

상속

상속



학생의 추상화

값 (멤버 변수)

- 이름
- 나이
- 키
- 몸무게
- 학교
- 점수
- 동아리

기능 (메소드)

- 인사 하기
- 걷기
- 말하기
- 공부 하기
- 등교 하기

공통된 멤버 변수와 메소드 존재



선생님의 추상화

값 (멤버 변수)

- 이름
- 나이
- 키
- 몸무게
- 학교
- 과목
- 담당 학년
- 교무실

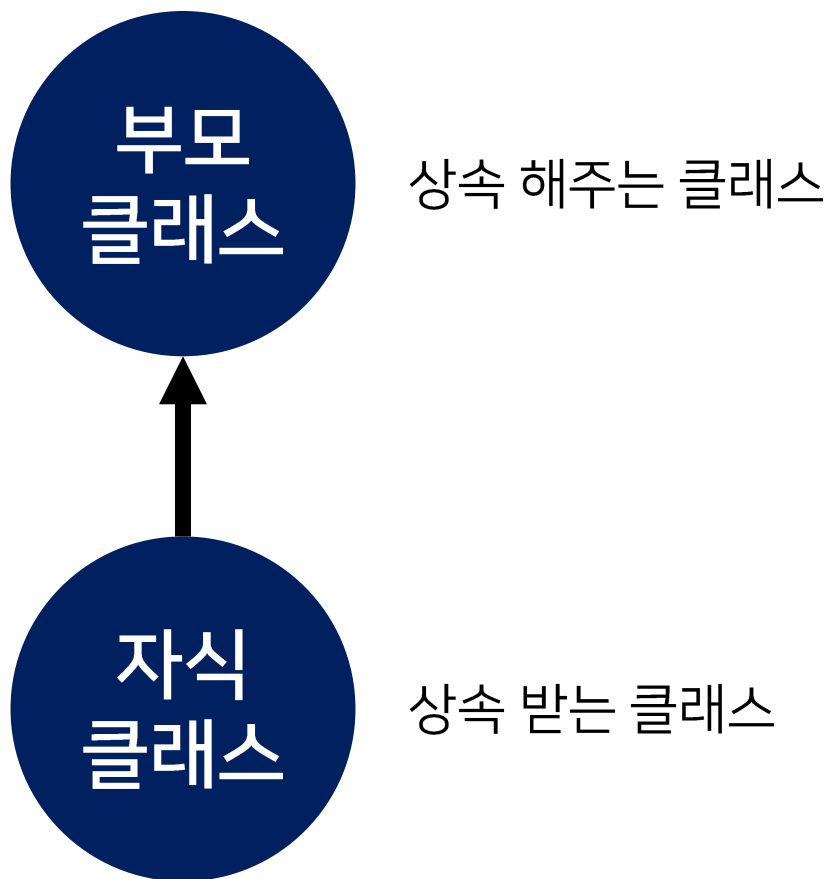
기능 (메소드)

- 인사 하기
- 걷기
- 말하기
- 정보 보여주기
- 수업하기
- 문제 내기

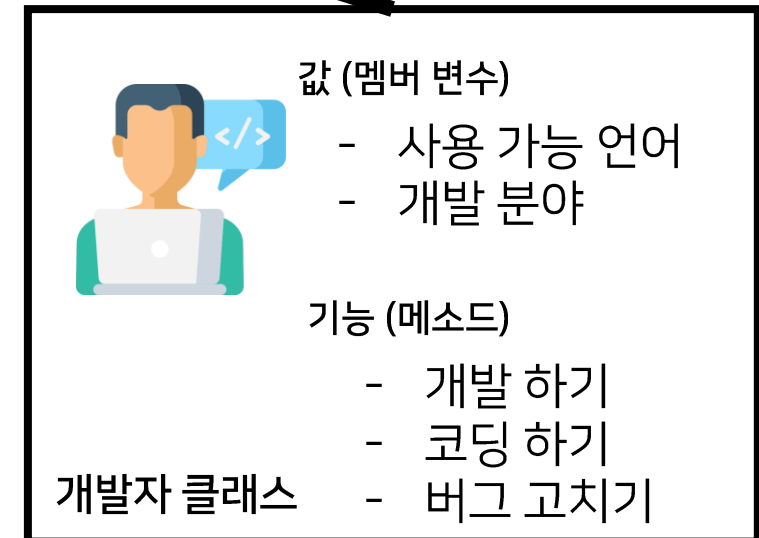
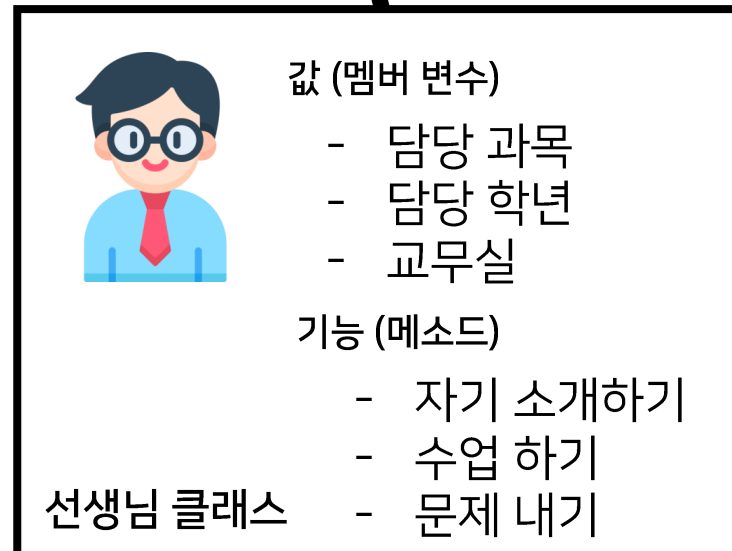
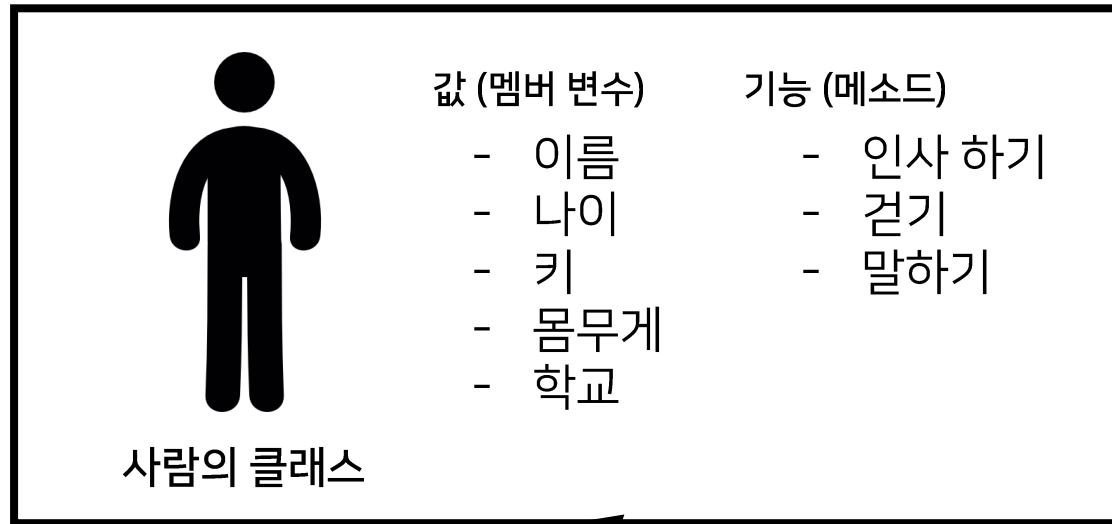
상속

상속 : 다른 클래스들의 멤버 변수와 메소드 들을 사용할 수 있게 한다.

Kotlin에서는 하나의 클래스만 상속 받을 수 있다.



상속



해당 클래스가
상속 할 수 있게 만든다.


```
open class Person(  
    val name : String,  
    val age : Int,  
    val height : Float,  
    val weight : Float,  
    val school : String  
) {  
    fun hello() = println("안녕하세요. ${name}입니다.")  
    fun walk() = println("$name : 걷기")  
    fun talk(message : String) = println("$name : $message")  
}
```

상속 받을 클래스는 : 뒤에 써준다.
이때 넘겨줘야 하는 인자들을
같이 넘겨줘야 한다.

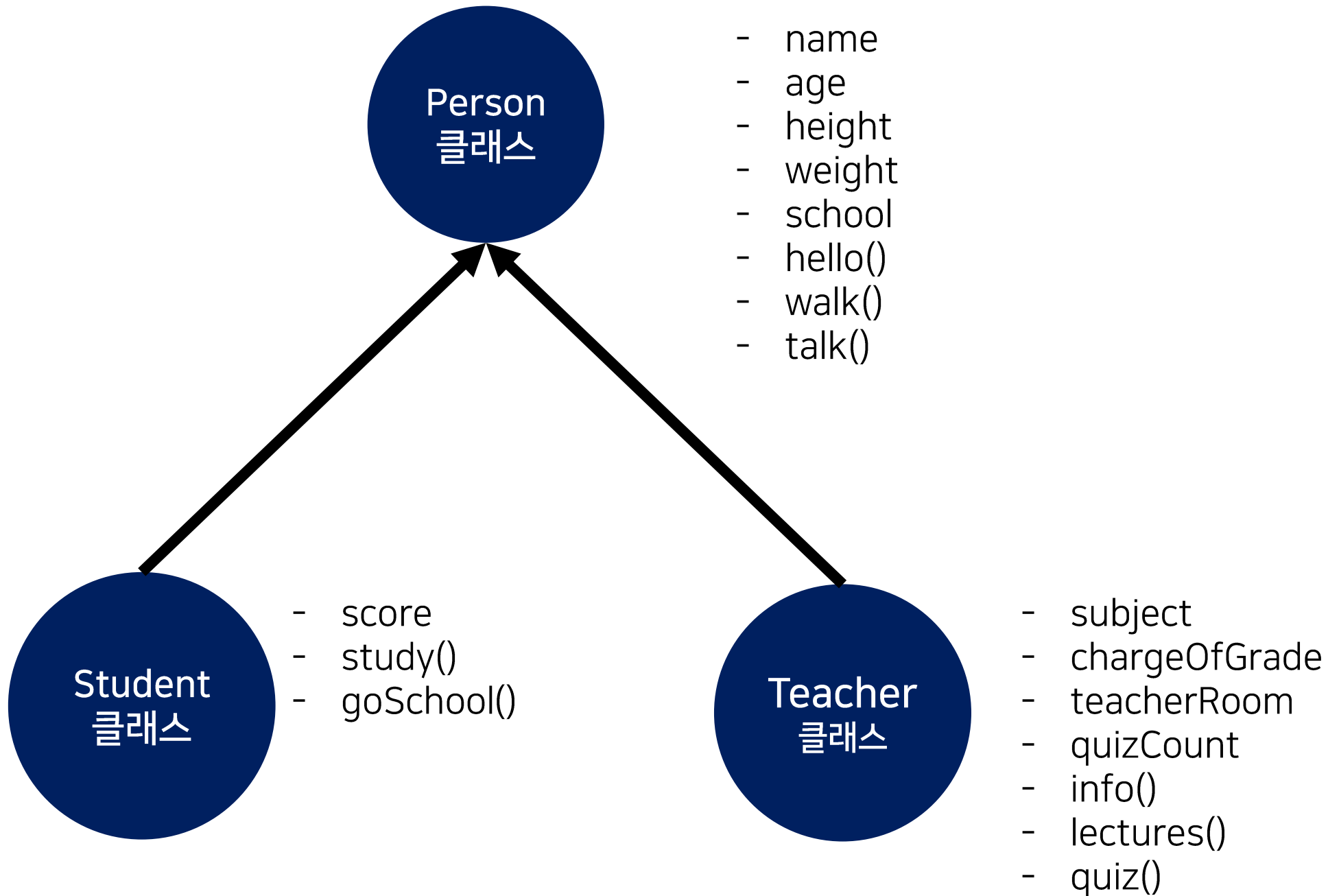
```
class Student(  
    name: String,  
    age: Int,  
    height: Float,  
    weight: Float,  
    school: String,  
) : Person(name, age, height, weight, school){  
    var score: Int = 0  
  
    fun study() {  
        println("$name : 공부 공부")  
        score += 10  
    }  
  
    fun goSchool() = println("$name : 등교 등교")  
}
```

상속 받는 클래스

```
class Teacher(  
    name: String,  
    age: Int,  
    height: Float,  
    weight: Float,  
    school: String,  
    val subject : String,  
    val chargeOfGrade : Int    상속 받는 클래스  
) : Person(name, age, height, weight, school){  
    lateinit var teacherRoom : String  
    var quizCount = 0;  
  
    fun info(){  
        println("이름 : $name")  
        println("나이 : $age")  
        println("담당 과목 : $subject ${chargeOfGrade}학년")  
        println("교무실 : $teacherRoom")  
    }  
  
    fun lectures(time : Int) = println("${name}선생님(${subject})이 ${time}교시 수업을 합니다.")  
  
    fun quiz(difficulty : String) = println("$subject ${++quizCount}번 문제 (난이도 : ${difficulty})")  
}
```



```
fun main() {  
    val s1 = Student("박희찬", 18, 178f, 70f, "선린인터넷고등학교")  
    val s2 = Student("장인수", 18, 200f, 70f, "선린인터넷고등학교")  
  
    val t1 = Teacher("심희원", 20, 160f, 50f, "선린인터넷고등학교", "프로그래밍", 1)  
    val t2 = Teacher("이왕렬", 40, 170f, 50f, "선린인터넷고등학교", "웹프로그래밍 실무", 2)  
  
    s1.name // 박희찬,  
    s1.age // 18  
    s1.hello()  
    s1.goSchool()  
  
    t1.name // 심희원,  
    t1.age // 20  
    t1.quiz("상")  
}
```

생성자 실행 순서는?

```
open class Parent(){
```

```
    init {
```

```
        println("부모 객체가 생성 되었습니다.")
```

```
    }
```

```
}
```

첫번째 실행

부모 클래스의 생성자 실행



```
class Child() : Parent(){
```

```
    init {
```

```
        println("자식 객체가 생성 되었습니다.")
```

```
    }
```

```
}
```

두번째 실행

```
fun main() {
```

```
    Child()
```

```
}
```

메소드 오버로딩

- 부모 클래스의 메소드의 이름과 받는 인자를 같지만 서로 다른 기능을 만들고 싶을 때

```
open class Person(val name : String, val age : Int){  
    open fun work(time : Int) {  
        println("${time}시간 동안 일을 합니다")  
    }  
}  
  
class Student(name: String, age: Int) : Person(name, age){  
    override fun work(time: Int) {  
        println("${time}시간 동안 공부를 합니다")  
    }  
}
```

메소드 오버로딩

- 부모 클래스의 메소드의 이름과 받는 인자를 같지만 서로 다른 기능을 만들고 싶을 때

```
open class Person(val name : String, val age : Int){  
    open fun work(time : Int) { 이름과 인자는 같다.  
        println("${time}시간 동안 일을 합니다")  
    }  
}  
  
class Student(name: String, age: Int) : Person(name, age){  
    override fun work(time: Int) {  
        println("${time}시간 동안 공부를 합니다")  
    }  
}
```

메소드 오버로딩

- 부모 클래스의 메소드의 이름과 받는 인자를 같지만 서로 다른 기능을 만들고 싶을 때

```
open class Person(val name : String, val age : Int){  
    open fun work(time : Int) {  
        println("${time}시간 동안 일을 합니다")  
    } 오버로딩 해줄 부모 메소드에 open을 붙여준다.  
}  
  
class Student(name: String, age: Int) : Person(name, age){  
    override fun work(time: Int) {  
        println("${time}시간 동안 공부를 합니다")  
    } 자식 메소드에 override을 붙여준다.  
}
```

특수 클래스

특수 클래스

- 특수한 목적을 위해서 사용되는 클래스

데이터
클래스

추상 클래스

인터페이스


데이터 클래스

- 데이터를 저장하기 위해서 사용하는 클래스

예시

- 유저 데이터를 저장하는 클래스,
- 동아리 데이터를 저장하는 클래스,
- 게시물 데이터를 저장하는 클래스

데이터 클래스



```
data class Board(  
    val title : String, //게시물 제목  
    val content : String, //게시물 내용  
    val writer : User, //작성자 정보  
){}  

```

게시물 정보를 저장하는 클래스

데이터 클래스

앞에 data를 붙여준다.

```
data class Board(  
    val title : String, //게시물 제목  
    val content : String, //게시물 내용  
    val writer : User, //작성자 정보  
){}  

```

게시물 정보를 저장하는 클래스

데이터 클래스

사용자 정보를 저장하는
데이터 클래스

```
data class User(  
    var id : String, //유저의 id  
    var name : String, //사용자 이름  
    var email : String, //사용자 이메일  
  
    var userMSG : String, //사용자의 소개 메시지  
    var profileImgUrl : String, //Storage에 저장된 사용 프로필 사진  
  
    var likeBoardList : ArrayList<String>,  
    //사용자가 좋아요 누른 게시물의 id  
  
    var totalLikeCount : Int, //사용자가 지금까지 받은 좋아요 개수  
)
```

데이터 클래스를 사용하는 이유

- toString(), equals(), hashCode() 메소드 등이 자동으로 만들어진다.

```
fun main() {  
    val boardData = Board("안녕하세요.", "대충 내용")  
    println(boardData.toString())  
}
```

Board(title=안녕하세요., content=대충 내용)

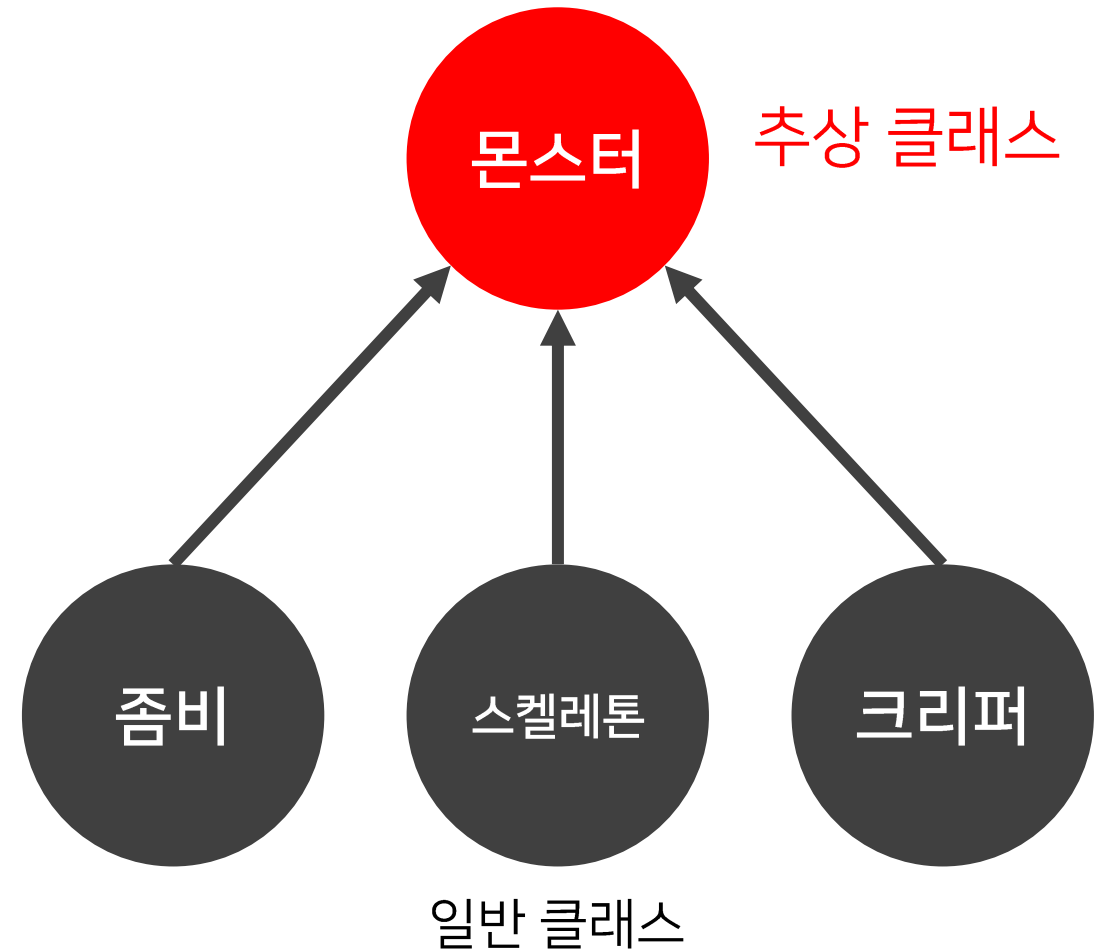
data class로 만들었을 때

Board@5b6f7412

일반 class로 만들었을 때

추상 클래스

- 추상화를 위해서 사용하는 클래스
- 직접 객체로 만들어 주지는 않는다.
- 추상클래스의 멤버 변수와 메소드는 선언만 해주고 상속받은 다음 구현 해준다.



추상 클래스

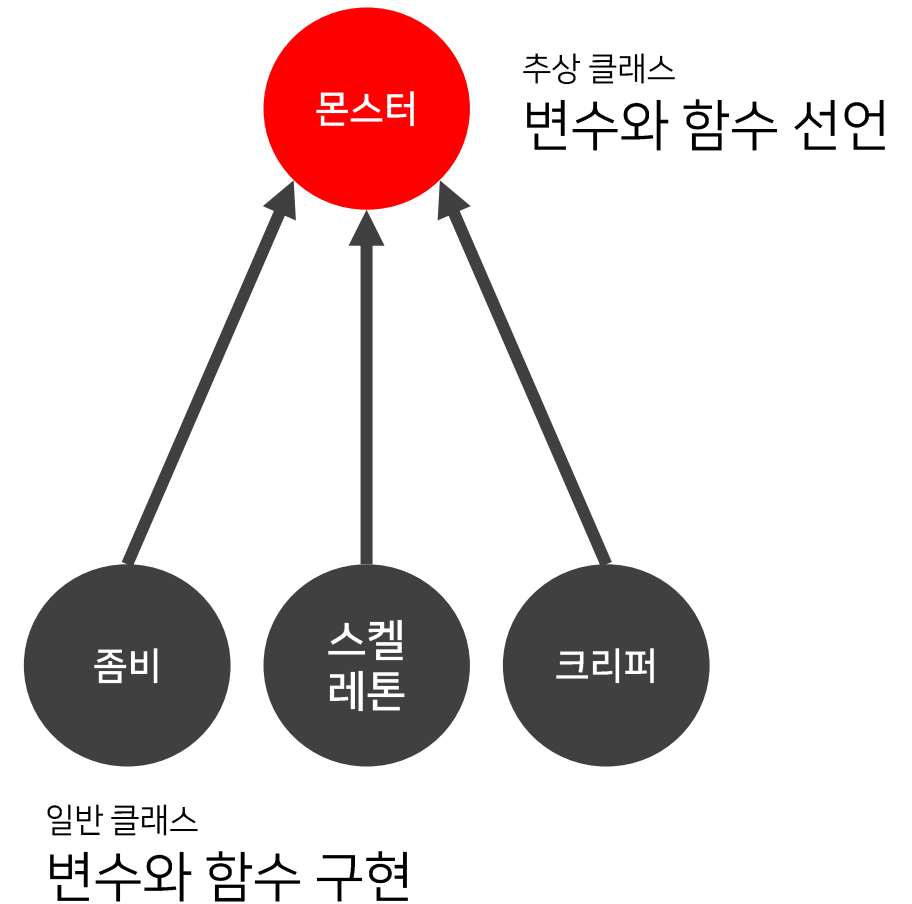
```
abstract class Monster(  
    val hp : Int, // 현재 체력  
    val maxHp : Int, // 최대 체력  
    val damage : Int, // 공격력  
) {  
    abstract val resources : String //몬스터 리소스  
    상속받은 다음 구현 해줄 변수와 함수  
    abstract fun attack()  
}
```

추상 클래스

```
class Zombie hp : Int, maxHp: Int, damage: Int) : Monster hp, maxHp, damage){
    override val resources: String = "res/zombie.png"
    override fun attack() {
        print("때리기")
    }
}

class Skeleton hp : Int, maxHp: Int, damage: Int) : Monster hp, maxHp, damage){
    override val resources: String = "res/skeleton.png"
    override fun attack() {
        print("활쏘기")
    }
}

class Creeper hp : Int, maxHp: Int, damage: Int) : Monster hp, maxHp, damage){
    override val resources: String = "res/creeper.png"
    override fun attack() {
        print("터지기")
    }
}
```

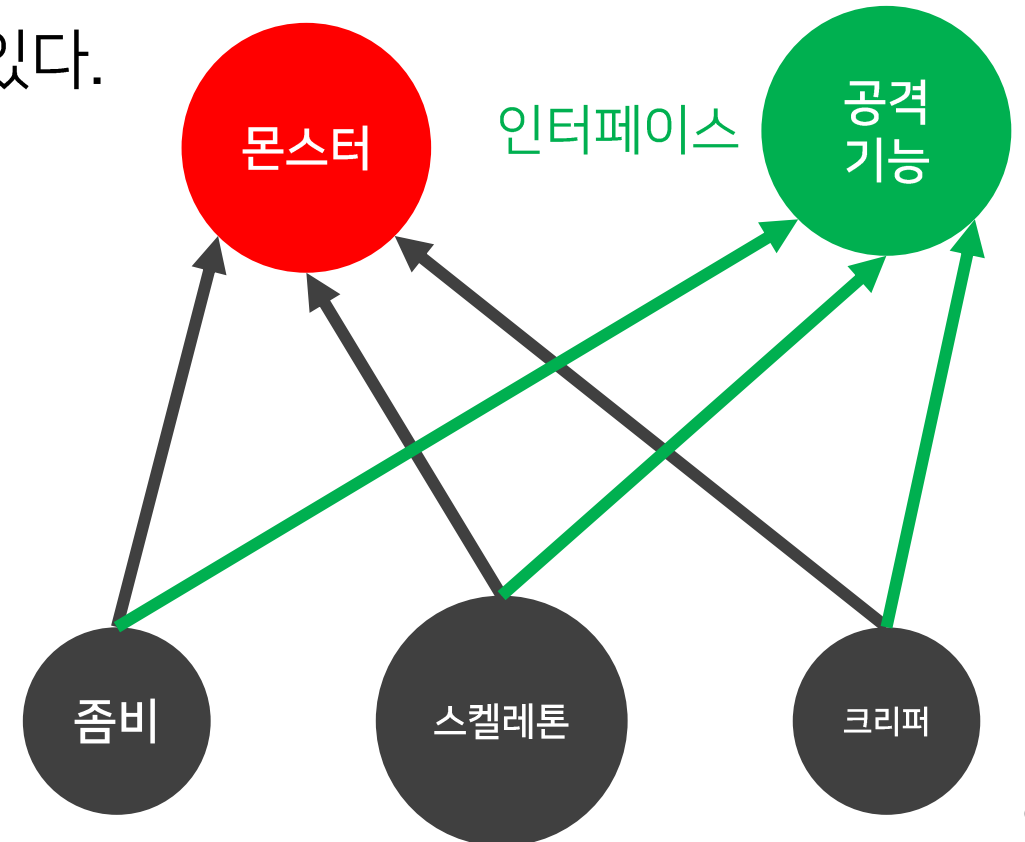


인터페이스

- 추상 클래스 만들기 귀찮을 때 사용하는 것

- 소프트웨어 마에스트로 연수생 **김형진**

- 추상 클래스와 비슷하지만 **메소드만 추상화** 할 수 있다.
- 하나의 클래스는 **여러 인터페이스를 상속** 받을 수 있다.
- 공통된 기능들을 추상화 할 때 사용 한다.



인터페이스



```
interface AttackMonster {  
    fun attack()  
}
```

```
abstract class Monster(  
    val hp : Int, // 현재 체력  
    val maxHp : Int, // 최대 체력  
    val damage : Int, // 공격력  
) {  
    abstract val resources : String //몬스터 리소스  
}
```

인터페이스

```
class Zombie (hp : Int, maxHp: Int, damage: Int) : Monster (hp, maxHp, damage), AttackMonster {  
    override val resources: String = "res/zombie.png"  
    override fun attack() {  
        print("때리기")  
    }  
}
```

```
class Skeleton (hp : Int, maxHp: Int, damage: Int) : Monster (hp, maxHp, damage), AttackMonster {  
    override val resources: String = "res/skeleton.png"  
    override fun attack() {  
        print("활쏘기")  
    }  
}
```

상속 과제

[마인크래프트를 코틀린으로 구현해보기]

조건

- 좀비, 스켈레톤, 크리퍼, 주민, 플레이어 구현
- 모든 일반 클래스에 멤버 변수와 메소드 각각 3개 이상 사용
- 추상 클래스와 인터페이스 각각 2개 이상 구현

기한

5월 31일 자정

클래스 이해 못했다면...?

1. 이 유튜브 영상 참고



<https://youtu.be/cg1xvFy1JQQ>

클래스 이해 못했다면...?

2. Kotlin 강의 수강

— 섹션 2. Kotlin 객체 지향 프로그래밍		13 강의 ⌚ 206 : 37
▶ 12강 객체지향 프로그래밍		⌚ 14 : 26
▶ 13강 생성자		⌚ 18 : 16
▶ 14강 상속		⌚ 13 : 03
▶ 15강 패키지		⌚ 12 : 59
▶ 16강 모듈		⌚ 08 : 44
▶ 17강 접근제한자		⌚ 28 : 32
▶ 18강 Property		⌚ 18 : 43
▶ 19강 지연초기화		⌚ 14 : 16
▶ 20강 Overriding		⌚ 22 : 54
▶ 21강 Any		⌚ 08 : 38
▶ 22강 this와 super		⌚ 17 : 26
▶ 23강 추상클래스		⌚ 12 : 00
▶ 24강 인터페이스		⌚ 16 : 40

들어야 하는 강의들