4. 생성자

#1.인강/0.자바/2.자바-기본

생성자 - 필요한 이유

객체를 생성하는 시점에 어떤 작업을 하고 싶다면 생성자(Constructor)를 이용하면 된다.

생성자를 알아보기 전에 먼저 생성자가 왜 필요한지 코드로 간단히 알아보자.

MemberInit

```
package construct;

public class MemberInit {
    String name;
    int age;
    int grade;
}
```

MethodInitMain1

```
package construct;

public class MethodInitMain1 {
    public static void main(String[] args) {
        MemberInit member1 = new MemberInit();
        member1.name = "user1";
        member1.age = 15;
        member1.grade = 90;

        MemberInit member2 = new MemberInit();
        member2.name = "user2";
```

```
        member2.age = 16;
        member2.grade = 80;

        MemberInit[] members = {member1, member2};

        for (MemberInit s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
        }
    }
}
```

실행 결과

```
이름:user1 나이:15 성적:90
이름:user2 나이:16 성적:80
```

회원 객체를 생성하고 나면 `name`, `age`, `grade` 같은 변수에 초기값을 설정한다. 아마도 회원 객체를 제대로 사용하기 위해서는 객체를 생성하자 마자 이런 초기값을 설정해야 할 것이다. 이 코드에는 회원의 초기값을 설정하는 부분이 계속 반복된다. 메서드를 사용해서 반복을 제거해보자.

```
package construct;

public class MethodInitMain2 {
    public static void main(String[] args) {
        MemberInit member1 = new MemberInit();
        initMember(member1, "user1", 15, 90);

        MemberInit member2 = new MemberInit();
        initMember(member2, "user2", 16, 80);

        MemberInit[] members = {member1, member2};

        for (MemberInit s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
        }

    }

    static void initMember(MemberInit member, String name, int age, int grade) {
        member.name = name;
```

```
        member.age = age;
        member.grade = grade;
    }
}
```

`initMember(...)` 메서드를 사용해서 반복을 제거했다. 그런데 이 메서드는 대부분 `MemberInit` 객체의 멤버 변수를 사용한다. 우리는 앞서 객체 지향에 대해서 학습했다. 이런 경우 속성과 기능을 한 곳에 두는 것이 더 나은 방법이다. 쉽게 이야기해서 `MemberInit`이 자기 자신의 데이터를 변경하는 기능(메서드)을 제공하는 것이 좋다.

this

MemberInit - initMember() 추가

```
package construct;

public class MemberInit {
    String name;
    int age;
    int grade;

    //추가
    void initMember(String name, int age, int grade) {
        this.name = name;
        this.age = age;
        this.grade = grade;
    }
}
```

```
package construct;

public class MethodInitMain3 {
    public static void main(String[] args) {
        MemberInit member1 = new MemberInit();
        member1.initMember("user1", 15, 90);

        MemberInit member2 = new MemberInit();
        member2.initMember("user2", 16, 80);
```

```
        MemberInit[] members = {member1, member2};

        for (MemberInit s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
 s.grade);
        }
    }
}
```

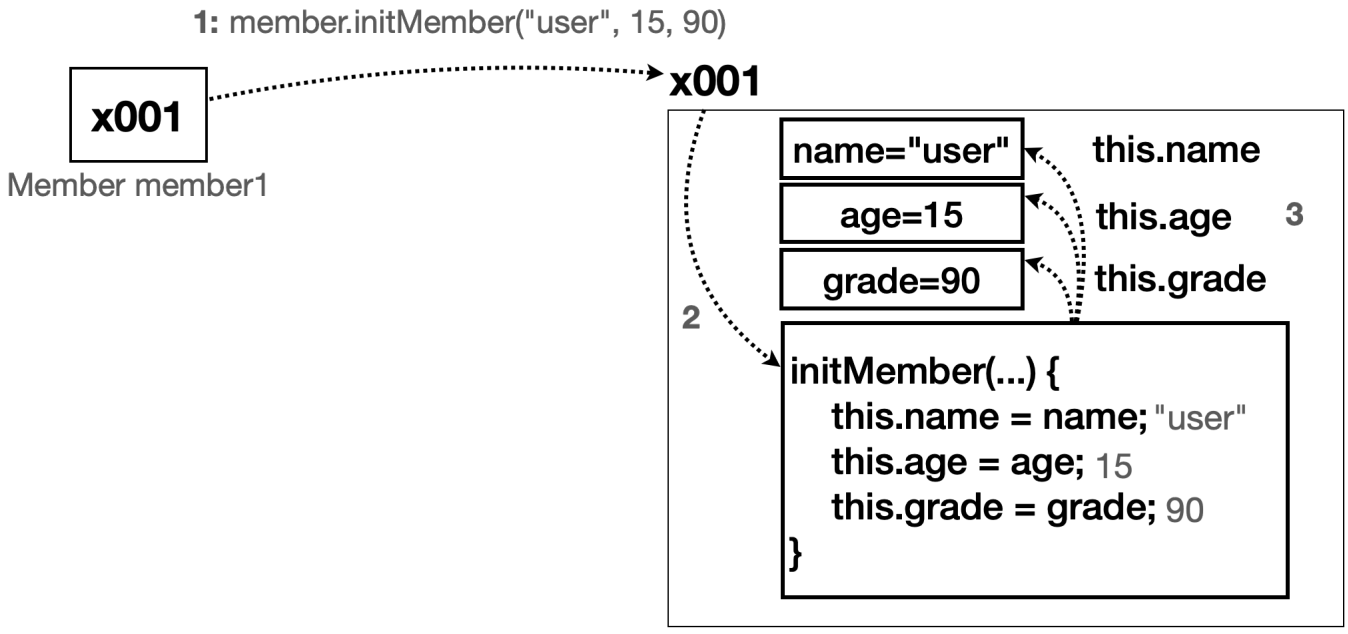
`initMember(...)`는 `Member`에 초기값 설정 기능을 제공하는 메서드이다.
다음과 같이 메서드를 호출하면 객체의 멤버 변수에 인자로 넘어온 값을 채운다.
`member1.initMember("user1", 15, 90)`

this

`Member`의 코드를 다시 보자
`initMember(String name...)`의 코드를 보면 메서드의 매개변수에 정의한 `String name`과 `Member`의 멤버 변수의 이름이 `String name`으로 둘다 똑같다. 나머지 변수 이름도 `name`, `age`, `grade`로 모두 같다.

멤버 변수와 메서드의 매개변수의 이름이 같으면 둘을 어떻게 구분해야 할까?

- 이 경우 멤버 변수보다 매개변수가 코드 블럭의 더 안쪽에 있기 때문에 **매개변수가 우선순위**를 가진다. 따라서 `initMember(String name,...)` 메서드 안에서 `name`이라고 적으면 매개변수에 접근하게 된다.
- 멤버 변수에 접근하려면 앞에 `this.`이라고 해주면 된다. 여기서 `this`는 인스턴스 자신의 참조값을 가리킨다.



Member 인스턴스

진행 과정

```
this.name = name; //1. 오른쪽의 name은 매개변수에 접근
this.name = "user"; //2. name 매개변수의 값 사용
x001.name = "user"; //3. this.은 인스턴스 자신의 참조값을 뜻함, 따라서 인스턴스의 멤버 변수에 접근
```

this 제거

만약 이 예제에서 `this`를 제거하면 어떻게 될까?

```
this.name = name
```

다음과 같이 수정하면 `name`은 둘다 매개변수를 뜻하게 된다. 따라서 맴버변수의 값이 변경되지 않는다.

```
name = name
```

정리

- 매개변수의 이름과 맴버 변수의 이름이 같은 경우 `this`를 사용해서 둘을 명확하게 구분해야 한다.
- `this`는 인스턴스 자신을 가리킨다.

this의 생략

`this`는 생략할 수 있다. 이 경우 변수를 찾을 때 가까운 지역변수(매개변수도 지역변수다)를 먼저 찾고 없으면 그 다음으로 멤버 변수를 찾는다. 멤버 변수도 없으면 오류가 발생한다.

다음 예제는 필드 이름과 매개변수의 이름이 서로 다르다.

```
package construct;

public class MemberThis {
    String nameField;

    void initMember(String nameParameter) {
        nameField = nameParameter;
    }
}
```

예를 들어서 `nameField`는 앞에 `this`가 없어도 멤버 변수에 접근한다.

- `nameField`는 먼저 지역변수(매개변수)에서 같은 이름이 있는지 찾는다. 이 경우 없으므로 맴버 변수에서 찾는다.
- `nameParameter`는 먼저 지역변수(매개변수)에서 같은 이름이 있는지 찾는다. 이 경우 매개변수가 있으므로 매개변수를 사용한다.

this와 코딩 스타일

다음과 같이 멤버 변수에 접근하는 경우에 항상 `this`를 사용하는 코딩 스타일도 있다.

```
package construct;

public class MemberThis {
    String nameField;

    void initMember(String nameParameter) {
        this.nameField = nameParameter;
    }
}
```

`this.nameField`를 보면 `this`를 생략할 수 있지만, 생략하지 않고 사용해도 된다.
이렇게 `this`를 사용하면 이 코드가 멤버 변수를 사용한다는 것을 눈으로 쉽게 확인할 수 있다. 그래서 과거에 이런 스타일을 많이 사용하기도 했다. 쉽게 이야기해서 `this`를 강제로 사용해서, 지역 변수(매개변수)와 멤버 변수를 눈에 보이도록 구분하는 것이다.
하지만 최근에 IDE가 발전하면서 IDE가 멤버 변수와 지역 변수를 색상으로 구분해준다.
다음을 보면 멤버 변수와 지역 변수의 색상이 다른 것을 확인할 수 있다.

```
no usages
public class MemberThis {
    1 usage
    public String nameField;

    no usages
    public void initMember(String nameParameter) {
        nameField = nameParameter;
    }
}
```

이런 점 때문에 `this`는 앞서 설명한 것 처럼 이름이 중복되는 것 처럼, 꼭 필요한 경우에만 사용해도 충분하다 생각한다.

생성자 - 도입

프로그래밍을 하다보면 객체를 생성하고 이후에 바로 초기값을 할당해야 하는 경우가 많다. 따라서 앞서 `initMember(...)`와 같은 메서드를 매번 만들어야 한다.
그래서 대부분의 객체 지향 언어는 객체를 생성하자마자 즉시 필요한 기능을 좀 더 편리하게 수행할 수 있도록 생성자라는 기능을 제공한다. 생성자를 사용하면 객체를 생성하는 시점에 즉시 필요한 기능을 수행할 수 있다.
생성자는 앞서 살펴본 `initMember(...)`와 같이 메서드와 유사하지만 몇가지 다른 특징이 있다.

기존 코드를 유지하기 위해 `MemberConstruct`라는 새로운 클래스를 작성하겠다.

MemberConstruct

```
package construct;

public class MemberConstruct {
    String name;
    int age;
    int grade;
```

```
    MemberConstruct(String name, int age, int grade) {
        System.out.println("생성자 호출 name=" + name + ",age=" + age + ",grade=" +
grade);
        this.name = name;
        this.age = age;
        this.grade = grade;
    }

}
```

다음 부분이 바로 생성자이다.

```
MemberConstruct(String name, int age, int grade) {
    System.out.println("생성자 호출 name=" + name + ",age=" + age + ",grade=" +
grade);
    this.name = name;
    this.age = age;
    this.grade = grade;
}
```

생성자는 메서드와 비슷하지만 다음과 같은 차이가 있다.

- 생성자의 이름은 클래스 이름과 같아야 한다. 따라서 첫 글자도 대문자로 시작한다.
- 생성자는 반환 타입이 없다. 비워두어야 한다.
- 나머지는 메서드와 같다.

```
package construct;

public class ConstructMain1 {
    public static void main(String[] args) {
        MemberConstruct member1 = new MemberConstruct("user1", 15, 90);
        MemberConstruct member2 = new MemberConstruct("user2", 16, 80);

        MemberConstruct[] members = {member1, member2};

        for (MemberConstruct s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
        }
    }
}
```

실행 결과

```
생성자 호출 name=user1,age=15,grade=90
생성자 호출 name=user2,age=16,grade=80
이름:user1 나이:15 성적:90
이름:user2 나이:16 성적:80
```

생성자 호출

생성자는 인스턴스를 생성하고 나서 즉시 호출된다. 생성자를 호출하는 방법은 다음 코드와 같이 `new` 명령어 다음에 생성자 이름과 매개변수에 맞추어 인수를 전달하면 된다.

```
new 생성자이름(생성자에 맞는 인수 목록)
new 클래스이름(생성자에 맞는 인수 목록)
```

참고로 생성자이름이 클래스 이름이기 때문에 둘다 맞는 표현이다.

```
new MemberConstruct("user1", 15, 90)
```

이렇게 하면 인스턴스를 생성하고 즉시 해당 생성자를 호출한다. 여기서는 `Member` 인스턴스를 생성하고 바로 `MemberConstruct(String name, int age, int grade)` 생성자를 호출한다.

참고로 `new` 키워드를 사용해서 객체를 생성할 때 마지막에 괄호 `()` 도 포함해야 하는 이유가 바로 생성자 때문이다. 객체를 생성하면서 동시에 생성자를 호출한다는 의미를 포함한다.

생성자 장점

중복 호출 제거

생성자가 없던 시절에는 생성 직후에 어떤 작업을 수행하기 위해 다음과 같이 메서드를 직접 한번 더 호출해야 했다. 생성자 덕분에 객체를 생성하면서 동시에 생성 직후에 필요한 작업을 한번에 처리할 수 있게 되었다.

```
//생성자 등장 전
MemberInit member = new MemberInit();
member.initMember("user1", 15, 90);

//생성자 등장 후
MemberConstruct member = new MemberConstruct("user1", 15, 90);
```

제약 - 생성자 호출 필수

방금 코드에서 생성자 등장 전 코드를 보자. 이 경우 `initMember(...)` 를 실수로 호출하지 않으면 어떻게 될까? 이 메서드를 실수로 호출하지 않아도 프로그램은 작동한다. 하지만 회원의 이름과 나이, 성적 데이터가 없는 상태로 프로그램이 동작하게 된다. 만약에 이 값들을 필수로 반드시 입력해야 한다면, 시스템에 큰 문제가 발생할 수 있다. 결국 아무 정보가 없는 유령 회원이 시스템 내부에 등장하게 된다.

생성자의 진짜 장점은 객체를 생성할 때 직접 정의한 생성자가 있다면 **직접 정의한 생성자를 반드시 호출**해야 한다는 점이다. 참고로 생성자를 메서드 오버로딩 처럼 여러개 정의할 수 있는데, 이 경우에는 하나만 호출하면 된다.

`MemberConstruct` 클래스의 경우 다음 생성자를 직접 정의했기 때문에 직접 정의한 생성자를 반드시 호출해야 한다.

```
MemberConstruct(String name, int age, int grade) {...}
```

다음과 같이 직접 정의한 생성자를 호출하지 않으면 컴파일 오류가 발생한다.

```
MemberConstruct member3 = new MemberConstruct(); //컴파일 오류 발생
member3.name = "user1";
```

컴파일 오류 메시지

```
no suitable constructor found for MemberConstruct(no arguments)
```

컴파일 오류는 IDE에서 즉시 확인할 수 있는 좋은 오류이다. 이 경우 개발자는 객체를 생성할 때, 직접 정의한 생성자를 필수로 호출해야 한다는 것을 바로 알 수 있다. 그래서 필요한 생성자를 찾아서 다음과 같이 호출할 것이다.

```
MemberConstruct member = new MemberConstruct("user1", 15, 90);
```

생성자 덕분에 회원의 이름, 나이, 성적은 항상 필수로 입력하게 된다. 따라서 아무 정보가 없는 유령 회원이 시스템 내부에 등장할 가능성을 원천 차단한다!

생성자를 사용하면 필수값 입력을 보장할 수 있다

> **참고:** 좋은 프로그램은 무한한 자유도가 주어지는 프로그램이 아니라 적절한 제약이 있는 프로그램이다.

기본 생성자

생각해보면 생성자를 만들지 않았는데, 생성자를 호출한 적이 있다. 다음 코드들을 다시 확인해보자.

```
public class MemberInit {
    String name;
    int age;
    int grade;
}
```

```
public class MethodInitMain1 {
    public static void main(String[] args) {
        MemberInit member1 = new MemberInit();
        ...
    }
}
```

여기서 `new MemberInit()` 이 부분은 분명히 매개변수가 없는 다음과 같은 생성자가 필요할 것이다.

```
public class MemberInit {
    String name;
    int age;
    int grade;

    MemberInit() { //생성자 필요
    }
}
```

기본 생성자

- 매개변수가 없는 생성자를 기본 생성자라 한다.
- 클래스에 생성자가 하나도 없으면 자바 컴파일러는 매개변수가 없고, 작동하는 코드가 없는 기본 생성자를 자동으로 만들어준다.
- **생성자가 하나라도 있으면 자바는 기본 생성자를 만들지 않는다.**

`MemberInit` 클래스의 경우 생성자를 만들지 않았으므로 자바가 자동으로 기본 생성자를 만들어준 것이다.

예제를 통해서 기본 생성자를 확인해보자.

MemberDefault

```
package construct;

public class MemberDefault {
    String name;
}
```

MemberDefaultMain

```
package construct;
```

```
public class MemberDefaultMain {

    public static void main(String[] args) {
        MemberDefault memberDefault = new MemberDefault();
    }
}
```

`MemberDefault` 클래스에는 생성자가 하나도 없으므로 자바는 자동으로 다음과 같은 기본 생성자를 만들어준다. (우리 눈에 보이지는 않는다.)

MemberDefault - 기본 생성자

```
package construct;

public class MemberDefault {
    String name;

    //기본 생성자
    public MemberDefault() {
    }
}
```

참고: 자바가 자동으로 생성해주시는 기본 생성자는 클래스와 같은 접근 제어자를 가진다. `public`은 뒤에 접근 제어자에서 자세히 설명한다.

물론 다음과 같이 기본 생성자를 직접 정의해도 된다.

```
package construct;

public class MemberDefault {
    String name;

    MemberDefault() {
        System.out.println("생성자 호출");
    }
}
```

실행 결과

```
생성자 호출
```

기본 생성자를 왜 자동으로 만들어줄까?

만약 자바에서 기본 생성자를 만들어주지 않는다면 생성자 기능이 필요하지 않은 경우에도 모든 클래스에 개발자가 직접 기본 생성자를 정의해야 한다. 생성자 기능을 사용하지 않는 경우도 많기 때문에 이런 편의 기능을 제공한다.

정리

- 생성자는 반드시 호출되어야 한다.
- 생성자가 없으면 기본 생성자가 제공된다.
- **생성자가 하나라도 있으면 기본 생성자가 제공되지 않는다.** 이 경우 개발자가 정의한 생성자를 직접 호출해야 한다.

생성자 - 오버로딩과 this()

생성자도 메서드 오버로딩처럼 매개변수만 다르게 해서 여러 생성자를 제공할 수 있다.

MemberConstruct - 생성자 추가

```
package construct;

public class MemberConstruct {
    String name;
    int age;
    int grade;

    //추가
    MemberConstruct(String name, int age) {
        this.name = name;
        this.age = age;
        this.grade = 50;
    }

    MemberConstruct(String name, int age, int grade) {
        System.out.println("생성자 호출 name=" + name + ",age=" + age + ",grade=" +
grade);
        this.name = name;
        this.age = age;
        this.grade = grade;
    }

}
```

기존 `MemberConstruct`에 생성자를 하나 추가해서 생성자가 2개가 되었다.

```
MemberConstruct(String name, int age)
MemberConstruct(String name, int age, int grade)
```

새로 추가한 생성자는 `grade`를 받지 않는다. 대신에 `grade`는 `50`점이 된다.

```
package construct;

public class ConstructMain2 {
    public static void main(String[] args) {
        MemberConstruct member1 = new MemberConstruct("user1", 15, 90);
        MemberConstruct member2 = new MemberConstruct("user2", 16);

        MemberConstruct[] members = {member1, member2};

        for (MemberConstruct s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
        }
    }
}
```

실행 결과

```
생성자 호출 name=user1,age=15,grade=90
이름:user1 나이:15 성적:90
이름:user2 나이:16 성적:50
```

생성자를 오버로딩 한 덕분에 성적 입력이 꼭 필요한 경우에는 `grade`가 있는 생성자를 호출하면 되고, 그렇지 않은 경우에는 `grade`가 없는 생성자를 호출하면 된다. `grade`가 없는 생성자를 호출하면 성적은 `50`점이 된다.

this()

두 생성자를 비교해 보면 코드가 중복 되는 부분이 있다.

```
public MemberConstruct(String name, int age) {
    this.name = name;
    this.age = age;
    this.grade = 50;
}
```

```
public MemberConstruct(String name, int age, int grade) {
    this.name = name;
    this.age = age;
    this.grade = grade;
}
```

바로 다음 부분이다.

```
this.name = name;
this.age = age;
```

이때 `this()` 라는 기능을 사용하면 생성자 내부에서 자신의 생성자를 호출할 수 있다. 참고로 `this` 는 인스턴스 자신의 참조값을 가리킨다. 그래서 자신의 생성자를 호출한다고 생각하면 된다.
코드를 다음과 같이 수정해보자.

MemberConstruct - this() 사용

```
package construct;

public class MemberConstruct {
    String name;
    int age;
    int grade;

    MemberConstruct(String name, int age) {
        this(name, age, 50); //변경
    }

    MemberConstruct(String name, int age, int grade) {
        System.out.println("생성자 호출 name=" + name + ",age=" + age + ",grade=" +
grade);
        this.name = name;
        this.age = age;
        this.grade = grade;
    }

}
```

이 코드는 첫번째 생성자 내부에서 두번째 생성자를 호출한다.

```
MemberConstruct(String name, int age) -> MemberConstruct(String name, int age,
int grade)
```

`this()`를 사용하면 생성자 내부에서 다른 생성자를 호출할 수 있다. 이 부분을 잘 활용하면 지금과 같이 중복을 제거할 수 있다. 물론 실행 결과는 기존과 같다.

this() 규칙

- `this()`는 생성자 코드의 첫줄에만 작성할 수 있다.

다음은 규칙 위반이다. 이 경우 컴파일 오류가 발생한다.

```
public MemberConstruct(String name, int age) {
    System.out.println("go");
    this(name, age, 50);
}
```

`this()`가 생성자 코드의 첫줄에 사용되지 않았다.

문제와 풀이

문제 - Book과 생성자

`BookMain` 코드가 작동하도록 `Book` 클래스를 완성하세요.
특히 `Book` 클래스의 생성자 코드에 중복이 없도록 주의하세요.

```
package construct.ex;

public class Book {
    String title; //제목
    String author; //저자
    int page; //페이지 수

    //TODO 코드를 완성하세요.
}
```

```
package construct.ex;

public class BookMain {
    public static void main(String[] args) {
        // 기본 생성자 사용
```

```
        Book book1 = new Book();
        book1.displayInfo();

        // title과 author만을 매개변수로 받는 생성자
        Book book2 = new Book("Hello Java", "Seo");
        book2.displayInfo();

        // 모든 필드를 매개변수로 받는 생성자
        Book book3 = new Book("JPA 프로그래밍", "kim", 700);
        book3.displayInfo();
    }
}
```

실행 결과

```
제목: , 저자: , 페이지: 0
제목: Hello Java, 저자: Seo, 페이지: 0
제목: JPA 프로그래밍, 저자: kim, 페이지: 700
```

정답

```
package construct.ex;

public class Book {
    String title;
    String author;
    int page;

    // 기본생성자
    Book() {
        this("", "", 0);
    }

    // title과 author만을 매개변수로 받는 생성자
    Book(String title, String author) {
        this(title, author, 0);
    }

    // 모든 필드를 매개변수로 받는 생성자
    Book(String title, String author, int page) {
        this.title = title;
        this.author = author;
        this.page = page;
```

```
    }

    void displayInfo() {
        System.out.println("제목: " + title + ", 저자: " + author + ", 페이지: " +
page);
    }

}
```

정리

생성자는 객체 생성 직후 객체를 초기화 하기 위한 특별한 메서드로 생각할 수 있다.