Override, 객체 형변환, 객체 배열

#01. Object 클래스

자바의 모든 클래스는 특별히 부모를 지정하지 않을 경우 자바에 내장되어 있는 Object 클래스를 상속받는다.

Object 클래스는 toString() 메서드를 갖고 있다.

예를 들어 임의로 작성한 클래스에 대한 객체는 toString() 메서드를 사용할 수 있다.

```
class Foo {}

Foo f = new Foo();
System.out.println(f.toString());
```

Object 클래스의 toString() 메서드는 객체의 Hash값을 반환한다.

> Hash값 : 컴퓨터에서 해당 객체를 식별하기 위해 부여한 일련번호 ex) Protoss@4517d9a3

자바에서는 이 메서드를 현재 객체가 갖고 있는 멤버변수의 상태를 반환하도록 재정의 할 것을 권장한다.

1) VSCode 사용시

Ctrl + Shift + P --> Generate toString() 선택

#02. 메서드 Override

부모가 갖는 메서드를 자식 클래스가 동일하게 재정의하는 형태

자식 클래스가 부모의 메서드를 Override할 경우 부모가 갖는 원래 기능은 가려진다.

> 자식 클래스를 통한 부모의 기능(=메서드) 수정 : 기능의 확장

1) 어노테이션(Annotation)

코드 사이에 주석처럼 쓰이며 특별한 의미, 기능을 수행하도록 하는 기술

2) @Override 어노테이션

어떤 메서드가 반드시 부모 클래스의 메서드를 재정의 해야 한다는 강제성을 부여함.

메서드를 Override 하는 과정에서 오탈자를 방지하기 위해 사용됨

3) 생성자의 Override

생성자는 상속되지 않는다.

하지만 부모 클래스가 기본 생성자가 아닌 다른 형태의 생성자를 갖고 있는 경우 그 생성자는 반드시 실행되어야만 한다. 그렇지 않으면 부모의 멤버변수는 초기화될 수 없다.

자식 클래스는 자신의 생성자를 통해 부모의 생성자를 강제로 호출해야 한다.

이 과정에서 부모 생성자가 파라미터를 받는다면 자식 클래스는 자신이 부모와 동일한 파라미터를 받는 생성자를 정의하고 자신의 생성자로 주입되는 파라미터를 부모에게 전달해야 한다.

4) `super`

`this`가 현재 클래스를 의미하고, `super`는 부모 클래스를 의미한다.

`super` 예약어의 용법

1. `super`를 메서드처럼 사용하면 부모의 생성자를 의미한다.
2. 일반 메서드 안에서 `super` 키워드를 통해 부모의 기능에 접근할 수 있다.
    - 자식이 Override 하지 않은 기능일 경우 상속된 기능이므로 `this`를 통해 접근하는 경우와 차이가 없다.
    - 자식이 Override 한 기능일 경우 `this`는 자식이 재정의한 기능에 접근하고, `super`는 재정의하기 전의 부모가 갖는 원본 기능에 접근한다.

#03. 지금까지의 내용을 적용한 클래스 설계

```
@startuml MyDiagram
scale 3

class Protoss {
    +<b>Protoss(name: String, hp: int, speed: int, dps: int)
    ----
    -name: String
    -hp: int
    -speed: int
    -dps: int
    ----
    +getName(): String
    +setName(name: String): void
    +getHp(): String
    +setHp(name: String): void
    +getSpeed(): String
    +setSpeed(speed: String): void
    +getDps(): String
    +setDps(dps: String): void
    +move(position: String): void
    +attack(target: String): void
    +toString(): String
}

class Zilot {
    +<b>Zilot(name: String, hp: int, speed: int, dps: int)
    ----
    +move(position: String): void
    +attack(target: String): void
    +<b>swardAttack(target: String): void
}
```

```
class Dragun {
    +<b>Dragun(name: String, hp: int, speed: int, dps: int)
    ----
    +move(position: String): void
    +attack(target: String): void
    +<b>fireAttack(target: String): void
}

Protoss <|-- Zilot
Protoss <|-- Dragun

@enduml
```

```
@startuml MyDiagram

class Protoss {
    +<b>Protoss(name: String, hp: int, speed: int, dps: int)
    ----
    -name: String
    -hp: int
    -speed: int
    -dps: int
    ----
    +getName(): String
    +setName(name: String): void
    +getHp(): String
    +setHp(name: String): void
    +getSpeed(): String
    +setSpeed(speed: String): void
    +getDps(): String
    +setDps(dps: String): void
    +move(position: String): void
    +attack(target: String): void
    +toString(): String
}

class Zilot {
    +<b>Zilot(name: String, hp: int, speed: int, dps: int)
    ----
    +move(position: String): void
    +attack(target: String): void
    +<b>swardAttack(target: String): void
}

class Dragun {
    +<b>Dragun(name: String, hp: int, speed: int, dps: int)
    ----
    +move(position: String): void
    +attack(target: String): void
    +<b>fireAttack(target: String): void
```

```
}

Protoss <|-- Zilot
Protoss <|-- Dragun

@enduml
```

#04. 객체 형변환

자식 클래스로부터 생성된 객체를 부모 클래스 형태로 변환하는 처리

1) 상속 관계의 클래스 정의

```
class Foo {
    ...
}

class Hello extends Foo {
    ...
}

class World extends Foo {
    ...
}
```

2) 형변환 방법

a) 자식 객체를 부모 객체에 참조 시킴

```
// 자식의 객체 생성
Hello h = new Hello();
World w = new World();

// 형변환을 통해 서로 다른 두 객체의 타입이 통일된다.
Foo f1 = h;
Foo f2 = w;
```

b) 선언은 부모 형태로, 할당은 자식 형태로 처리

```
Foo f1 = new Hello();
Foo f2 = new World();
```

3) 객체 형변환시 발생하는 현상

1. 부모 객체로 변환되더라도 자식 클래스가 Override 한 기능은 자식의 기능을 그대로 유지된다.
2. 부모에게 상속받은 기능 이외에 자식 클래스가 독자적으로 추가한 기능은 사용할 수 없도록 잠긴다.

4) 자식 객체를 부모 형태로 변환하는 경우

상위 클래스 형태로 형변환하기 때문에 `UpCasting`이라고 부름

자식 클래스가 독자적으로 추가한 기능을 사용할 수 없게 박스에 넣어 놓는다는 의미로 `Boxing` 이라고 하기도 함

5) 부모 형태를 자식 형태로 역변환 하는 경우

하위 클래스 형태로 형변환하기 때문에 `DownCasting`이라고 부름

자식 클래스가 독자적으로 추가한 기능을 사용하지 못하도록 박스에 넣어 놓았다가 다시 꺼낸다는 의미로 `UnBoxing`이라고 하기도 함

6) 성립하지 않는 경우

a) 처음 생성시 부모 클래스의 객체로 만들어지만 자식 형태로 변환 불가

```
Foo f = new Foo();
Hello h = f;        // 여기서 에러
```

b) 역변환시 최초 생성된 형태가 아닌 다른 자식 클래스 형태로는 변환 불가

```
Hello h = new Hello();
Foo f = h;
World w = f;        // 여기서 에러
```

#05. 객체 배열

배열의 원소가 객체인 경우

1) 단일 클래스의 경우

```
class Hello { ... }

// 배열의 크기 할당
Hello[] h = new Hello[3];
```

객체 배열의 원소는 지정된 클래스의 객체로 할당해야 한다.

```
h[0] = new Hello();
h[1] = new Hello();
h[2] = new Hello();
```

2) 서로 다른 클래스인 경우

원칙적으로 배열은 동일 데이터 타입만 저장 가능하다.

서로 다른 클래스의 객체를 하나의 배열에 저장하기 위해서는 같은 부모로부터 파생된 경우만 가능하다.

```
class Foo {
    ...
}

class Hello extends Foo {
    ...
}

class World extends Foo {
    ...
}

// 부모 클래스에 대한 객체 배열 생성
Foo[] f = new Foo[5];

// 부모 클래스 형태의 배열에 자식 클래스의 객체를 저장하면 Boxing처리되어 원소로 할당된
다.
f[0] = new Hello();
f[1] = new World();
f[2] = new Hello();
f[3] = new World();
f[4] = new Hello();
```


1902
6910
98/105
MENU
INVEN