06_3 Abstract Class

Object-Oriented Programming

Abstract class에 대해 강의하겠습니다.

Abstract Class

- Created by extracting the common characteristics (fields and methods) of concrete classes that can create objects.
- There can be one or more abstract methods in an abstract class.
- Abstract method has only the signature, no body.
- Abstract class cannot make an object.

페이지 2

우리가 지금까지 만들어 써 왔던 class는 concrete class라고 부릅니다.
Abstract class는 여러개의 concrete class들 중 공통된 성질,
즉 field나 method들을 추출하여 만들 수 있는 class를 말합니다.
~~Abstract class에는 적어도 한 개 이상의 abstract method가 존재해야 합니다.~~
Abstract class에는 abstract method가 존재할 수도 있고 존재하지 않을 수도 있습니다.
Abstract method는 signature만 존재하고 body는 없는 method를 말합니다.
Abstract method의 존재여부와 상관없이 Abstract class로는 object를 생성할 수 없습니다.

Example) PaymentDemo (1/4)

```java
abstract class Payment { // Abstract class: cannot make an object

    abstract void makePayment(double amount); // Abstract method (no body)

    void transactionDetails() { // Regular method
        System.out.println("Processing payment...");
    }
}

class CreditCardPayment extends Payment { // (Concrete) Subclass
    void makePayment(double amount) { // abstract method implementation
        System.out.println("Payment of $" + amount + " Credit Card.");
    }
}
```


페이지 3

실제로 abstract class를 만들어 사용하는 예를 보도록 하겠습니다.
class keyword 앞에 'abstract' 라는 keyword를 더 붙여서
abstract class를 정의합니다.
이 abstract class는 object를 만들지 못합니다.
abstract class 안에는 적어도 한 개 이상의 abstract method가 있는데
여기에서는 'makePayment' 가 abstract method입니다.
abstract method는 signature만 있고, body가 없습니다.
그 아래의 transactionDetails() 는 regular method입니다.
Abstract class가 object 즉 instance를 만들지 못한다는 것은
그 abstract class를 inherit한 descendant class를
concrete class로 정의하여 사용해야 한다는 뜻입니다.
여기서는 첫번째 concrete subclass로 'CreditCardPayment' class를
정의하여 abstract class인 Payment를 inherit하였습니다.
이렇게 concrete subclass가 되려면
parent인 abstract class가 정의를 끝마치지 않은
abstract method의 definition을 완성해야 합니다.
여기에서는 makePayment의 body를 implement하여
payment amount가 credit card로 이루어진다는
메시지를 프린트 하였습니다.

Example) PaymentDemo (2/4)

```
class PayPalPayment extends Payment { // Another (concrete) subclass
    // Providing implementation of abstract method
    void makePayment(double amount) {
        System.out.println("Payment of $" + amount + " PayPal.");
    }
}

class BankTransferPayment extends Payment { // concrete subclass
    // Providing implementation of abstract method
    void makePayment(double amount) {
        System.out.println("Payment of $" + amount+“ Bank Transfer.");
    }
}
```


페이지 4

비슷한 방법으로 이제 PayPalPayment concrete subclass를
Payment의 child로 구현하였습니다.
이번 makePayment는 'PayPal' 을 이용하여 이루어진다는
메시지를 프린트 하였습니다.
또 하나의 concrete subclass는 BankTransferPayment입니다.
여기서도 abstract method인 makePayment의
implementation을 완성했는데
amount의 금액이 Bank Transfer를 이용하여 지불된다는
메시지를 프린트합니다.

Example) PaymentDemo (3/4)

```
public class PaymentDemo {
    public static void main(String[] args) {
        // Payment payment = new Payment(); //Error:no instance allowed

        Payment creditCardPayment = new CreditCardPayment();
        Payment payPalPayment = new PayPalPayment();
        Payment bankTransferPayment = new BankTransferPayment();

        creditCardPayment.transactionDetails();
        creditCardPayment.makePayment(100.50);

        payPalPayment.transactionDetails();
        payPalPayment.makePayment(250.75);

        bankTransferPayment.transactionDetails();
        bankTransferPayment.makePayment(400.00);
    }
}
```

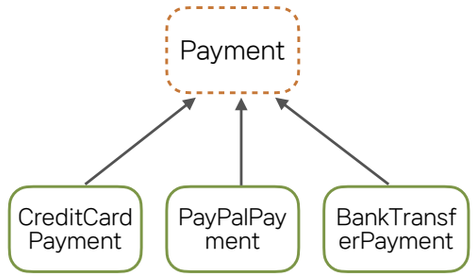

페이지 5

이제 PaymentDemo에서 앞에서 정의된 class hierarchy를 이용하여
데모 프로그램을 구현하였습니다.
먼저 abstract class인 Payment의 object를 생성하려하였으나
이것은 error가 됩니다.
abstract class의 instance는 만들 수 없기 때문입니다.
이제 CreditCardPayment, PayPalPayment, BankTransferPayment의
Payment의 concrete children class들의
object를 하나씩 생성하였습니다.
그리고 각각의 object마다 transactionDetails() method를 call 한 후
makePayment method를 call 하여
payment message를 출력하게 하였습니다.

Example) PaymentDemo (4/4)

```
OUTPUT:

Processing payment...
Payment of $100.5 Credit Card.
Processing payment...
Payment of $250.75 PayPal.
Processing payment...
Payment of $400.0 Bank Transfer.
```



페이지 6

이것은 앞의 demo program의 output입니다.
그림은 이 프로그램의 class hierarchy를 보여주고 있는데
abstract class인 Payment는 점선으로 나타내었습니다.

(Concrete) Subclass should implement abstract classes (1/2)

```
abstract class Player {
    int currentPos;

    Player() {
        currentPos = 0;
    }

    abstract void play(int pos);  // abstract method
    abstract void stop();         // abstract method

    void play() { // overloaded play(), not abstract
        play(currentPos);
    }
}
```

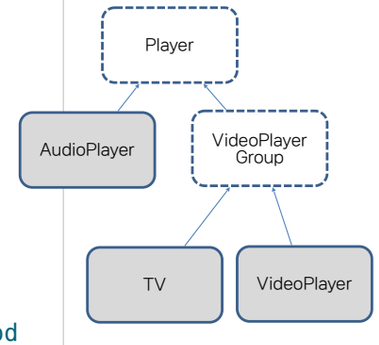

페이지 7

여기에 Player라는 abstract class가 있습니다.
현재 play되는 position을 나타내는 instance variable인
currentPos를 가지고 있습니다.
default constructor에서는 currentPos를 0으로 초기화합니다.
그리고 두 개의 abstract method play와 stop이 있습니다.
마지막에 있는 play() method는 abstract가 아니라 concrete method인데
위의 abstract method play(int pos)와는 다른 것으로
overloaded 되어 있습니다.
두 method의 parameter가 다른 것을 유의해야 합니다.

(Concrete) Subclass should implement abstract classes (2/2)

```java
class AudioPlayer extends Player { // concrete class
    void play(int pos) {
        System.out.println("play audio from " + pos);
    }
    void stop() {
        System.out.println("stop audio");
    }
}

// another abstract class in the class hierarchy
abstract class VideoPlayerGroup extends Player {
    // doesn't implement play(int pos) and stop()
    abstract void diplayOn(); // another abstract method
}
```



TV and VideoPlayer should implement play(int pos), stop(), and displayOn()

페이지 8

AudioPlayer는 Player를 inherit한 concrete class입니다.
Concrete class가 되려면 parent인 abstract class의
abstract method들을 모두 다 implement 해야 합니다.
따라서 play(int pos) 와 stop(), 두 개의 method를
모두 implement하였습니다.
VideoPlayerGroup은 Player를 inherit하였는데
abstract method들을 implement하지 않았고
따라서 그 자신이 다시 abstract class가 됩니다.
게다가 자신이 abstract method를 하나 더 추가했는데
displayOn() 이 그것입니다.
그림의 class hierarchy를 보면
Player와 VideoPlayerGroup은 abstract class로
점선으로 표시되어 있으며
AudioPlayer, TV, VideoPlayer의 세 class들은
concrete class들 입니다.
특히 TV와 VideoPlayer의 두 concrete class들은
play(int pos), stop()과 함께 displayOn() 이라는
abstract method를 모두 implement해야 합니다.

Designing Abstract Class (1/2)

```
class Marine {
    int x, y;
    void move(int x, int y) { ... } // move to (x,y)
    void stop() { ... } // stop at current position
    void stimPack() { ... } // using stimPack (무기 한 종류)
}
class Tank {
    int x, y;
    void move(int x, int y) { ... }
    void stop() {...}
    void changeMode() { } // change attack mode
}
class Dropship {
    int x, y;
    void move(int x, int y) { ... }
    void stop() { ... }
    void load() { ... }  // load the selected object
    void unload() { ... } // unload the selected object
}
```

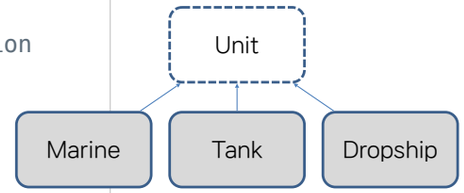
- Extract common parts from existing classes to construct the abstract class

페이지 9

여기에서는 비슷한 형태의 class들을 묶어서 그 공통된 variable들과 method를
parent class의 소유로 만들어서
class들을 hierarchical하게 조직하는 과정을 보여주고 있습니다.
class Marine, Tank, Dropship은 모두
어떤 game에 출현하는 character들로서
자신의 현재 위치인 x, y와
한번에 움직이는 방법을 나타내는 move(int x, int y) method
그리고 현재 위치에서 움직임을 멈추는 stop() method를
공통적으로 가지고 있습니다.
이럴 경우 이 세 class의 parent를 만들어
공통부분을 묶어주면
class들의 정의를 중복되는 부분을 제거하면서
효율적으로 관리할 수 있습니다.
그런데, 간혹 이렇게 만들어진 parent class가
그 자신의 object를 만들기에는 적절하지 않는 경우가 있습니다.
예를 들면 Cat, Dog, Horse, Lion 등의 parent를 Animal 이라 할때
Animal의 object를 딱히 생성할 필요성은 없는 것입니다.
이런 경우 그 parent를 abstract class로 만들어서
object를 생성할 수 없게 하는 것이
적절해 보이는 것입니다.

Designing Abstract Class (2/2)

```java
abstract class Unit {
    int x, y;
    abstract void move(int x, int y);
    void stop() { ... } // stop at current position
}
class Marine extends Unit {
    void move(int x, int y) { ... }
    void stimPack() { ... } // using stimPack
}
class Tank extends Unit {
    void move(int x, int y) { ... }
    void changeMode() { } // change attack mode
}
class Dropship extends Unit {
    void move(int x, int y) { ... }
    void load() { ... }  // load the selected object
    void unload() { ... } // unload the selected object
}
```



페이지 10

이제 우리는 Marine, Tank, Dropship의 parent로써
Unit 이라는 abstract parent class를 만들었습니다.
Unit 에는 세 개의 child class들이 공통적으로 가지고 있는
int x, y와
abstract method move를 가지고 있으며
stop method도 concrete method로 가지고 있습니다.
남은 세개의 child class들은
공통된 부분을 제외하고
각자 더 필요한 variable이나 method들을 가지고 있으며
특히 abstract method인 move를 반드시 구현하여
가지고 있어야 합니다.

Abstract Class Can be Used as Parameter (1/3)

```java
abstract class Document { // Abstract class
    abstract void printContent(); // Abstract method
}

class PDFDocument extends Document { // Subclass
    void printContent() { // implementation of abstract method
        System.out.println("Printing PDF..");
    }
}

class WordDocument extends Document { // Subclass
    void printContent() { // implementation of abstract method
        System.out.println("Printing Word..");
    }
}
```


페이지 11

Abstract class의 object를 생성하는 것은 불가능하지만
abstract class variable을 method parameter로 사용하여
자신의 descendant class object를
parameter로 받을 수 있는 예를 살펴봅니다.
먼저 Document라는 abstract class가 있고
그 안에 printContent() 라는 abstract method를 가지고 있습니다.
PDFDocument는 Document의 subclass로
printContent를 implement하여 concrete class가 되었습니다.
WordDocument도 또 하나 다른 Document subclass로서
printContent를 implement하였습니다.

Abstract Class Can be Used as Parameter (2/3)

```
class Printer {
    void printDocument(Document doc) { // parameter: abstract class
        System.out.println("Starting..");
        doc.printContent();
        System.out.println(”Job completed");
    }
}
```


페이지 12

Printer class에는 printDocument라는 method가 있는데 parameter로 Document abstract class를 받고 있고 doc.printContent() 를 call 하고 있습니다.

Abstract Class Can be Used as Parameter (3/3)

```java
public class DocumentDemo {
    public static void main (String[] args) {
        // Create instances of subclasses
        Document pdf = new PDFDocument();
        Document word = new WordDocument();

        // Create Printer instance
        Printer printer = new Printer();

        // Use Printer to print documents
        printer.printDocument(pdf);
        // Output: Starting.. Printing PDF.. Job completed
        printer.printDocument(word);
        // Output: Starting.. Printing Word.. Job completed
    }
}
```


페이지 13

여기서는 먼저 두 개의 Document object를 생성했는데
pdf는 original class가 PDFDocument 이고
word는 original class가 WordDocument입니다.
그리고 Printer object인 printer를 하나 생성하여
printer.printDocument를 call하는데
한번은 pdf를, 다른 한번은 word를 parameter로 pass하였습니다.
이처럼 abstract class를 parameter로 하였을 경우
그 child인 concrete class들을 parameter로 pass하여
사용할 수 있습니다.