06_1 Inheritance

Object-Oriented Programming

Inheritance에 대해 강의하겠습니다.

Inheritance

- Reduce duplicate code by reusing already well-developed classes to create new ones

```java
// A: parent class (= base class = super class)
public class A {
    int field1;             // A has field1
    void method1() { }      // A has method1()
}

// B: child class (= derived class = sub class)
public class B extends A { // B inherits from A
    int field2;             // B has field1 and field2
    void method2() { }      // B has method1() and method2()
}
```


페이지 2

Inheritance, 즉, 상속의 목적은 중복되는 코드를 줄이고
제대로 개발된 클래스들을 재사용하여
새로운 클래스를 생성하는데 있습니다.

class A를 parent class로 가정할 때,
A는 다른 이름으로 base class 또는 super class로도 불립니다.
class A에는 instance variable field1이 있고,
method1() 이 정의되어 있습니다.

class B를 child class (또는 derived class 또는 sub class) 이라 할 때
class B extends A 라고 "extends" 라는 keyword를 써서
A와 B간에는 parent, child관계가 성립되게 됩니다.
이 때 B는 A에서 inherit한다, 즉, B가 A에게서 상속받는다 라고 합니다.
코드만으로 볼 때에는 class B에는 instance variable field2가 있고
method2() 가 있습니다.
그러나 B는 A로 부터 상속을 받았기 때문에
비록 눈에 보이지는 않지만 B에는 원래 A가 가지고 있던
field1과 method1() 도 존재하는 것입니다.

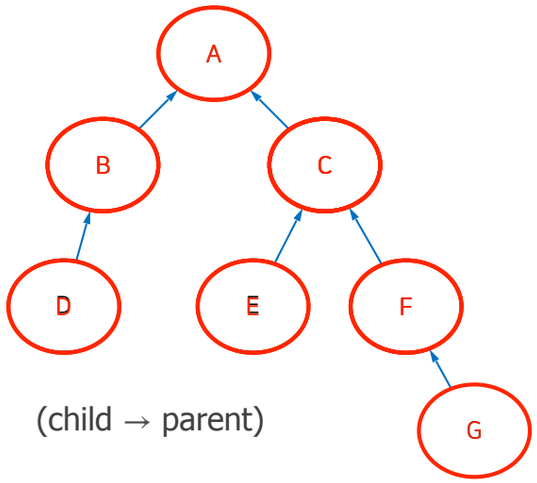
Inheritance Rule

1. Only one parent class is allowed
2. Private fields and methods in the parent class cannot be accessed directly by the child class.
3. If the parent class exists in a different package, fields and methods with default (package) access cannot be directly accessed from the child class.

페이지 3

Inheritance에 대해 정해져 있는 rule들을 살펴보겠습니다.
먼저 parent로는 단 하나의 class만을 가질 수 있습니다.
이것은 Java 언어가 C++ 언어에 대해 가지고 있는 큰 차이점 중의 하나 입니다.
C++ 언어는 parent class가 여러개가 될 수 있는
multiple inheritance가 가능한데
사실 이러한 속성은 프로그램을 이해하기 어렵게 만들고
디버깅을 힘들게 하는 이유 중의 합니다.
Java는 multiple inheritance를 금지하고
오직 단 하나의 parent만을 가능하게 했기 때문에
class hierarchy가 명확해지고
error를 훨씬 줄일 수 있습니다.
두번째 rule로는 parent의 private field들은
child class에서도 direct access가 불가능하다는 것입니다.
따라서 parent class의 private field를 access 하기 위해서는
child에서 accessor와 mutator를 이용해야 하는 것입니다.
세번째 rule로는 child class와 parent class가 다른 package안에 있을 때에는
parent class의 default fields와 methods를
child에서 direct access가 불가능하다는 것입니다.

Class Hierarchy



(child → parent)

- A is a parent of B, C
- A is a grand parent of D, E, F
- A is an ancestor of B, C, D, E, F, G

- G is a child of F
- G is a grand child of C
- G is a descendant of A, C, F

- B is not an ancestor of E
- D is not a descendant of C

페이지 4

class hierarchy를 나타내는 tree를 그림과 같이 그려 놓았습니다.
tree의 edge는 child에서 parent로 가는 arrow로 표시되었습니다.
이 hierarchy를 보면서 class들의 관계를 해석해 볼 수 있습니다.
먼저 class A는 B와 C의 parent이고
따라서 A는 D, E, F의 grand parent입니다.
결국 A는 B, C, D, E, F, G의 ancestor입니다.

한편, G는 F의 child이고
G는 C의 grand child 입니다.
따라서, G는 A, C, F의 descendant입니다.

반면에 B는 E의 ancestor가 아닙니다.
또한 D는 C의 descendant가 아닙니다.

Example: AnimalTest (1/7)

```java
public class Animal {
    private String name;  // private access
    private int age;      // private access

    // Default constructor
    public Animal() { }

    // Constructor with parameters
    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Accessors and Mutators
    public String getName() {
        return name;
    }
```


페이지 5

class Animal에는 private instance variable name과 age가 있습니다.
하는 일이 없지만 권장하는대로 만들어진 default constructor도 있습니다.
두 instance variable의 초기값을 parameter로 받는
constructor도 있습니다.
instance variable name이 private이기 때문에
그 값을 읽어서 return 해 주는 public method getName() 이
accessor로 define 되었습니다.

Example: AnimalTest (2/7)

```
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    // Method makesound()
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
    @Override // why? see later chapters
    public String toString() {
        return "Animal{name='" + name + "', age=" + age + '}';
    }
}
```


페이지 6

역시 instance variable name의 값을 바꾸어 주기 위해서는
mutator인 public void setName이 정의되어야 합니다.
private instance variable인 age를 위한
accessor인 getAge()와 mutator인 setAge()도 define 되었습니다.
public method인 makeSound() 는
어떤 동물의 sound를 발생시킨다는 message를 print하고 있습니다.
그 아래 toString() method가 재정의 되었는데
재정의를 영어로는 "overriding" 이라 합니다.
이렇게 overriding 을 위한 method 앞에는
@Override 라는 표시가 IntelliJ 에 의해 자동으로 붙여지는데
이렇게 at (@) 표시 뒤에 붙는 특정 표시를
annotation이라 합니다.
사실 annotation은 붙이지 않아도 전혀 상관이 없습니다만
우리 IDE가 친절하게 그 아래에 define된 method가
overriding 하고 있는 method라는 사실을 알려주는
annotation @Override를 붙여주는 것입니다.
우리도 overriding (재정의) 할 때에는 @Override annotation을
붙이는 습관을 들이는 것이 좋습니다.

Example: AnimalTest (3/7)

```java
public class Cat extends Animal { // inherit Animal class
    private String color;  // more instance variable

    public Cat() {
        super();    // must call the parent constructor first
    }

    // name and age are private in the parent class, so cannot directly accessed
    public Cat(String name, int age, String color) {
        super(name, age); // call the parent constructor instead of direct access
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
```


페이지 7

Cat class는 Animal class를 inherit 합니다.
Animal의 기존 instance variable들 (name, age) 에 더해
color를 instance variable로 더 추가합니다.
Cat의 default constructor에서는 super() 를 call 했는데
이와 같이 child class에서는 자신의 parent class의
대응하는 constructor를 call해 주어야 합니다.
또 이 call은 constructor에서 가장 먼저 실행되어야 합니다.
다음에 나오는 constructor에서는
name, age, color의 초기값을 parameter로 받는데
name과 age는 parent class인 Animal의 instance variable들입니다.
그런데 그들이 Animal의 private이기 때문에
child인 Cat에서도 direct access가 불가능합니다.
그런 이유도 있고 해서 여기서도 super(name, age) 를
가장 먼저 call합니다.
이렇게 하면 Animal 의 constructor Animal(name, age)가
call 되며, private 문제도 해결됩니다.
남은 parameter인 color는 this.color에 assign합니다.
그 아래의 getColor, setColor는 accessor와 mutator method들입니다.

Example: AnimalTest (4/7)

```
    // Method overriding: change the original method
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }

    // toString method
    @Override
    public String toString() {
        return "Cat{name='"+getName()+ "', age=" + getAge() + ", color='" + color + "'}";
    }
}

public class Dog extends Animal {
    private String breed; // more instance variable

    // Default constructor
    public Dog() {
        super();  // call the parent constructor
    }
```


페이지 8

이번에는 inherit된 method makeSound를
overriding하여 새로운 기능을 가지게 하려고 합니다.
overriding method는 overloading과 달리
ancestor의 method와 정확히 똑같은 method header를
가지고 있어야 합니다.
이 경우에는 Animal의 'public void makeSound()' 를
overriding하여 Cat의 'public void makeSound()' 로
다시 정의하는 것입니다.
여기서는 고양이 울음소리인 "Meow" 를 print하도록 했습니다.
그 아래에는 toString() 이 overriding 되었습니다.

이제 새로운 class Dog를 Animal의 child로 정의하기 시작합니다.
Dog에는 String breed 라는 새 instance variable이 추가되었는데
breed는 개의 품종을 말합니다.
역시 default constructor에서는 super() 로
parent의 default constructor를 call하였습니다.
그러나 역시 default constructor는 아무 하는 일이 없습니다.

Example: AnimalTest (5/7)

```
    // Constructor with parameters
    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }

    // Accessors and Mutators
    public String getBreed() {
        return breed;
    }

    public void setBreed(String breed) {
        this.breed = breed;
    }

    // Method overriding
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
```


페이지 9

Dog의 두번째 constructor는
name, age, breed를 parameter로 받아
super(name, age) 를 call하여 parent의 constructor를 실행되게 했고
this.breed에 breed를 assgin하였습니다.
그 아래 두개는 private instance variable인 breed를 위한
accessor와 mutator들입니다.
Dog에서도 makeSound method를 overriding하였는데
여기서는 개가 짖는 "Bark" 소리를 print하였습니다.

Example: AnimalTest (6/7)

```
    public String toString() {
        return "Dog{name='"+getName()+"', age=" + getAge() + ", breed='" + breed + "'}";
    }
}

public class AnimalTest {
    public static void main(String[] args) {
        Animal animal = new Animal("Generic Animal", 5);  // Animal object
        System.out.println(animal);
        animal.makeSound();  // print “Some generic animal sound”

        Dog dog = new Dog("Buddy", 3, "Golden Retriever");  // Dog object
        System.out.println(dog);
        dog.makeSound();     // print “Bark”

        Cat cat = new Cat("Whiskers", 2, "Black");  // Cat object
        System.out.println(cat);
        cat.makeSound();     // print “Meow”
    }
}
```


페이지 10

그 다음에는 toString()을 overriding 하였습니다.
AnimalTest class에서는
지금까지 정의한 Animal, Cat, Dog class들이
잘 작동하는지를 test합니다.
먼저 “Generic Animal” 이라는 name과 age 5를 가지는
animal object를 생성하였습니다.
toString을 이용하여 animal의 정보를 print하였고
animal의 makeSound()를 call 하여
“Some generic animal sound” 를 print합니다.
이번에는 name이 Buddy이고 age 3이며
breed가 Golden Retriever인 Dog object를 생성합니다.
dog의 정보를 print하고
makeSound를 call하여 “Bark” 라고 프린트 합니다.
Cat에 대해서도 비슷한 test가 이루어지고 있습니다.

Example: AnimalTest (7/7)

```
OUTPUT:

Animal{name='Generic Animal', age=5}
Some generic animal sound

Dog{name='Buddy', age=3, breed='Golden Retriever'}
Bark

Cat{name='Whiskers', age=2, color='Black'}
Meow
```


페이지 11

이 slide는 AnimalTest 프로그램의 output을 보여주고 있습니다.

Example: VehicleTest (1/4)

```java
public class Vehicle {
    private String brand;
    private int year;

    public Vehicle() { }

    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }
```


페이지 12

이번에는 Vehicle class를 보겠습니다.
이 class에는 brand를 나타내는 String과
출시년도를 나타내는 int year 라는
instance variable들이 있습니다.

default constructor가 있고
brand와 year를 초기화 하는 constructor가 있습니다.
instance variable들이 모두 private이므로
accessor와 mutator가 있습니다.

Example: VehicleTest (2/4)

```
    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public void startEngine() {
        System.out.println("The engine is starting...");
    }

    @Override
    public String toString() {
        return "Vehicle{brand='" + brand + "', year=" + year + '}';
    }
}
```


페이지 13

private instance variable인 year에 대한
accessor와 mutator가 있고
startEngine이라는 method에서는
"The engine is starting ..." 이라는 string을 print합니다.
맨 아래는 toString() method가 overriding 되어 있습니다.

Example: VehicleTest (3/4)

```java
public class Car extends Vehicle {
    private int doors; // more instance variable

    public Car() {
        super();
    }

    public Car(String brand, int year, int doors) {
        super(brand, year);
        this.doors = doors;
    }

    public int getDoors() {
        return doors;
    }

    public void setDoors(int doors) {
        this.doors = doors;
    }
```


페이지 14

class Car가 Vehicle의 child class로 정의되어 있습니다.
문의 갯수를 나타내는 doors가 instance variable로 추가되어 있습니다.
default constructor에서는 super()를 사용하여
parent인 Vehicle의 default constructor를 call하였습니다.
brand, year, doors를 parameter로 받는 constructor에서는
먼저 brand와 year의 초기값을 set하는 parent의 constructor를 call하고
doors의 초기값을 assign하였습니다.
getDoors()와 setDoors()는 doors를 위한 accessor, mutator입니다.

Example: VehicleTest (4/4)

```
    // method overriding
    @Override
    public void startEngine() {
        super.startEngine(); // call the parent's method
        System.out.println("The car engine is starting...");
    }

    @Override
    public String toString() {
        return "Car{brand='"+getBrand()+"', year="+getYear() + ", doors=" + doors + '}';
    }
}
```

```
OUTPUT:
Vehicle{brand='Generic Vehicle', year=2010}
The engine is starting...
Car{brand='Toyota', year=2020, doors=4}
The engine is starting...
The car engine is starting...
```


페이지 15

startEngine() method를 overriding 했는데
먼저 super.startEngine()을 call하여
parent의 startEngine()을 실행하여
"The engine is starting..." 이라는 메시지를 프린트하고
거기에 "The car engine is starting..." 이라는 메시지를
더 프린트 하였습니다.
마지막으로 toString() method를 overriding하여
Car object의 정보를 print하게 했습니다.

’protected’ Access members

- Can be directly accessed from
  - other classes in the same package
  - any descendant classes

페이지 16

여기에서 ‘protected’ access에 대해 다시 살펴 보겠습니다.
‘protected’ 로 정의된 member는
먼저 class 외부라 할지라도 같은 package 내에서는 access가 가능하며
descendant class에서는 (심지어 같은 package 내에 있지 않더라도)
access가 가능합니다.
특히 descendant class에는 그 ancestor class의 member가
정의에 포함되어 있기 때문에
object와 dot operator를 제외한 direct access가 가능해 집니다.
이것에 대해서는 다음 slide에서 더 자세히 살펴보겠습니다.

Example: ProtectedExample

```java
public class Parent {
    protected String name = "Parent Name";
    protected void display() {
        System.out.println("This is a protected method in Parent class.");
    }
}

class Child extends Parent {
    public void showName() {
        System.out.println("Name: " + name); // read parent's protected variable
        display(); // call the parent's protected method
    }
}

public class ProtectedExample {
    public static void main(String[] args) {
        Child c = new Child();
        c.showName();
    }
}
```

```
OUTPUT:
Name: Parent Name
This is a protected method in Parent class.
```


페이지 17

Parent class에 String name instance variable과 display() method가
protected access로 정의되어 있습니다.
Child class가 Parent class를 inheirt 하면
name과 display()는 자동으로 자신의 member가 되며
그들이 protected이기 때문에 dot operator 없이
direct access 가 가능해 집니다.
showName method 안에서 name이 direct access되었으며
display() method call도 direct access되었습니다.
ProtectedExample class program에서는
Child class c를 생성하고 c.shoname()을 call하였는데
"Name: Parent Name" 과
"This is a protected method in Parent class."가
print됩니다.

Summary of Access Modifiers

- private: access only from the same class
- default (package): access only from the same package
- protected: access from the same package, from the descendant class in other packages
- public: no restriction

Access Modifier	The same Class	The same Package	Descendants	Everywhere
public	✓	✓	✓	✓
protected	✓	✓	✓	
default (package)	✓	✓		
private	✓			

페이지 18

이제 access modifier들에 대해 요약해 보겠습니다.
private은 같은 class 내에서만 access가 가능합니다.
default 또는 package 권한은 같은 package 내에서만 access가 가능합니다.
protected는 같은 pakcage이내와 함께
descendant들에서는 access가 가능합니다.
마지막으로 public은 아무런 제한이 없이 어디에서나 access가 가능합니다.