외가 발생하면 동작하지 못하는 코드가 된다. 프로그램을 구현 하다 보면 예외가 발생 여부와 관계없이 반드시 실행 돼야 할 코드가 있다. 이때 사용하는 구문이 finally 구문 이고 finally구문 안에 기술한 코드는 예외 발생 유무와 관계없이 반드시 실행이 된다.

catch문에서 catch (Exception e) {} 과 catch (NumberFormatException e) {} 블럭의 위치가 바뀐다면 모든 예외가 catch (Exception e) {} 에 선택되므로 이후 어떤 기술된 catch문도 접근할 수 없는 코드가 된다. catch (Exception e) {} 문은 switch문의 default: 문처럼 마지막에 기술해야 한다. Exception클래스가 중간에 기술되면 이후 기술된 예외 클래스에 접근할 수 없으므로 기술하면 에러가 발생한다.

다음 코드를 확인해보자.

```
int []arr= {1,2,3,4,5};  
int index=10;  
java.util.Scanner sc=new java.util.Scanner(System.in);  
while(true) {  
    try {  
        System.out.println("출력하고싶은 배열의 인덱스를 입력>>");  
        index=Integer.parseInt(sc.nextLine());  
        System.out.println(arr[index]);  
        break;  
    }catch(ArrayIndexOutOfBoundsException e) {  
        System.out.println("잘못된 인덱스를 입력하였습니다.");  
    }catch(NumberFormatException e) {  
        System.out.println("숫자를 입력하였습니다.");  
    }catch(Exception e) {//e.printStackTrace();  
        System.out.println("알수없는예외 발생");  
    }  
}
```

```
출력하고싶은 배열의 인덱스를 입력>>  
ㅁㄴㅇ  
숫자를 입력하세요.  
출력하고싶은 배열의 인덱스를 입력>>  
12  
잘못된 인덱스를 입력하였습니다.  
출력하고싶은 배열의 인덱스를 입력>>  
5  
잘못된 인덱스를 입력하였습니다.  
출력하고싶은 배열의 인덱스를 입력>>  
1  
2  
프로그램종료
```

상위 코드는 왼쪽처럼 사용자가 잘못 입력하였을 경우에 처리 되도록 구현된 코드이다. 왼쪽 처럼 잘못된 입력을 확인해보자. eclipse에서 실행 할 때 한글로 입력하면 문자열 처리에 문제가 발생하는 버그가 있다. 영어로 입력하거나 방향키를 잘 사용해서 입력해 보자.

다음 코드는 수를 0으로 나누었을 때 발생하는 예외처리 방법이다.

```
try { // 0으로 수를 나눌 수 없어서 ArithmeticException이 발생한다.
    int a = 4 / 0;
} catch (ArithmeticException e) {
}
```

다음 문제를 풀어보자.

1. 다음 코드의 실행 결과를 출력해보자.

```
package com.human.ex;

public class MyExceptioni {
    public static void main(String[] args) {
        for(int i=3;i>=-3;i--) {
            try {
                int a=5/i;
                // 동작 할 수도 있고 안 할 수도 있다. 예외가 발생하면 동작하지 않음
                System.out.println(i);
            }catch(ArithmeticException e) {
                e.printStackTrace();
                System.out.println("0으로 나눌 수 없습니다.");
            }catch(Exception e) {
                e.printStackTrace();
                System.out.println("알 수 없는 예외");
            }finally {
                System.out.println(i+"작업을 완료하였습니다."); // 반드시 실행된다. 예외가
                발생해도 실행 않아도 실행
            }
        }
    }
}
```

```
}
```

```
}
```

```
}
```

```
1 public class TestClass9 {
2     public static void main(String[] args) {
3         try {
4             //throw new Exception();
5             throw new NumberFormatException();
6         } catch (NumberFormatException e) {
7             System.out.println(e);
8             e.printStackTrace();
9         } catch (Exception e) {
10            System.out.println(e);
11            e.printStackTrace();
12        } finally {
13            System.out.println("예외 관계없이 항상 출력");
14        }
15    }
16 }
```

일반적으로 예외는 프로그램이 동작중 어쩔수 없는 경우 스스로 발생시키지만 프로그래머가 throw 문을 사용하여 강제로 예외를 발생시킬 수 있다. 5번 라인처럼 기술하면 NumberFormatException이 발생하고 해당 예외의 처리부분의 catch 문으로 이동하므로 7,8번 라인이

```
int a=15;

try {
    if(a>10) {
        //강제로 예외 발생
        throw new Exception();
    }else {
        a=a+5;
    }
}catch(Exception e) {
    System.out.println(
        "a가 10보다 커서 예외발생");
}

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

실행된다.

만약 수학적 예외를 강제로 만들고 싶다면 throw new ArithmeticException();이라고 기술하면 된다.

왼쪽에 예제는 a가 10보다 큰경우 강제로 예외를 발생하는 프로그램을 구현해 본것이다.

throws는 메소드 안의 코드중 예외가 발생하 였을때 예외처리를 해당 메소드에서 하지 않고 해당 메소드를 호출한 부분에서 예외처리를 넘기는 방법이다.

다음 코드를 통해서 throws를 이해해 보자.

```

public static void exceptionFunction1() throws Exception{
    throw new Exception();
}
public static void exceptionFunction2() throws Exception{
    throw new NumberFormatException();
}
public static void main(String[] args) throws Exception,NumberFormatException {
    try{
        exceptionFunction1();
    }catch(Exception e) {};
    exceptionFunction2();
}

```

exceptionFunction1() 안에서 throw new Exception();를 사용하여 예외가 발생하면 프로그램이 종료되기 때문에 예외 처리를 해주어야 한다. 하지만 상위 프로그램에서 예외처리를 하지 않았는데 프로그램이 멈추지 않고 잘 동작하는 것을 확인 할 수 있다. throws 다음에 예외처리가 필요한 예외클래스를 기술하면 메소드 안에서 해야할 예외처리를 해당 메소드를 호출한 부분에서 할 수 있다.

메인 안에서 try catch문으로 exceptionFunction1() 메소드 안에서 발생 할 수 있는 예외를 메소드를 호출한 곳에서 처리 한 것을 확인 할 수 있다.

main 메소드의 throws 다음에 기술한 예외 클래스도 마찬 가지로 main를 호출한 부분에서 처리 한다. main은 운영체제가 호출한 것이니 운영체제가 알아서 처리 한다고 생각하면 된다.

main 메소드의 throws 의 기술로 exceptionFunction2()에서 발생한 예외를 main를 호출한 운영체제가 알아서 처리한다.

```

3 public class Java534 {
4     public static int test1(int a) {
5         try {
6             if(a>10) {
7                 throw new Exception();
8             }else {
9                 a=a+5;
10            }
11         }catch(Exception e) {
12             System.out.println("a가 10보다 커서 예외발생");
13             a=0;
14         }
15         return a;
16     }

```

다음 이미지들은 이전 예제를 메소드로 만들어 throw와 throws 예제를 만든 것이다. test1 메소드의 경우 메소드 내부에서 예외처리가 되어서 호출만 하면 되는 상태이고 test2는 메소드 안에서 예외처리를 하지 않고 throws를

이용해서 호출한 곳에서 예외처리를 하도록 한 경우이다.

다음 코드를 확인해 보자.

```
17  public static int test2(int a) throws Exception {  
18      if(a>10) {  
19          throw new Exception();  
20      }else {  
21          a=a+5;  
22      }  
23      return a;  
24  }  
25  public static void main(String[] args) throws Exception {  
26      System.out.println(test1(5));  
27      try {  
28          System.out.println(test2(5));  
29      } catch (Exception e) {  
30          e.printStackTrace();  
31      }  
32  }  
33 }
```