



클래스



- 클래스와 객체의 개념을 알고, 구분할 수 있다.
 - 클래스 선언에서 중요한 속성과 키워드를 알 수 있다.
 - 클래스 선언에서 구성요소들을 알고, 해당 내용에 대하여 명확히 구분할 수 있다.
 - 클래스에서 필드, 생성자, 메소드를 활용할 수 있다.
 - **this, static, final** 의 개념을 알고 구분할 수 있다.
 - 패키지 개념을 알고 사용할 수 있다.
 - 접근 제어자를 통해서 필드, 메소드에 대한 접근범위를 알고 활용할 수 있다.
-



생각해봅시다 :

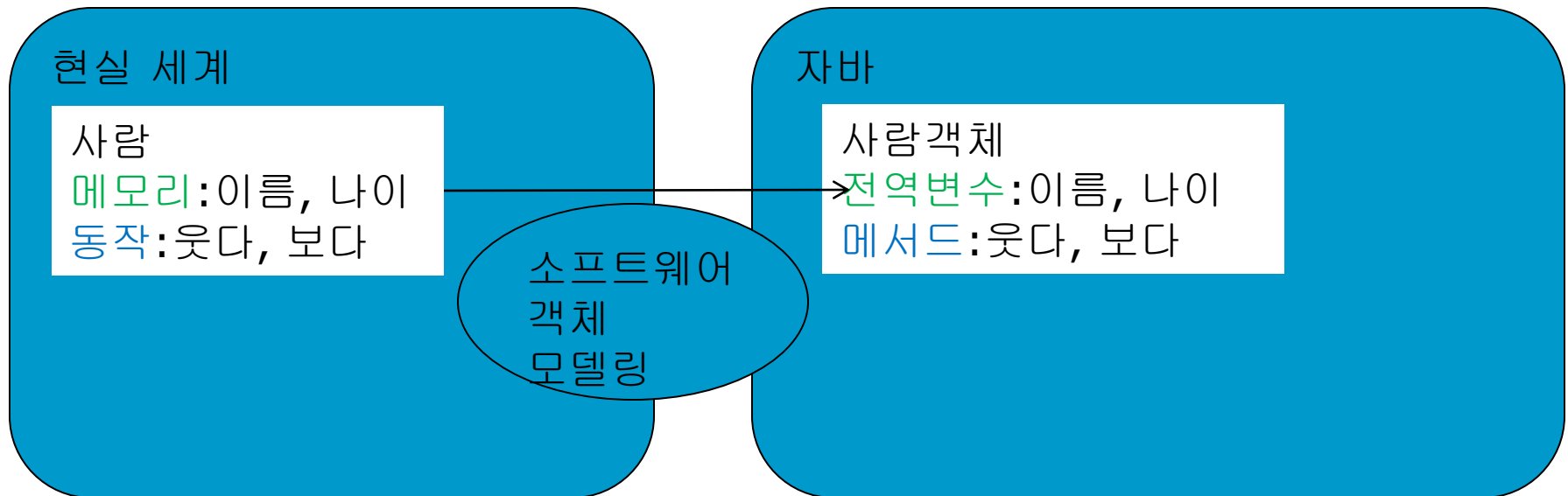
- 객체지향프로그램이 어떤 면에서 효율적인가?
 - 객체에는 속성과 메서드는 상호간의 어떤 작용을 할까?
 - 객체 **VS** 객체는 어떤 프로그램에서 활용될 것인가?
 - 객체 간의 접근제어는 왜 발생하는가?
 - **static**이라는 개념이 있는데, 공통 메모리에 쓰이는 개념이다. 어떤 경우에 이를 활용할까?
-



객체지향 프로그램!! :

■ 객체란?

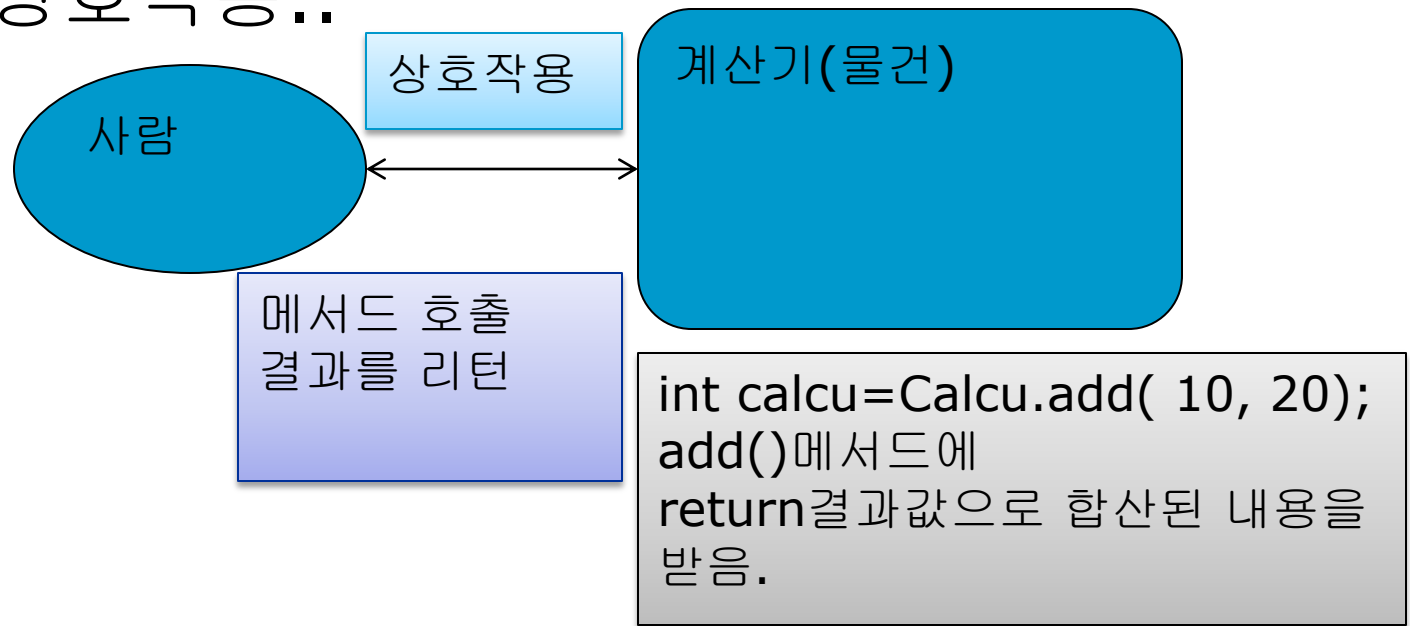
- 물리적으로 존재하거나 추상적으로 생각할 수 있는 것 중에서 자신의 속성이 있고, 다른 것과 식별 것을 말한다.





객체지향 프로그램 :

- 객체 모델링
 - 현실 세계나 추상적인 내용의 속성과 동작을 추려내어 소프트웨어 객체의 필드와 메서드로 정의해 나가는 과정.
- 객체의 상호작용..





객체 지향 프로그램의 특징 :

- 캡슐화(Encapsulation)
 - 객체가 포함한 속성과 메서드는 객체간의 관계에 있어서 감추거나 권한에 따라 접근이 가능하게 처리하는 것을 말한다. 여기서 사용되는 keyword로 접근제어자(access modifier)가 있다.
- 상속(Inheritance)
 - 상속이란 일반적으로 재산을 부모가 물려주는 개념이 있듯이, 자바에서는 부모가 가지고 있는 클래스의 속성과 메서드를 활용할 수 있는 개념으로부터 시작한다.
- 다형성(Polymorphism)
 - 같은 type이지만, 기능적으로 여러 다양한 객체를 이용할 수 있는 성질을 말한다. 프로그래밍 입장에서 상속받는 여러 객체를 대입함으로써 다양한 기능을 이용할 수 있다.



객체와 클래스 :

- 건물을 만들 때..
 - 설계
 - 현실성 있는 구현으로 건물을 만들어 낼 수 있다.
- 클래스는 건물을 만들 때, 설계로 보면 된다. 실제 만들어진 건물은 객체라고 비유할 수 있다.
- 자바프로그래밍 단계(코드, 컴파일, 메모리, cpu)
 - `public class Person{}`(코드) `Person.java`
 - `main()` 메서드가 있는 클래스에서 실행하였을 때
 - `Person p1 = new Person();` ➔ 객체로 실제 처리할 수 있는 구체화 된다.
 - `Person p2 = new Person();` ➔ 여러 객체들이 만들어짐.

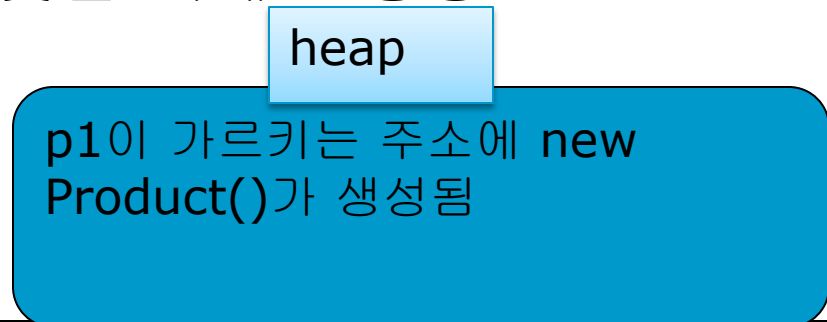
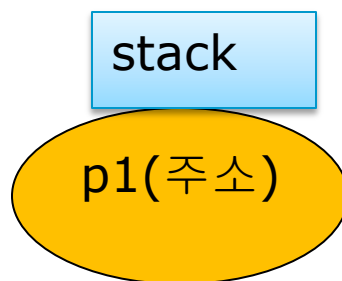


클래스 선언 :

- 문자열로 만들어지고, 변수선언 규칙과 유사
 - 첫번째 글자는 숫자가 올 수 없다.
 - \$, _ 외의 특수 문자를 사용할 수 없다.
 - 자바 내장 키워드를 사용할 수 없다.
- 파일명 **.java** → **public class** 클래스명
 - 파일 안에는 여러 클래스명을 선언할 수 있지만, **public**이 붙은 클래스명은 하나만 사용할 수 있다. **public**이 붙은 클래스명이 파일명으로 사용된다.
 - **Hello.java** **public** class **Hello**{}
 - class **Person**{}
- 클래스명은 일반적으로 첫 자를 대문자로 시작한다.
 - public class **Person**{}
- 모든 객체는 **main()**메서드가 포함된 클래스에서 객체로 호출해야 실행이 가능하다.

객체 생성 :

- 클래스를 선언한 다음, `main()`에서드가 있는 클래스에서 객체를 아래의 형식으로 생성할 수 있다.
 - `public class Product{}`
 - `main(){`
 - `// stack(주소) = heap(실제객체)`
 - `Product p1 = new Product();`
 - `// class가 선언된 것을 객체로 생성함.`





```
public class Person{  
    // 필드(전역변수)  
    int fieldName;  
    // 생성자(클래스명과 동일한  
    메서드)  
    Person(){}  
    // 메서드  
    void show(){}
```

클래스의 구성 멤버 :

- 필드
 - 객체를 저장하는 영역
 - 선언형태는 변수와 비슷하지만 전역변수라는 의미로 **field**로 사용되고 있다.
 - 메서드나 생성자에서 선언되는 지역변수와 구분
- 생성자
 - **new** 연산자로 호출되는 클래스명과 동일한 이름을 가진 메서드. 객체 생성할 때, 1번만 호출 됨
 - 메서드와 달리 **return**값이 없음.
- 메서드
 - 객체의 동작(기능처리)를 하는 것을 말한다.



필드 초기화 :

- 클래스가 생성될 때, 필드(전역변수)는 자동으로 초기값이 할당된다.
 - 객체는 `null`, 숫자:0, `boolean :false`
- 필드를 선언할 때, 초기값을 입력해서 처리할 수 있다.
 - `class Product{`
 - `int price = 25; // 선언할 때, 데이터할당 X`
- 필드 초기값이 필요한 경우, 생성자를 통해 입력한다.
 - `Product(int price){`
 - `this.price=price; // 생성자의 입력값을 통해 초기값`
 - `}`



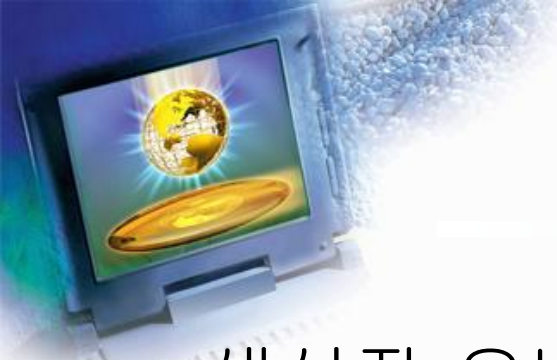
필드 선언 확인예제 :

- A05_FieldInitExp.java main() 클래스에서 선언
- BaseballTeam 객체 클래스 선언
 - 필드 : 팀이름, 승, 무, 패, 승률
 - 필드 팀이름 만 초기값 설정 "팀이름 없음";
 - 생성자를 통한 초기값 선언
 - 필드에 입력값 처리..
- 출력 예제.(객체2개 생성)
 - @@@ 팀, @@승, @@무, @@패, 승률 @@@
 - 입력값이 없는 생성자를 통해 출력
 - field값을 처리한 생성자를 통해 초기 데이터 설정
 - 출력 처리.



생성자 오버로딩(overloading) :

- 외부에서 입력되는 값이 다양하여 객체의 **field**값을 여러 형태로 처리할 때, 여러 형태의 생성자를 선언할 수 있다. 자바에서 클래스명과 동일한 생성자를 2가지 이상 선언할 수 있는데, 이를 오버로딩이라고 한다.
- 생성자 오버로딩선언규칙(이름이 동일하여도...)
 - 입력값의 갯수가 다르면 선언할 수 있다..
 - `Person(){} Person(int i){} Person(int i, int j){}`
 - 입력값의 갯수가 동일하더라도 다른 데이터**type**으로 선언하면 가능하다.
 - `Person(int i){} Person(String n){}`
 - 입력값의 갯수가 동일하더라도 순서가 다른 데이터**type**으로 선언하면 가능하다.
 - `Person(String name, int age){}`
 - ex) `new Person("홍길동",25);`
 - `Person(int age, String name)`
 - ex) `new Person(25,"홍길동")`
 - 생성자는 이름뿐만 아니라 입력값을 확인해서 인식한다.



다른 생성자 호출 처리 :

- 생성자 오버로딩에 의해서 다수의 생성자를 구현했을 때, 특정한 생성자에서 다른 생성자를 호출할 수 있다.
 - ex) **this**(입력값 갯수) : 입력값 갯수와 **type**과 동일한 생성자를 호출
- 호출 규칙
 - **생성자명**(매개변수1){}
 - 생성자명(매개변수1, 매개변수2){
 - **this**(매개변수1) :
 - **this**.field명 = 매개변수2;
 - }



다른 생성자 호출 처리 :

호출 예제

```
class Person{
```

```
    String name;
```

```
    int age;
```

```
    Person(String name){
```

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

```
    }
```

```
    Person( String name, int age){
```

```
        this(name) :
```

```
        // Person(String name)호출.
```

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

```
    }
```

```
....main(){
```

```
    Person p1 =new Person("홍길동",25);
```

```
    // name="홍길동", age = 25
```

```
}
```

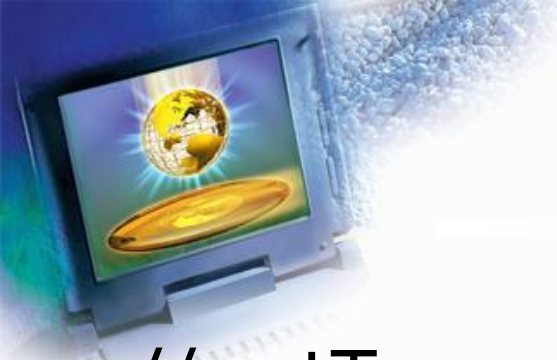
여러 생성자를 통해서 **생성자 재사용**을 할 수 있다.



확인예제 :

- class는 (좋아하는 연예인 → 클래스명 지정)
 - field명 : 이름, 성별, 분야(개그맨, 가수...)
 - 생성자(), 생성자(1), 생성자(2), 생성자(3)
 - main 메서드에서 객체생성4개..
 - == 내가 좋아하는 연예인 ==
 - 1. @@ @@ @@
 - 2. @@ @@ @@
 - 3. @@ @@ @@
 - 4. @@ @@ @@
-

- 객체의 동작에 해당하는 기능으로 {}을 포함해서 처리한다.
- 선언
 - 리턴type 메소드이름(매개변수1,2..){
 - 매개변수로 처리할 프로세스(field명에 할당)
 - 조건, 반복문..등 원하는 데이터를 처리
 - return 실제호출될 값(리턴type으로 정해 놓은 data Type이어야한다)
 - }




메소드 기본예제 :

```
// retType 메서드명    입력값
int      plus ( int num01, int num02){
    int sum=num01+num02;
    return sum;
}
```

```
// main()    int tot=cal.plus(5 ,7 );
```

int sum;//으로 선언이 가능해야 한다.

return 할 데이터 값이 없을 때는 **void**라는
keyword를 **return type**을 설정하는 부분이
코딩해 놓아야 한다.

- 
- Mart 클래스
 - 물건명, 단가, 갯수
 - cart(물건명, 단가, 갯수) :
 - 데이터 할당,
 - return 총액
 - calcu() : 출력!!!
 - 물건명, 가격, 갯수
-

- 은행에 가서..(2단계)
 - Account(계좌계정)
 - field명 : 계정명, (입금액, 출금액), 총잔액, (날짜), (내용)
 - 생성자1 : 계정명 입력.
 - 생성자2 : 계정명, 초기 입금액
 - save(입금액, 내용, 날짜)
 - 계산처리
 - return 입금된 금액
 - withdraw(출금액, 내용, 날짜)
 - print()
 - 날짜 용돈 [@@@] [@@@@] [@@@@@]



■ 은행(1단계)

– 계좌

- field : 계좌명, (총액)
- 생성자(계좌명 입력)
- input(입금액)
 - @@@님 @@@원 입금했습니다.(현잔액@@)
- output(출금액)
 - @@@님 @@@원 출금했습니다.(현잔액@@)

■ 자기소개(1단계)

– field : 이름, 사는곳, 나이


- 생성자(이름 입력)
- inputData (사는곳, 나이)
- introMyself() : 출력 처리.
- 내이름은 @@이고, 나이는 @@이며, 사는 곳은 @@@입니다.



메소드 오버로딩(overloading) :

- 클래스 내에서 생성자와 같이 같은 이름의 메소드를 선언(**중복정의**)해서 사용할 수 있다. 단!!
 - 매개변수 갯수가 다를 때
 - void buyProd(String prod)
 - void buyProd(String prod, String marName)
 - 매개변수 data Type이 다를 때
 - void buyProd(String prod)
 - void buyProd(int cnt)
 - 매개변수가 갯수가 같을 지라도 순서 data Type 다른 때
 - void buyProd(String prod, int cnt)
 - void buyProd(int cnt, String prod)
-

기본 예제 :



```
class ShoppingMall{  
    String name;  
    String prodName01; //  
    String prodName02;  
    String prodName03;  
    int price01;  
    ShoppingMall(String name){  
        System.out.println("온라인 쇼핑몰 "+name+"에 오신 것을 환영합니다!!");  
    }  
    // 매개 변수 갯수가 다른 것도 선언가능.  
    void buyProduct(String prodName01){  
        this.prodName01 = prodName01;  
    }  
    void buyProduct(String prodName01, String prodName02){  
        this.prodName01 = prodName01;  
        this.prodName02 = prodName02;  
    }  
}
```



기본 예제 :

*// 매개변수의 **type** 이 다르면 동일한 이름의 메서드 선언가능.*

```
void buyProduct(int price01){  
this.price01=price01;  
}
```

*// 매개변수의 **type** 과 숫자가 동일하더라도, 다른 **type** 의
매개변수의*

// 순서가 다르면 선언가능.

```
void buyProduct(String prodName01, int price01){  
this.prodName01=prodName01;  
this.price01 = price01;  
}
```

```
void buyProduct( int price01, String prodName01){  
this.prodName01=prodName01;  
this.price01 = price01;  
}
```




정적 멤버와 **static**

- **static** : '고정된'이라는 뜻으로 객체생성과 상관없이 사용할 수 있는 필드와 메서드에 활용된다.
 - static int **com_money**
 - 객체명.static변수 ex) Person.**com_money**;
 - Person p1 = new Person();
 - p1.money; // p1의 고유 속성..1000
 - p1.**com_money**=3000;
 - Person p2 = new Person();
 - p2.money; // p2의 고유 속성..p1의 money상관없이 데이터를 할당..
- 선언 형식
 - class Person{
 - **static** int com_money;
 - 객체 생성(new Person)을 하지 않더라도 사용가능



static 멤버 확인예제(숙제):


- 곰돌이 3형제가 사과 먹는 이야기..
 - 클래스 Bear
 - field명: 이름, 각자가 먹은 갯수, 현재 전체 남은갯수
 - 생성자 : 이름 할당.
 - 메서드 : `restoreApple()` (사과갯수 추가)
 - `eatApple()` : 각 곰돌이가 사과를 먹을 때 처리:
 - 각자가 먹은 갯수, 현재 전체 남은갯수
 - 출력 : @@@ 사과를 먹는다.
 - @@@ 먹은 사과갯수는 @@@
 - 현재 남은 사과갯수 @@



정적 필드의 초기화 :

- 클래스명 하위에 바로 필드 선언과 동시에 초기값을 주는 것일 보통
 - `static double pi = 3.14159;`
 - 정적필드는 초기화를 위하여 정적 블록 제공한다
 - `static{`
 - 정적필드의 초기값 할당.
 - `}`
 - 메모리 : 클래스가 메모리로 로딩될 때, 자동적으로 실행된다.
-

기본예제 :



```
class Television{
// static은 객체생성 후, 사용되는 필드가 아니기에
// 클래스명 바로 밑에서 초기화를 해주는 것이 보통이다.
static String company="삼성";
static String model="OLED";
int price;
static String info;
// static{}을 주어서 선언된 초기필드를 초기데이터로 할당할 수
// 있다.
static{
info = company+" - "+model;
//static 필드가 아닌 필드는 static{}에서 사용할 수 없다.
//price=15000000; (컴파일에러)
showAll(); // static{}에서는 static 메서드를 사용가능.
//show(); 일반 메서드는 사용이 불가능하다.(컴파일에러)
}
```



기본예제 :

```
void show(){}  
// static 메서드도 객체생성없이 사용할 수 있다.  
static void showAll(){  
    System.out.println(" 좋은 TV 입니다!!!");  
//this.price=2000000; staic 메서드 안에서 객체 생성후  
현재객체(this) 필드에 데이터를 할당하지 못 한다.  
//this.show(); static 메서드 안에서 일반메서드를 사용하지 못  
한다.  
// static 블록이나 메서드에서 static 필드나 메서드를  
사용하기위해서  
// 는 객체생성을 하여야한다.  
    Television t = new Television();  
    t.price=2500000;}}
```



싱글톤(Singleton) :

- 프로그래밍에 있어서, 하나의 객체만 활용할 때가 있다.
 - 하나의 객체만 활용: **class code**로 클래스명을 지정하는 설계를 위한 것이다. 실제 **main**에서 호출하는 여러 객체를 생성할 수 있게 프로그래밍이 되어 있다.
 - 이것이 기본적인 객체 생성 구조인데, 일 사전에 하나만 생성되게 프로그래밍또는 설계한 클래스에서 생성된 것을 싱글톤 객체라고 한다.
-



싱글톤(Singleton) :

- 프로그래밍에 있어서, 하나의 객체만 활용할 때가 있다.
 - 하나의 객체만 활용: **class code**로 클래스명을 지정하는 설계를 위한 것이다. 실제 **main**에서 호출하는 여러 객체를 생성할 수 있게 프로그래밍이 되어 있다.
 - 이것이 기본적인 객체 생성 구조인데, 사전에 하나만 생성되게 프로그래밍 또는 설계한 클래스에서 생성된 것을 싱글톤 객체라고 한다.
-

싱글톤 구조 :

- public class 클래스{
 - // private : 외부 객체에서 접근하지 못하게 하는 제어자.
 - // 정적 필드 생성 클래스명.XXX(x)
 - private static 클래스 **single** = new 클래스();
 - // 생성자 main() 클래스 ob = new 클래스() (X)
 - private 클래스(){}
 - // 선언 클래스에서 단일 객체만 생성되게 하는 목적
 - 유일하게 객체를 접근해서 return;
 - // 정적메서드를 선언..
 - static 클래스 getInstance(){
 - return **single**;
 - }
- }
- main() 이나 다른 클래스에서 호출
- 클래스 참조변수1 = 클래스.getInstance();



final 필드와 상수 :

- **final** : 최종이라는 의미로 필드 앞에 붙이면 데이터를 변경하게 못하게 하는 것을 말한다. 초기값 할당 후, 데이터가 변경되지 못하게 처리된다.
 - final 타입 필드명 = 초기값;
 - ex) final String nation="korean";
 - main() p1.nation="usa";// 에러발생
수정불가능



패키지(package) :

- 프로그래밍에서 여러 클래스를 관리 하기 위해 기능적으로 영향을 미칠 수 있는 클래스끼리 묶어 놓고, 접근 범위 안에 효과적으로 호출하기 위해서 사용하는 개념이다.
- 패키지는 물리적인 파일 시스템 기능이상으로 기능적 처리의 **밀접성**으로 접근범위를 설정하는 개념과 종속관계에 있어서 상위폴드개념을 추가하여 최상위package.하위패키지가 동일해야지 같은 package로 인식한다.
 - 상위패키지.하위패키지
 - 상위패키지명만 같은 것으로 같은 패키지라고 할 수 없다.



패키지 선언과 활용 :

- 패키지는 클래스명 위에 최상에 위치한다.
 - `package` 상위패키지.하위패키지
 - `public class` 클래스명{}
 - 명명규칙
 - 숫자(X), `_`, `$`(제외 특수문자)도 사용 안됨.
 - 모두 소문자로 작성하는 것이 **naming** 규칙
 - **java**로 시작하는 패키지는 자바 표준 **api**에 사용되므로 사용해서는 안 된다.
 - 같은 패키지끼리는 **import**라는 개념 없이 사용이 가능하며, 동일한 클래스를 정의할 수 없다.
-



import문 :

- 같은 패키지를 사용하는 객체끼리는 특별한 키워드 없이 클래스(객체)를 호출해서 사용할 수 있다..

```
<<Note.java>>
package com.util;
public class Note{}
```

```
<<Pen.java>>
package com.util;
public class Pen{
    Note n = new Note();
}
```

- 패키지가 다르면..
 - 패키지명까지 포함해서 클래스 선언
 - package명 밑에 import 패키지명.사용할클래스

```
<<Computer.java>>
package com.util.prog;
public class Computer {
    com.util.Note note01 =
new com.util.Note();
}
```

```
<<Computer.java>>
package com.util.prog;
import com.util.Note;
public class Computer {
    Note note01 = new Note();
}
```



package와 import 확인예제 :

- 다음 같은 구조를 만들고, 해당 객체간 호출 처리 방법을
 - import를 사용하는 방법
 - Cousin 클래스에서 Mine 클래스 호출하기
 - import 사용 없이 호출하는 방법으로 객체를 생성하세요.
 - Mine 클래스에서 Father 클래스 호출하기
 - Mother 클래스에서 Aunt 클래스 호출하기
 - **package javaexp.a06_object** 패키지 하위에
 - **ourhome** 패키지 생성
 - **Father, Mother, Mine**
 - **cousinhome** 패키지 생성
 - Aunt, Uncle, Cousin
-



접근제어자(**Access Modifier**) :

- **main()** 메서드를 가지지 않는 클래스는 외부 클래스를 사용할 목적으로 설계된 라이브러리 클래스이다. 이런 클래스는 상호간의 필드, 생성자, 메소드에 대한 적절한 접근제어가 필요하다. 아래 내용은 **접근제어자**를 통해서 처리가 가능하다.
 - 특정한 필드는 **클래스 내**에서만 사용되어야 하고,
 - 특정한 메서드는 **같은 package**에서만 사용되어야 한다.
 - 특정한 메서드는 **모든 클래스에서 사용** 가능해야 한다.
 - 특정한 필드나 메서드는 **상속 관계**에 있는 클래스만 접근이 가능하다



접근제어자들~~~

접근 제어자	적용 대상	접근 범위
public	클래스, 필드, 생성자, 메소드	모두 접근 가능
protected	필드, 생성자, 메소드	상속한 다른 패키지
default(입력하지 X)	클래스, 필드, 생성자, 메소드	같은 패키지끼리
private	필드, 생성자, 메소드	클래스 내에서만 사용

1) 클래스명에 선언..
public class Person{}

2) 메서드에 선언
protected void showAll(){}

3) 생성자에 선언
public Person(){}

4) 필드에 활용
private String name;



클래스 선언 접근 제어자 :

- 클래스를 선언할 때, 접근 제어자가 영향을 미치는 범위를 설정.
 - 범위 : `public`, `[default]`
 - `[public] class 클래스명{}`
 - `main()`, 외부 클래스에서 접근 범위로 영향
 - `public`일 때는 어떤 클래스이든지 선언가능.
 - `public class Person{}`
 - `main() : Person p; // 외부 패키지에서 선언이 가능.`
 - `class Woman{}`
 - `main() : Woman p; // 패키지 내에만 선언 가능.`



생성자, 필드, 메서드 접근제어 :

- 접근 제어 범위 : `public`, `protected`, `[default]`, `private`
 - `public` : 전체 패키지 관련 없이 접근 가능.
 - `protected` : 상속관계에 있는 객체의 생성, 메서드, 변수 접근
 - `[default]` : 패키지 단위 접근 가능
 - `private` : 클래스 내에서만 사용 가능..
-




숙제 (접근제어자) :

- `javaexp.a06_object.access`에
 - `company` 패키지 생성
 - `manager`(총무부) 패키지 생성
 - `HeadMember`(부장)
 - 가족생일(개인정보) - `private`
 - `ChiefMember`(과장)
 - 총무부기획서 - 패키지 내에서만 정보
 - 부장 정보 확인 (`searchInf()`)
 - `StaffMember`(사원)
 - 장가 - 공지사항 정보
 - 과장정보 확인 (`searchInf()`)
 - `personnel`(인사부) 패키지 생성
 - `ChiefMember`(과장) - 해당 접근제어자에 따라 접근여부를 확인 처리.. (`searchInf()`)




가변 입력값 :

- 메서드나 생성자를 통해서 **같은 데이터 타입**의 입력값의 갯수가 정해지지 않고 **여러 개**로 입력될 수 있을 때, 가변인자로 등록할 수 있게 한다.
- 형식
 - [return Type] 메서드명(데이터type ... 입력변수)
 - 입력변수는 배열(입력변수.length) for문을 활용해서 입력값을 받아 들인다
 - 호출하는 곳 (`main()`, 다른 클래스의 메서드)
 - 참조변수.가변인자메서드(데이터1);
 - 참조변수.가변인자메서드(데이터1,데이터2);
 - 참조변수.가변인자메서드(데이터1,데이터2,데이터3);



```
class GoMountain{
// String ...names 가변 입력값 선언..
// 호출하는 곳에서 여러 개의 입력값을
// 가변적으로 입력이 가능하도록 처리
    public void callName(String ...names){
// names는 배열
        for(String name:names){
// 부름이름..!!
            System.out.println(name+"~~~ 잘
            있는지?");
        }
    }
}
```



```
public static void main(String[] args) {  
  // TODO Auto-generated method stub  
  GoMountain m1 = new GoMountain();  
  System.out.println("산에 가서 부른 이름!!");  
  m1.callName("홍길동");  
  System.out.println("산에 가서 부른 이름!!");  
  m1.callName("김길동","신길동");  
  System.out.println("산에 가서 부른 이름!!");  
  m1.callName("오철수","김영희","신국주");  
}
```



확인예제 :

■ 1단계

- 맛있는 음식에서 가서 (클래스 생성)
 - `oderFood`(가변인자 처리)
- 출력내용
 - 주문하신 음식은 : @@
 - 주문하신 음식은 : @@, @@ 가변적으로 처리

■ 2단계

- 가게의 이름을 생성자로 입력 받게 처리.
- `oderFood` 를 통해서 주문된 음식의 종류에 따라 매핑되어 있는 가격을 계산해서 계산서까지 출력처리




Getter와 Setter 메서드 :

- 접근제어자를 만들어 둔 이유는 필드의 내용이 직접 접근하는 것을 막기 위한 것이 있다. 보통 필드의 접근제어자는 **private** 설정해서 이를 제한 해준다. 아래와 같이 메서드를 통해서


데이터 무결성 확보

- **setXXX**: 데이터 저장
- **getXXX**: 데이터 호출

```
class Car{  
    private int curVelocity;  
    int speedDown(){  
        curVelocity--;  
        if(curVelocity < 0 ){  
            curVelocity=0;  
        }  
        return curVelocity;  
    }  
}  
main(), 외부 클래스 호출  
c.curVelocity--; // 음수...
```


- 
- 퀵보드를 타고!!!
 - QuickBoard
 - 필드: rider, curVelocity
 - 생성자 : 타는 사람 이름 설정..
 - 메서드 : speedUp(), speedDown()
 - @@@가 퀵보드를 타서~~
 - 속도를 올립니다. 제한속도 20
 - 속도를 내립니다. 멈춰있네요. 0
-

확인예제 :



```
class QuickBoard{
    private String rider;
    private int curVelocity;
    public QuickBoard(String rider) {
        this.rider = rider;
        System.out.println(rider+"가 쿼보드를 타서");
    }
    public void speedUp(){
        System.out.println("속도를 올립니다.");
        curVelocity++;
        if(curVelocity>20){
            System.out.println("제한 속도 20(km/h) 입니다.");
            curVelocity=20;
        }
        System.out.println("현재 속도 "+curVelocity+"(km/h)");
    }
}
```

확인예제 :



```
public void speedDown(){
    System.out.println("속도를 내립니다.");
    curVelocity--;
    if(curVelocity<0){
        System.out.println("현재 멈춰 있습니다.");
        curVelocity=0;
    }
    System.out.println("현재 속도 "+curVelocity+"(km/h)");
}
```



정리 및 확인하기 :

- 객체와 클래스에 대한 설명을 틀린 것은?
 1. 클래스는 객체를 생성하기 위한 설계도와 같은 것이다.
 2. **new** 연산자로 클래스의 생성자를 호출함으로써 객체가 생성된다.
 3. 하나의 클래스로 하나의 객체만 생성할 수 있다.
 4. 객체는 클래스의 인스턴스이다.
- 클래스의 구성 멤버가 아닌 것은 무엇인가?
 1. 필드(field)
 2. 생성자(constructor)
 3. 메소드(method)
 4. 로컬 변수(local variable)
- 필드에 대한 설명으로 틀린 것은 무엇인가?
 1. 필드는 메소드에서 사용할 수 있다.
 2. 인스턴스 필드 초기화는 생성자에서 할 수 있다.
 3. 필드는 반드시 생성자 선언 전에 선언되어야 한다.
 4. 필드는 초기값을 주지 않더라도 기본값으로 자동 초기화된다.



정리 및 확인하기 :

- 메소드 오버로딩에 대한 설명으로 틀린 것은 무엇입니까?
 1. 동일한 이름의 메소드를 여러 개 선언하는 것을 말한다.
 2. 반드시 리턴 타입이 달라야 한다.
 3. 매개 변수의 타입, 수, 순서를 다르게 선언해야 한다.
 4. 매개값의 타입 및 수에 따라 호출될 메소드가 선택된다.
- **final** 필드와 상수(**Static final**)에 대한 설명으로 틀린 것은 무엇입니까?
 1. **final** 필드와 상수는 초기값이 저장되면 값을 변경할 수 없다.
 2. **final** 필드와 상수는 생성자에서 초기화될 수 있다.
 3. 상수의 이름은 대문자로 작성하는 것이 관례이다.
 4. 상수는 객체 생성 없이 클래스를 통해 사용할 수 있다.



- 접근 제한에 대한 설명으로 틀린 것은 무엇입니까?
 1. 접근 제한자는 클래스, 필드, 생성자, 메소드의 사용을 제한한다.
 2. **public** 접근 제한은 아무런 제한 없이 해당 요소를 사용할 수 있게 한다.
 3. **default** 접근 제한은 해당 클래스 내부에서만 사용을 허가한다.
 4. 외부에서 접근하지 못하도록 하려면 **private** 접근 제한을 해야 한다.
- 다음 클래스에서 해당 멤버가 필드, 생성자, 메소드 중 어떤 것인지 빈칸을 채우세요
- public class Member{
 - private String name ←----- ()
 - public Member(String name)←-----()
 - public void setName() ←-----()



정리 및 확인하기 :

- 은혜 계좌 객체인 **Account** 객체는 잔고(**balance**) 필드를 가지고 있습니다. **balance** 필드는 음수값이 될 수 없고, 최대 백만원까지만 저장할 수 있습니다. 외부에서 **balance** 필드를 마음대로 변경하지 못하도록 하고, $0 \leq \text{balance} \leq 1,000,000$ 범위의 값만 가질 수 있도록 **Account** 클래스를 작성해 보세요
 - **setXXX**와 **getXXX**를 이용하세요.
 - 0과 1,000,000은 **MIN_BALANCE**, **MAX_BALANCE** 상수를 선언해서 사용하세요.
 - **setXXX** 매개값이 음수이거나 백만원을 초과하면 현재 **balance** 값을 유지하세요.



감사합니다 !
