

5

제어구조

지도 프로그램에 사용되었던 조건문

```
/** Location 클래스: 위도 경도 표현 및 다른 위치까지의 거리 계산 */
```

```
public class Location
```

```
{ /** 생성자: 초기화한다 */
```

```
    private double latitude;
```

```
    private double longitude;
```

```
    ...
```

```
    public void setLatitude(double lat) {
```

```
        if (lat < -90 || lat > 90) {
```

```
            System.out.println("Illegal value of latitude!");
```

```
        }
```

```
        else { this.latitude = lat; }
```

```
    }
```

- 외부에서 유효하지 못한 값으로 경도를 수정하는 것을 방지
- 유효한 값이 들어올 경우만 값 변경 허용

은행계좌관리

입력

 명령 D/W/Q와 금액을 입력하세요.

취소 승인

입력

 명령 D/W/Q와 금액을 입력하세요.

취소 승인

Deposit of \$ 50.50
 Current balance = \$50.50
 Withdrawal of \$ 30.33
 Current balance = \$20.17

입력

 명령 D/W/Q와 금액을 입력하세요.

취소 승인

제어

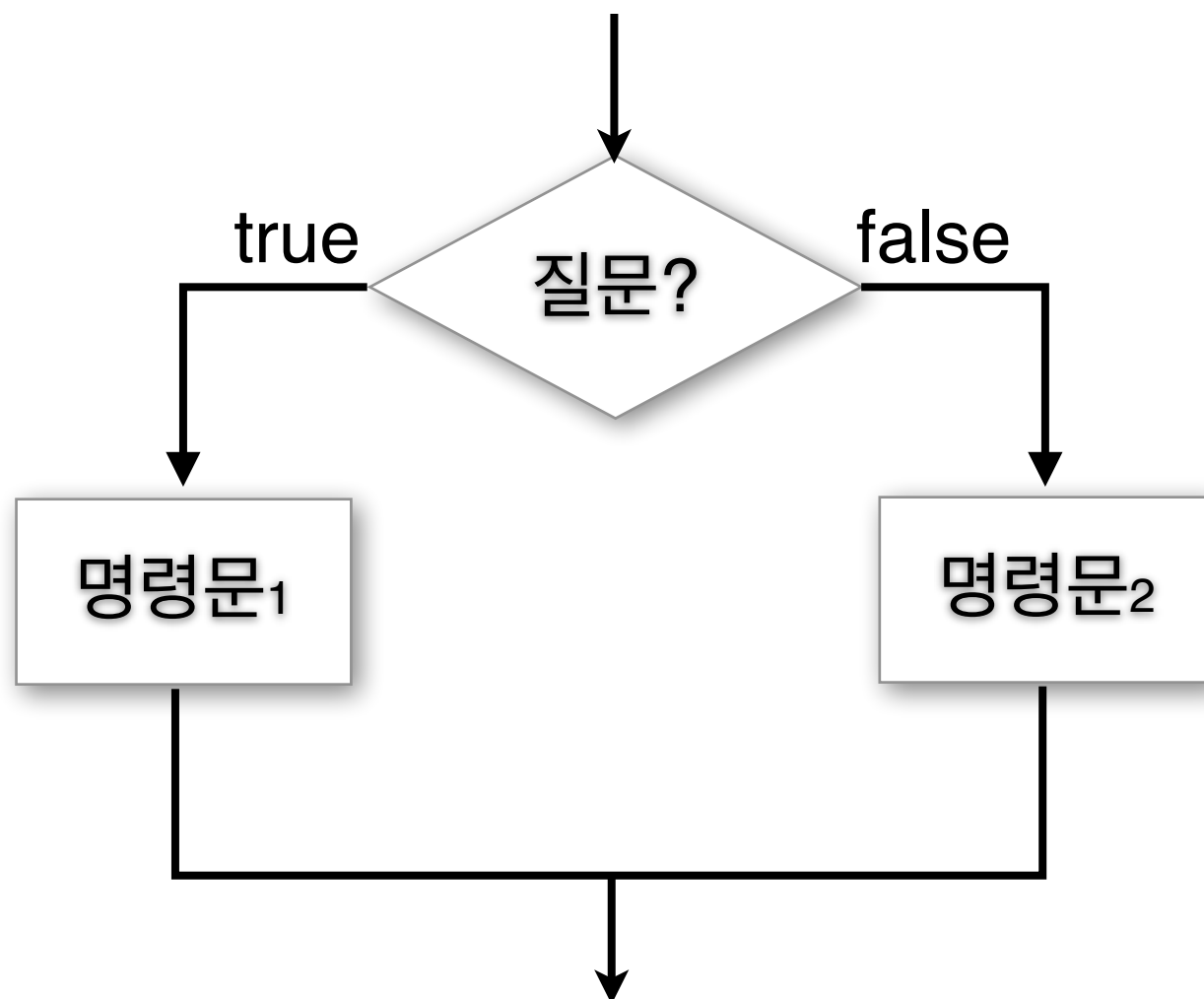
- 제어 흐름 (control flow)
 - 프로그램의 명령문이 수행되는 순서
- 제어구조
 - 순차: 명령문1; 명령문2; 명령문3;
 - 메소드 호출: 수취객체.메소드(인수);
 - 조건문
 - ...

조건제어구조 (Conditional Control Structure)

- 질문이 참이냐 거짓이냐에 따라 두 가지 명령문 수행이 가능한 구조
- 문법
 - `if( 질문 ) 명령문1 else 명령문2`
 - `if( 질문 ) { 명령문들1 } else { 명령문들2 }`
- 의미구조
 - 질문이 참이면 명령문1을 수행한다.
 - 질문이 거짓이면 명령문2를 수행한다.
 - 반대쪽은 무시된다.

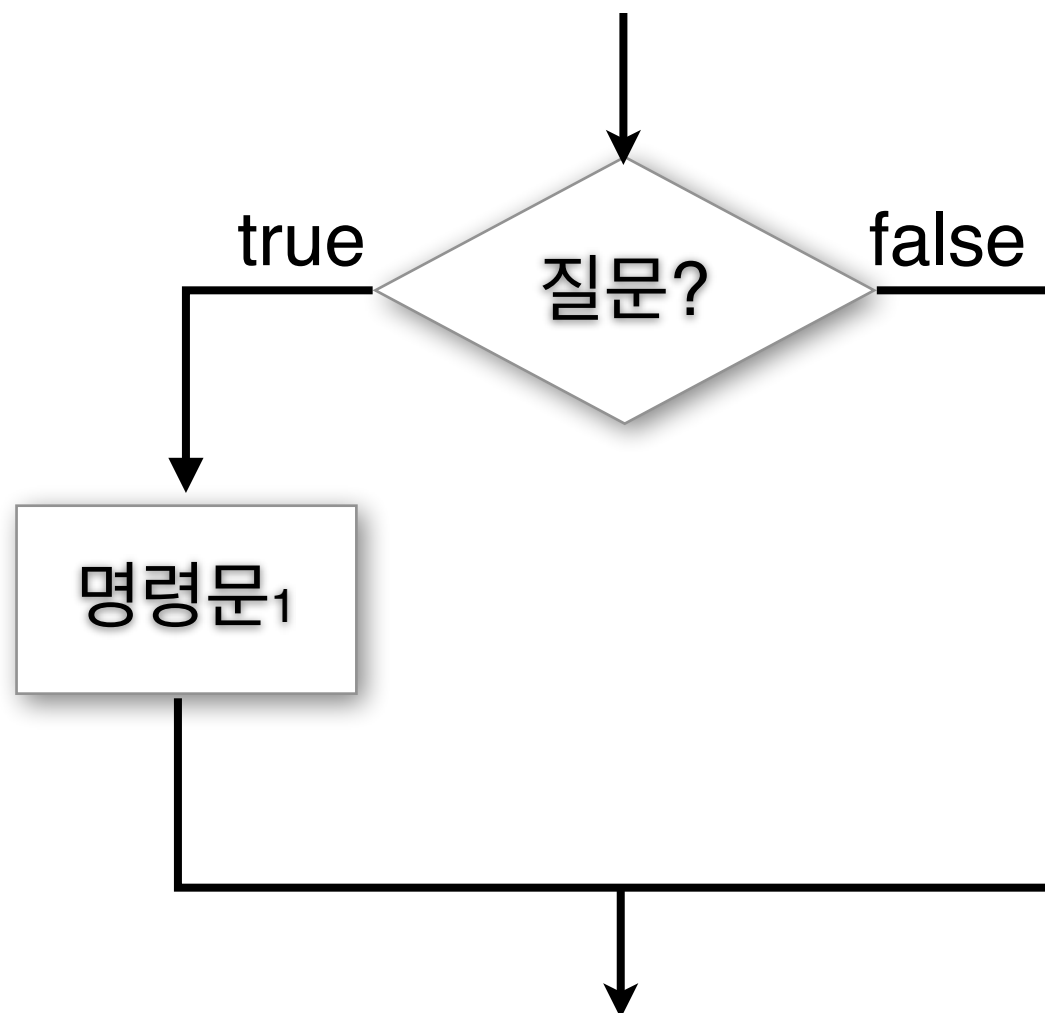
그림으로 보는 조건문의 제어흐름

```
if (질문?)  
{  
    명령문1  
}  
else  
{  
    명령문2  
}
```



else는 생략 가능하다

```
if (질문?)  
{  
  명령문1  
}
```



예, 음수냐 양수냐

```
/** printPolarity: 인수가 음수인지 양수인지 출력
 * @param i - 인수 */

public void printPolarity(int i)
{
    System.out.print("인수 " + i + "는 ");
    if ( i < 0 )
        System.out.print("음수");
    else
        System.out.print("양수");
    System.out.println("입니다.");
}
```


예, 잘못된 입력 걸러내기: 시를 초로

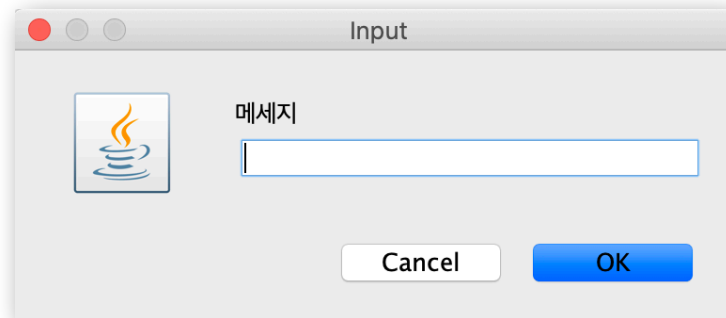
```
import javax.swing.*;
/** ConvertHours: 시를 초로 변환
 *  입력: 양수
 *  출력: 입력 시간을 초로 변환한 수 */

public class ConvertHour
{
    public static void main(String[] args) {
        String s = JOptionPane.showInputDialog("시를 입력하세요:");
        int hours = Integer.parseInt(s);
        if ( hours >= 0 ) {
            // 시가 양수이기 때문에 초를 계산할 수 있다.
            int seconds = hours * 60 * 60;
            JOptionPane.showMessageDialog(null, hours + " 시간은 " +
                                         seconds + " 초입니다.");
        }
        else {
            // 시가 음수이기 때문에 오류이다.
            JOptionPane.showMessageDialog(null,
                                         "ConvertHours error: negative input " + hours);
        }
    }
}
```

입력창 띄우기

```
import javax.swing.*;

String s =
JOptionPane.showInputDialog("메시지");
```



- new를 사용해서 객체를 생성할 필요가 없다.
- JOptionPane 가 모든 일을 다 해 준다.
 - 메시지 창을 만들고 입력을 받아서 받은 입력을 넘겨 준다.
 - 취소(Cancel)클릭 시 null을 넘겨준다 (null = 값이 없음)

입력창 띄우기

```
import javax.swing.*;
/** ConvertHours: 시를 초로 변환
 *  입력: 양수
 *  출력: 입력 시간을 초로 변환한 수 */

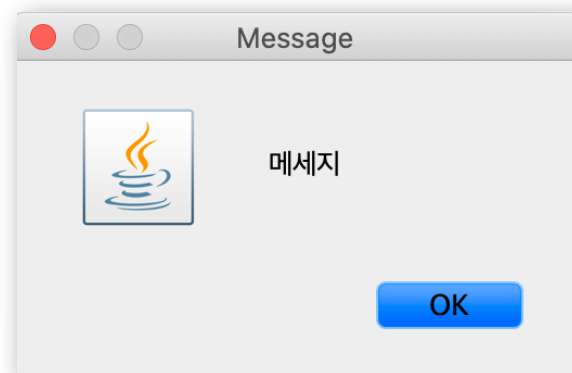
public class ConvertHour
{
    public static void main(String[] args) {

        String s = JOptionPane.showInputDialog("시를 입력하세요.");
        int hours = Integer.parseInt(s);
        ...
    }
```

문자열 → 정수 변환

결과창 띄우기

```
import javax.swing.*;  
  
JOptionPane.showMessageDialog(null, "메시지");
```

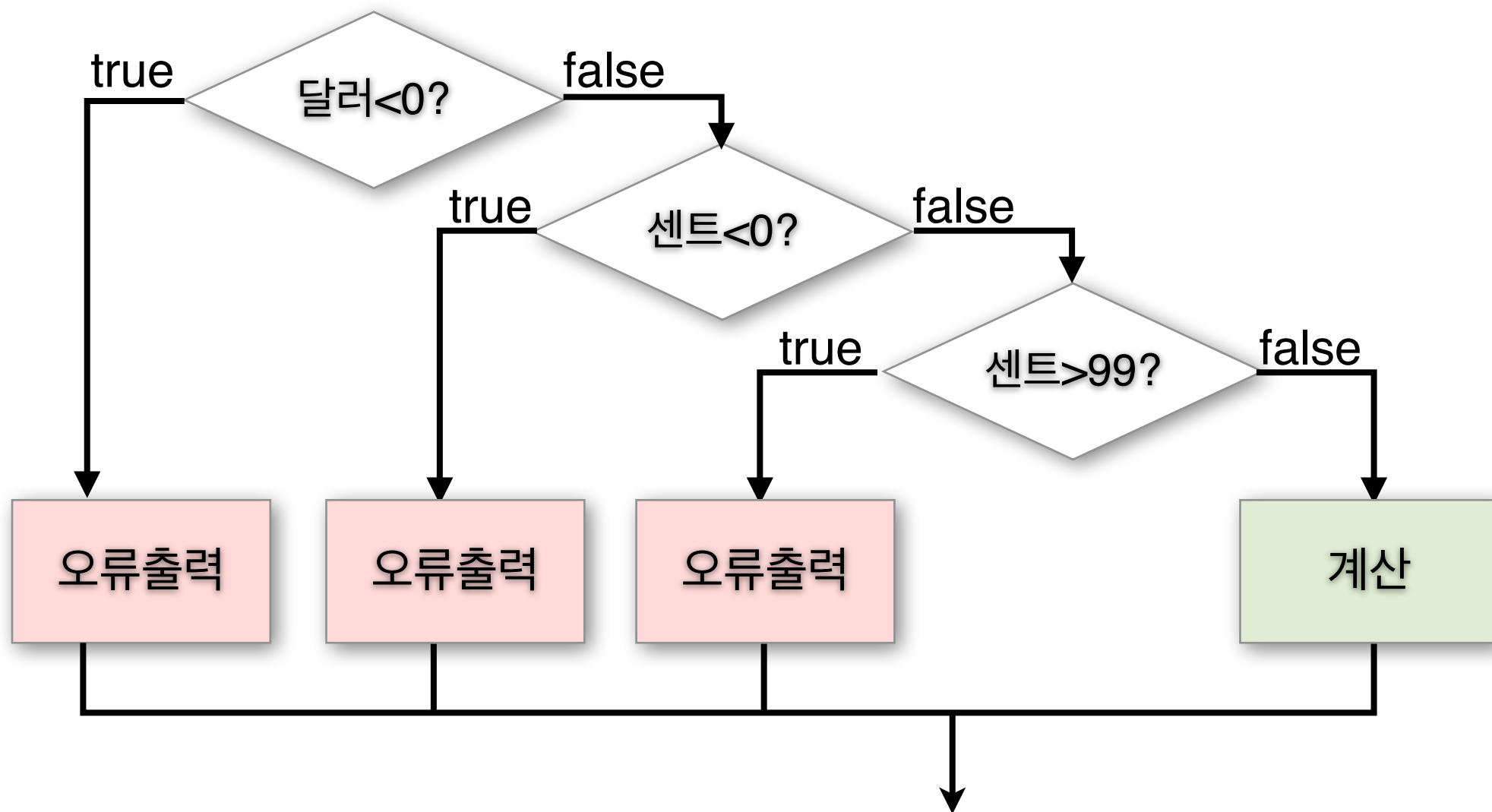


- new를 사용해서 객체를 생성할 필요가 없다.
- JOptionPane 가 모든 일을 다 해 준다.
 - 메시지 창을 만들고
 - “승인” 단추까지 만들어 주고
 - 단추를 누르면 창을 닫는다.

계속되는 조건문 (Nested Conditional)

- 달러와 센트를 입력으로 받아, 필요한 동전(quarter, dime, nickel, penny)의 수를 구하라.
 - 달러는 양수, 센트는 100보다 작은 양수인지 검사해야 한다.
- 알고리즘
 - 달러를 입력으로 받는다.
 - 달러가 음수? 오류메시지 출력
 - 아니면,
 - 센트를 입력 받는다.
 - 센트가 음수? 오류메시지 출력
 - 아니면,
 - 센트가 99보다 큰가? 오류메시지 출력
 - 아니면, 동전의 개수를 계산해서 출력한다.

제어 흐름



```

import javax.swing.*;
public class MakeChangeAgain {
    public static void main(String[] args) {
        String s = JOptionPane.showInputDialog("Type dollars, an integer:");
        int dollars = Integer.parseInt(s);
        if (dollars < 0) {
            JOptionPane.showMessageDialog(null, "MakeChangeAgain error: negative dollars: " + dollars);
        }
        else { // dollars amount is acceptable, so process cents amount:
            s = JOptionPane.showInputDialog("Type cents, an integer:");
            int cents = Integer.parseInt(s);
            if (cents < 0) {
                JOptionPane.showMessageDialog(null, "MakeChangeAgain error: negative cents: " + cents);
            }
            else {
                if (cents > 99) {
                    JOptionPane.showMessageDialog(null, "MakeChangeAgain error: bad cents: " + cents);
                }
                else { // dollars and cents are acceptable, so compute answer:
                    int money = (dollars * 100) + cents;
                    String output = " quarters = " + (money/25);
                    money = money % 25;
                    output = output + "\n dimes = " + (money/10);
                    money = money % 10;
                    output = output + "\n nickels = " + (money/5);
                    money = money % 5;
                    output = output + "\n pennies = " + money;
                    JOptionPane.showMessageDialog(null, output);
                }
            }
        }
    }
}

```

중첩을 줄여볼까?

```
import javax.swing.*;
public class MakeChange {
    ...
    public static void main(String[] args) {
        boolean ok = true;
        int dollars = getInt("달러?");
        int cents = getInt("센트?");
        if (dollars < 0)
        { showError(dollars + "는 음수입니다."); ok = false; }
        if (cents < 0)
        { showError(cents + "는 음수입니다."); ok = false; }
        if (cents > 99)
        { showError(cents + "가 100 이상입니다."); ok = false; }

        if (ok) { // 제대로 된 입력이니 계산을 시작해 볼까?
            ...
        }
    }
}
```


논리값의 관계연산자

- 논리값(boolean)의 관계연산자를 사용하면 조건문 구조를 단순하게 할 수 있다.

연산	이름	의미
$E1 \ \&\& \ E2$	논리곱 conjunction (and)	$\text{true} \ \&\& \ \text{true} \Rightarrow \text{true}$ $\text{false} \ \&\& \ E2 \Rightarrow \text{false}$ $E1 \ \&\& \ \text{false} \Rightarrow \text{false}$
$E1 \ \ E2$	논리합 disjunction (or)	$\text{false} \ \ \text{false} \Rightarrow \text{false}$ $\text{true} \ \ E2 \Rightarrow \text{true}$ $E1 \ \ \text{true} \Rightarrow \text{true}$
$!E$	논리역 negation (not)	$!\text{true} \Rightarrow \text{false}$ $!\text{false} \Rightarrow \text{true}$

논리 연산자를 사용해 보자.

```
import javax.swing.*;
public class MakeChange {
    ...
    public static void main(String[] args) {
        int dollars = getInt("달러?");
        int cents = getInt("센트?");
        if (dollars < 0 || cents < 0 || cents > 99)
            showError(dollars + " 또는 " +
                      cents + "는 잘못된 입력입니다.");
        else { // 제대로 된 입력이니 계산을 시작해 볼까?
            ...
        }
    }
}
```

주의: else 너 누구 소속이냐?

- `if (...) if (...) {...} else {...}` 의 해석은?

```
if (...)  
  if (...)  
    {...}  
else  
  {...}
```

○

```
if (...)  
  if (...)  
    {...}  
else  
  {...}
```

X

- 중괄호를 치는 것이 좋다.

주의: 논리 연산자는 단축계산된다

- 단축계산 (short-circuit evaluation)
 - 이항 연산자의 한 쪽만 보고 결과를 알 수 있을 때 다른 한 쪽을 계산하지 않는 것
- Java에서 논리연산식은 왼쪽에서 오른쪽으로 단축계산
 - | | | |
|----------|--|-----|
| x<0 | | y<0 |
| ==> -1<0 | | y<0 |
| ==> true | | y<0 |
| ==> true | | |
 - `x!=null && x.method()` 와
`x.method() && x!=null` 는 항상 같은 결과를 주지 않는다.

예, 24시간제를 12시간제로

- 24시간제로 시, 분을 입력받아 12시간제 문자열로 바꾸는 메소드를 작성하세요.
 - `twelveHourClock(16,5) ==> "오후 4:05"`
- 입력이 적당하지 않을 때는 오류 문자열을 반환하세요.
- 생각치 못한 고려사항
 - 1분은 01분으로

```

/** twelveHourClock: 24시간제를 12시간제로 변경
 * @param hour - 시 (0~23)
 * @param minute - 분 (0~59)
 * @return 12시간제 시간 (문자열) */

public String twelveHourClock(int hour, int minute)
{
    String answer = ""; // 답을 여기서 누적할 것이다.

    if ( hour < 0 || hour > 23 || minute < 0 || minute > 59 )
        answer = "입력 오류, " + hour + ":" + minute; // 오류
    else {
        if ( hour < 12 )
            answer = answer + "오전 ";           // 오전
        else
            answer = answer + "오후 ";           // 오후
        if ( hour >= 13 )
            answer = answer + (hour - 12);       // 시간이 13이상이면 12뺀 수로 붙인다.
        else
            answer = answer + hour;              // 0~12시의 경우 그대로 붙인다.
        answer = answer + ":";
        if ( minute < 10 )
            answer = answer + "0";              // 분이 한자리수인 경우 0을 추가
        answer = answer + minute;               // 분을 붙인다.
    }
    return answer;
}

```

다중 조건문 switch

- 변수에 대해 다음과 같은 조건문 패턴이 자주 나온다.
 - `if(x==0) ... else if (x==1) ... else if(x==2) ...`
 - 조금 아름답게 써 줄 수 없을까?
- 다중 조건문 switch
 - `switch(x) { case 0: ... case 1: ... case 2: ... }`
 - 훨씬 보기 좋다.

switch 문의 문법과 의미구조

문법

```
switch ( 계산식 )
{
  case 값1:
    명령문1; ...
    break;
  case 값2:
    명령문2; ...
    break;
  ...
  case 값n :
    명령문n; ...
    break;
  default:
    명령문n+1; ...
}
```

- 계산식은 정수나 문자
- 값은 정수나 문자 **상수**
 - 변수는 안된다
 - 예, 2 또는 'a'
- 두 경우가 같은 값을 가지면 안된다

의미구조

1. 계산식을 계산한다. 그 결과를 v 라 하자.
2. v 가 값 _{k} 와 동일하면, 따르면 명령문 _{k} 를 수행한다.
3. v 와 같은 값이 없으면 default의 명령문 _{$n+1$} 을 수행한다.

주의: switch 문은 매우 제한적

- case 에는 값만 쓸 수 있다.
 - `if(x==1) ... else if (x==2) ... else if (x==3)` 형태만 switch 문으로 대체 가능
- case y 와 같이 변수를 쓸 수는 없다.
 - `if(x==y)` 는 switch 문으로 대체될 수 없다.
- 비교연산자를 쓸 수는 없다.
 - `if(x<10)` 은 switch 문으로 대체될 수 없다.

주의: break;를 잊지말자

- switch 문의 각 경우에서 break;를 쓰지 않은 경우 매우 이상하게 작동
- 여러 경우에 같은 동작을 하고 싶을 때는 유용
 - ```
switch(x) {
 case 0:
 case 2:

 ... }
 ○
```
- 정확한 의미구조가 있지만, 가능한 사용하지 않을 것을 권장

# 흐름을 확 바꾸는 것들

- 예외 (exception)
  - 예외가 발생하면 즉각 흐름이 끊긴다. 즉, 프로그램이 종료하게 된다.
  - 발생한 예외에 대한 메시지가 출력된다.
- `System.exit(0);`
  - 바로 프로그램을 종료한다.
- `return;` 또는 `return <계산식>;`
  - 바로 메소드를 종료한다.

# 예외를 사용해 보자

```
import javax.swing.*;
public class MakeChange {
 ...
 public static void main(String[] args) {
 int dollars = getInt("달러?");
 int cents = getInt("센트?");
 if (dollars < 0 || cents < 0 || cents > 99)
 showError(dollars + " 또는 " +
 cents + "는 잘못된 입력입니다.");
 else { // 제대로 된 입력이니 계산을 시작해 볼까?
 ...
 }
 }
}
```

# 예외를 사용해 보자

```
import javax.swing.*;
public class MakeChange {
 ...
 public static void main(String[] args) {
 int dollars = getInt("달러?");
 int cents = getInt("센트?");
 if (dollars < 0 || cents < 0 || cents > 99) {
 String error = dollars + " 또는 " +
 cents + "는 잘못된 입력입니다.";
 throw new RuntimeException(error);
 }
 // 제대로 된 입력이니 계산을 시작해 볼까? (else가 필요없다)
 ...
 }
}
```

# return을 사용해 보자

```
public String twelveHourClock(int hour, int minute)
{
 String answer = ""; // 답을 여기다 누적할 것이다.

 if (hour < 0 || hour > 23 || minute < 0 || minute > 59)
 answer = "입력 오류, " + hour + ":" + minute; // 오류
 else {
 if (hour < 12)
 answer = answer + "오전 "; // 오전
 else
 answer = answer + "오후 "; // 오후
 if (hour >= 13)
 answer = answer + (hour - 12); // 시간이 13이상이면 12째 수로 붙인다.
 else
 answer = answer + hour; // 0~12시의 경우 그대로 붙인다.
 answer = answer + ":";
 if (minute < 10)
 answer = answer + "0"; // 분이 한자리수인 경우 0을 추가
 answer = answer + minute; // 분을 붙인다.
 }
 return answer;
}
```

# return을 사용해 보자

```

public String twelveHourClock(int hour, int minute)
{
 String answer = ""; // 답을 여기서 누적할 것이다.

 if (hour < 0 || hour > 23 || minute < 0 || minute > 59)
 return "입력 오류, " + hour + ":" + minute; // 오류

 if (hour < 12)
 answer = answer + "오전 "; // 오전
 else
 answer = answer + "오후 "; // 오후
 if (hour >= 13)
 answer = answer + (hour - 12); // 시간이 13이상이면 12째 수로 붙인다.
 else
 answer = answer + hour; // 0~12시의 경우 그대로 붙인다.
 answer = answer + ":";
 if (minute < 10)
 answer = answer + "0"; // 분이 한자리수인 경우 0을 추가
 answer = answer + minute; // 분을 붙인다.
}
return answer;
}

```

## 6.8 소프트웨어 설계

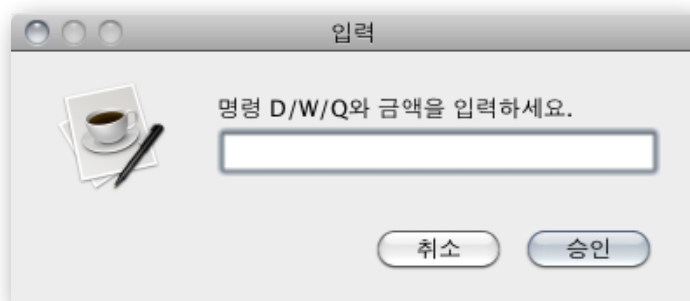
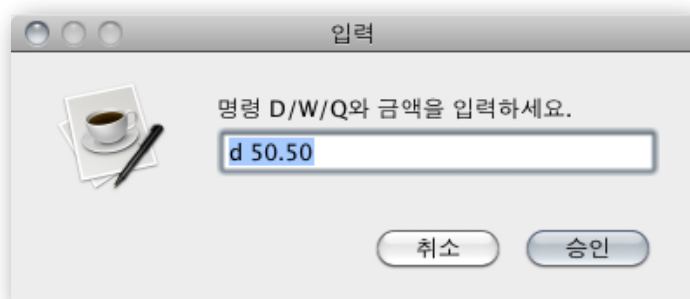
- 사용 행태 (use-case behaviour) 정의하기
  - 프로그램이 했으면 하는 일?
  - 어떻게 프로그램이 동작했으면?
- 사용 행태에 맞는 소프트웨어 구조 선택
- 각 구조 부품에 클래스 부여
- 각각의 클래스 작성
- 클래스를 구조에 녹여 결합하고, 시험하기



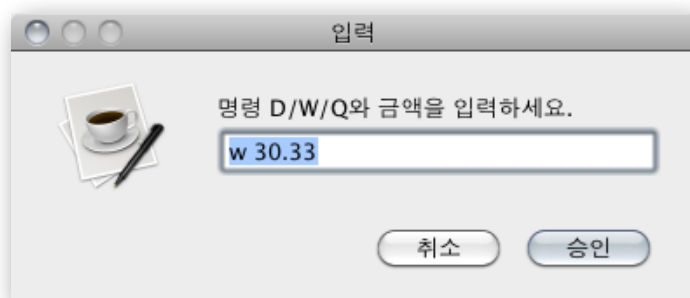
# 설계 예제: 은행 계좌 관리

- 고객의 계좌에 대한 출입을 처리하는 은행계좌 관리 프로그램을 작성하세요.
- 입력은 다음과 같은 명령들입니다.
  - D, 입금 (deposit), 달러와 센트
  - W, 출금 (withdraw), 달러와 센트
  - Q, 종료 (quit)
- 명령은 입력창을 통해 차례로 입력됩니다.

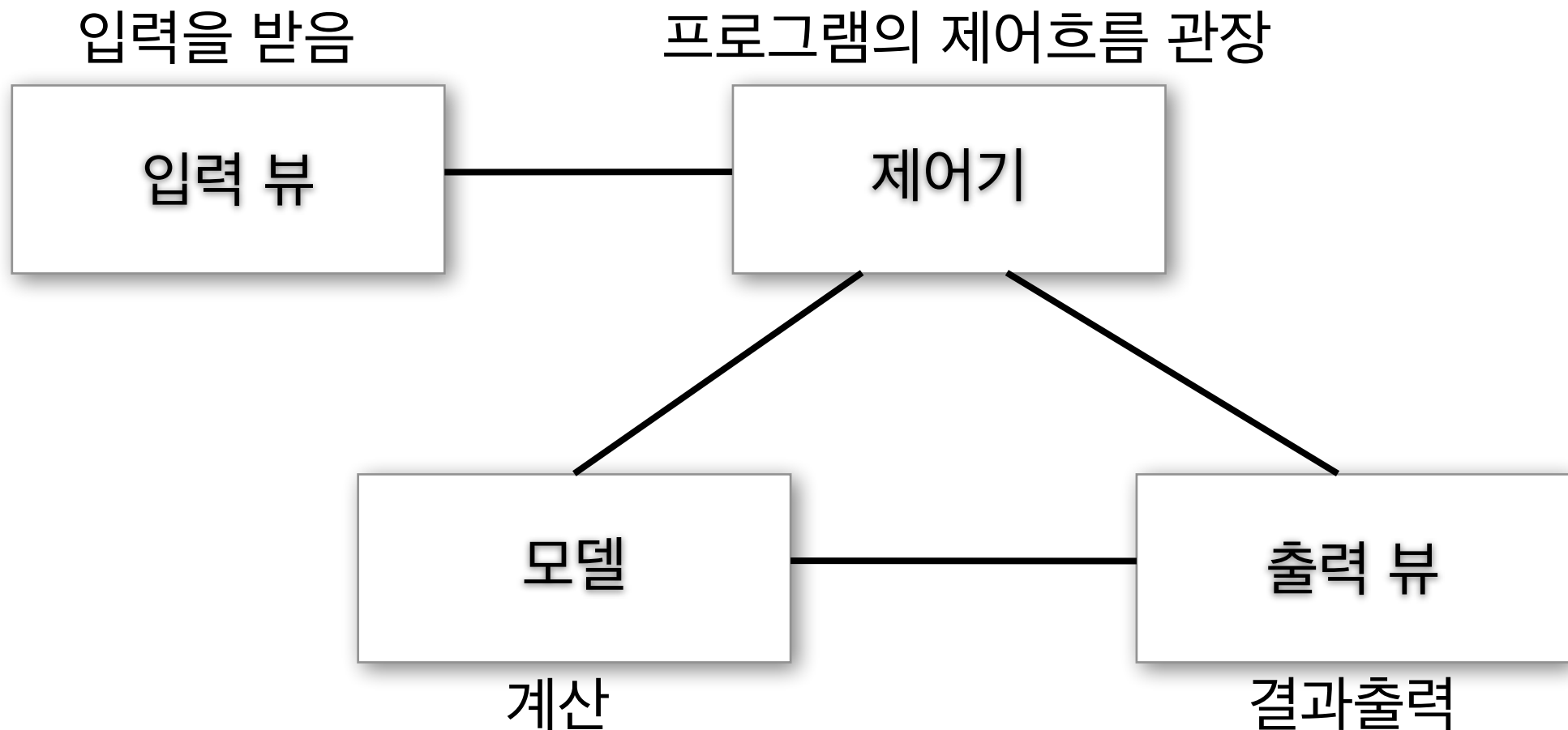
# 사용법

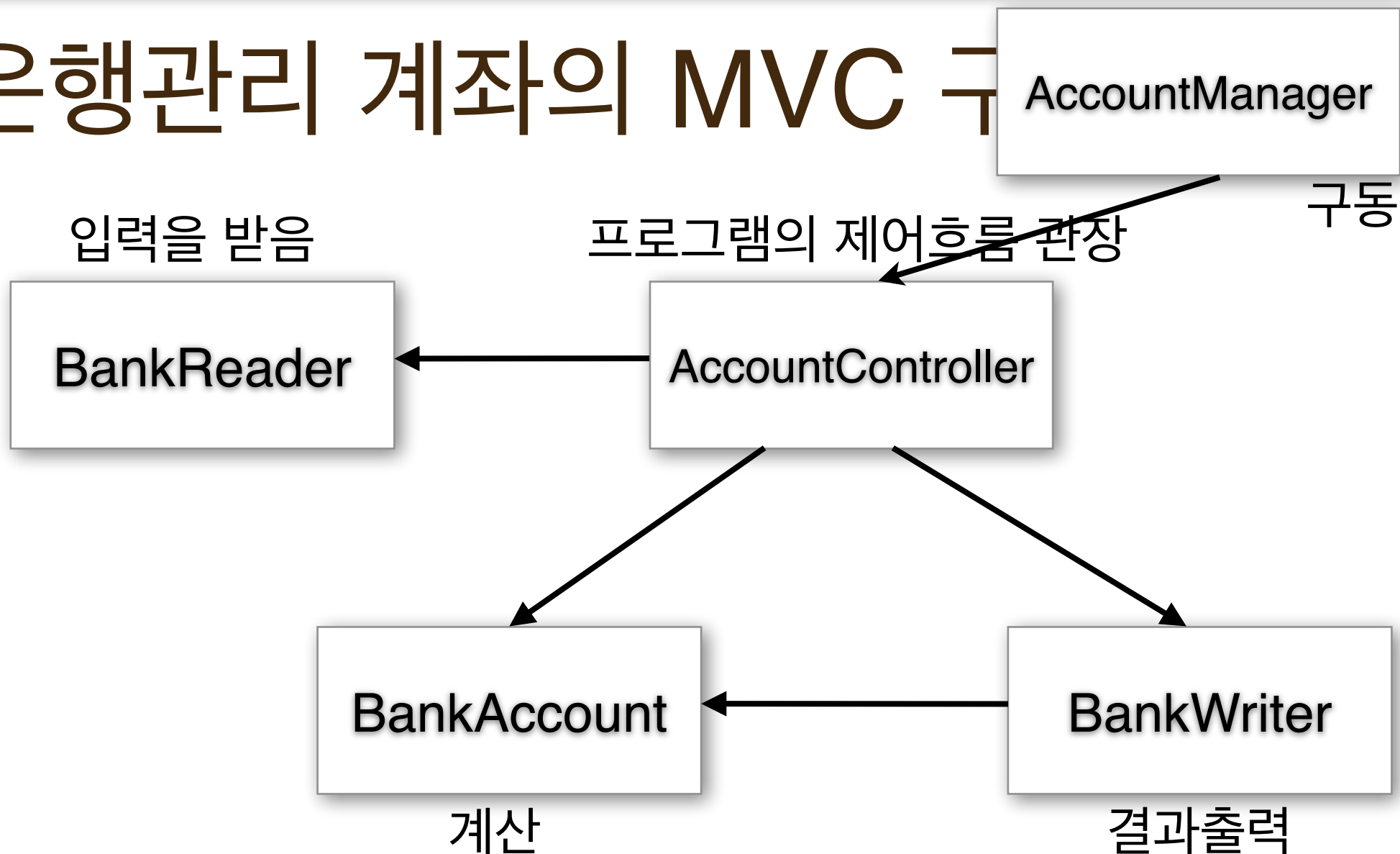
MyAccount: Deposit of \$ 50.50  
 Current balance = \$50.50  
 MyAccount: Withdrawal of \$ 30.33  
 Current balance = \$20.17



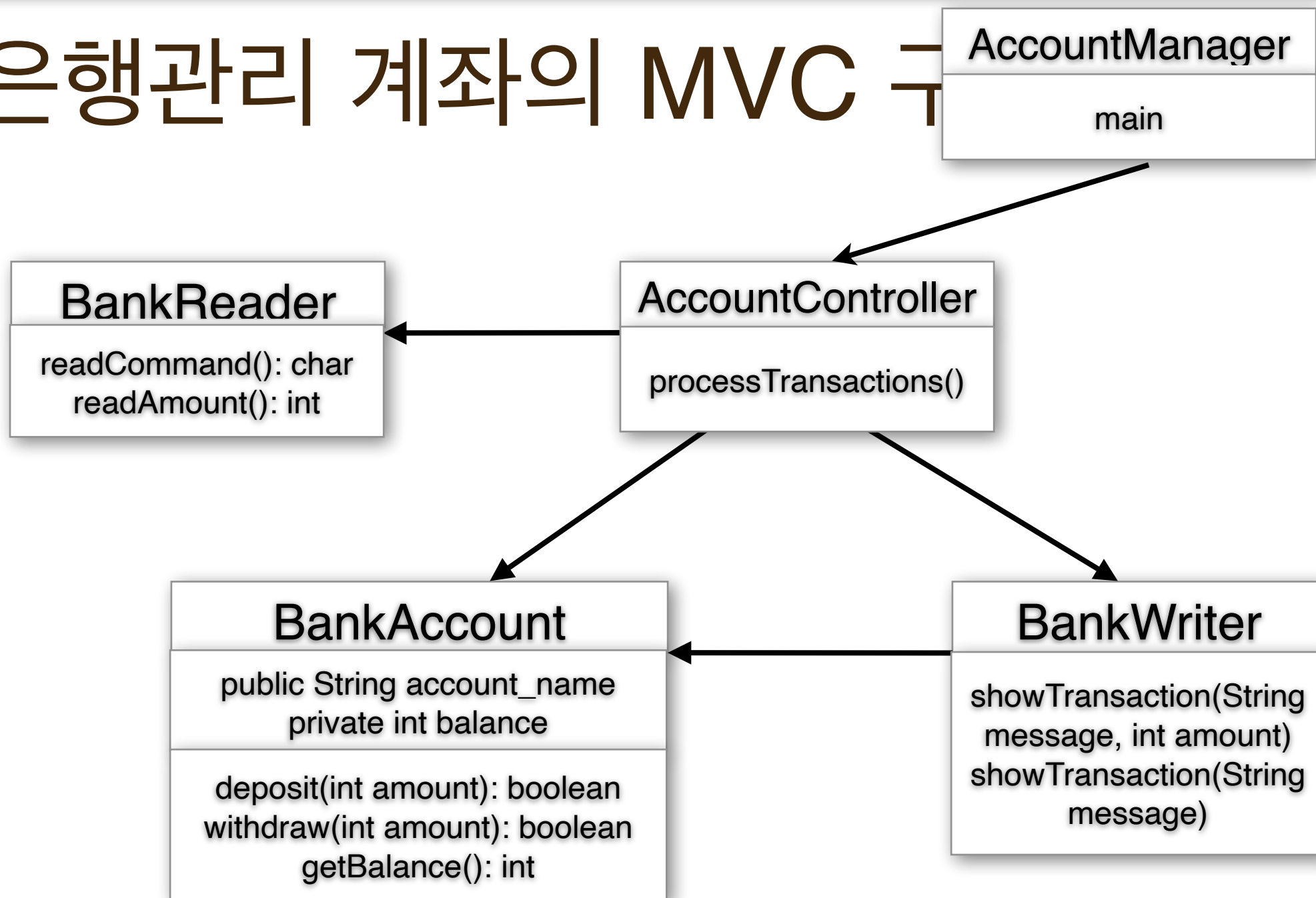
# Model-View-Controller 구조



## 은행관리 계좌의 MVC 구조



## 은행관리 계좌의 MVC 구조



# 명세: class BankAccount (모델)

| 생성자                                            |                                                                 |
|------------------------------------------------|-----------------------------------------------------------------|
| BankAccount (String name, int initial_balance) | 계좌 이름과 금액을 초기화                                                  |
| 속성                                             |                                                                 |
| public String account_name                     | 계좌의 이름                                                          |
| private int balance                            | 계좌의 잔액 (양수)                                                     |
| 메소드                                            |                                                                 |
| getBalance(): int                              | 잔액을 반환                                                          |
| deposit (int amount): boolean                  | amount가 양수이면 계좌 잔액에 더한다. 입금에 성공하면 true, 아니면 false를 반환           |
| withdraw (int amount): boolean                 | amount가 양수이면 계좌 잔액에서 뺀다. 출금에 성공하면 true, 계좌가 부족해서 실패하면 false를 반환 |

# BankAccount 필드, 생성자, getBalance

```
public class BankAccount {
 public String account_name;
 private int balance;

 public BankAccount(String name, int initial_amount) {
 account_name = name;
 if (initial_amount >= 0)
 balance = initial_amount;
 else
 balance = 0;
 }
 public int getBalance() {
 return balance;
 }
}
```

접근 메소드 (getter):  
필드 값을 읽기만 하는 메소드

# BankAccount.deposit

```
public boolean deposit(int amount) {
 boolean result = false;
 if (amount < 0)
 JOptionPane.showMessageDialog(null,
 "잘못된 입금액이라 무시합니다.");
 else {
 balance = balance + amount;
 result = true;
 }
 return result;
}
```

변경 메소드 (setter):  
필드 값을 변경하는 메소드



# BankAccount.withdraw

```
public boolean withdraw(int amount) {
 boolean result = false;
 if (amount < 0)
 JOptionPane.showMessageDialog(null,
 "잘못된 출금액이라 무시합니다.");
 else if (amount > balance)
 JOptionPane.showMessageDialog(null, "잔고가 부족합니다.");
 else {
 balance = balance - amount;
 result = true;
 }
 return result;
}}
```

# 명세: class BankReader (입력 뷰)

| 속성                                    |                                                                |
|---------------------------------------|----------------------------------------------------------------|
| private String<br>input_line          | 최근에 입력된 명령어                                                    |
| 메소드                                   |                                                                |
| readCommand (String<br>message): char | 입력창을 띄어 명령어를 입력받는다. 입력된<br>명령어의 첫글자를 반환한다.                     |
| readAmount(): int                     | 최근에 입력된 명령어에서 첫글자를 제외한 문<br>자열 xx.yy를 달러로 해석해서 센트로 변환하<br>여 반환 |

# BankReader.readCommand

```
import javax.swing.*;

public class BankReader {

 private String input_line = "";

 public char readCommand(String message) {
 input_line = JOptionPane.showInputDialog(message).
 toUpperCase();
 return input_line.charAt(0);
 }
}
```

# BankReader.readAmount

```
public int readAmount()
{
 int answer = 0;
 String s = input_line.substring(1, input_line.length());
 if(s.length() > 0) {
 double dollars_cents = new Double(s).doubleValue();
 answer = (int)(dollars_cents*100);
 }
 else
 JOptionPane.showMessageDialog(null,
 "금액을 입력하지 않아 0으로 처리합니다.");
 return answer;
}
```

# 명세: class BankWriter (출력 뷰)

| 생성자                                          |                                              |
|----------------------------------------------|----------------------------------------------|
| BankWriter (BankAccount b)                   | 은행계좌 b에 대한 출력 뷰 생성                           |
| 속성                                           |                                              |
| private BankAccount bank                     | 이 출력 뷰에 연관된 은행 계좌                            |
| private String last_transaction              | 최근 거래의 메시지                                   |
| 메소드                                          |                                              |
| showTransaction (String message, int amount) | 메시지(message)와 금액(amount)에 해당하는 최근 거래를 화면에 출력 |
| showTransaction (String message)             | 메시지(message)에 해당하는 최근 거래를 화면에 출력             |

# BankWriter 필드와 생성자

```
import java.text.*;
public class BankWriter {
 private BankAccount bank;
 private String last_transaction = "";
 public BankWriter(BankAccount b) { bank = b; }
 private String unconvert(int i) {
 return new DecimalFormat("0.00").format(i/100.0);
 }
 public void showTransaction(String message, int amount)
 { last_transaction = message + " " + unconvert(amount);
 System.out.println(bank.account_name + ": " + last_transaction);
 System.out.println("Current balance = $" +
 unconvert(bank.getBalance()));
 }
 public void showTransaction(String message) {
 last_transaction = message;
 System.out.println(bank.account_name + ": " + last_transaction);
 }
}
```

# 명세: class AccountController (제어기)

| 생성자                                                              |                                               |
|------------------------------------------------------------------|-----------------------------------------------|
| AccountController<br>(BankReader r, BankWriter w, BankAccount b) | 해당 인수들로 객체를 초기화                               |
| 속성                                                               |                                               |
| BankReader reader                                                | 입력 뷰                                          |
| BankWriter writer                                                | 출력 뷰                                          |
| BankAccount account                                              | 모델: 은행 계좌                                     |
| 메소드                                                              |                                               |
| processTransactions()                                            | 하나의 거래를 수행한다. 종료 명령어가 입력되지 않는 경우 다시 거래를 수행한다. |

# AccountController

```
public class AccountController {
 private BankReader reader;
 private BankWriter writer;
 private BankAccount account;

 public AccountController(BankReader r, BankWriter w,
 BankAccount a) {
 reader = r;
 account = a;
 writer = w;
 }
}
```



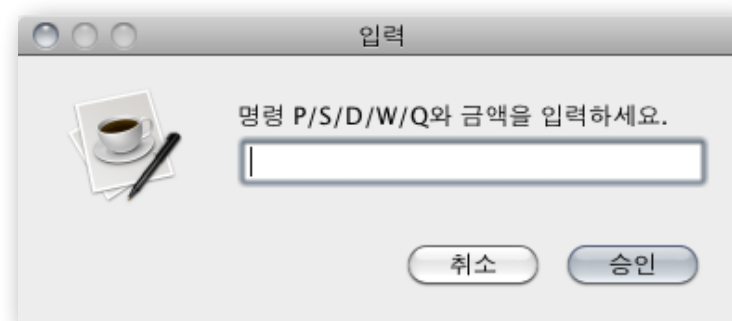
# AccountController.processTransactions

```
public void processTransactions()
{
 char command = reader.readCommand("명령 D/W/Q와 금액을 입력하세요.");
 switch (command) {
 case 'Q':
 return;
 case 'D':
 {
 int amount = reader.readAmount();
 if (account.deposit(amount)) writer.showTransaction("입금 $", amount);
 else writer.showTransaction("입금 오류 ", amount);
 break;
 }
 case 'W':
 {
 int amount = reader.readAmount();
 if (account.withdraw(amount)) writer.showTransaction("출금 $", amount);
 else writer.showTransaction("출금 오류 ", amount);
 break;
 }
 default:
 writer.showTransaction("잘못된 명령 " + command);
 }
 this.processTransactions();
}
```

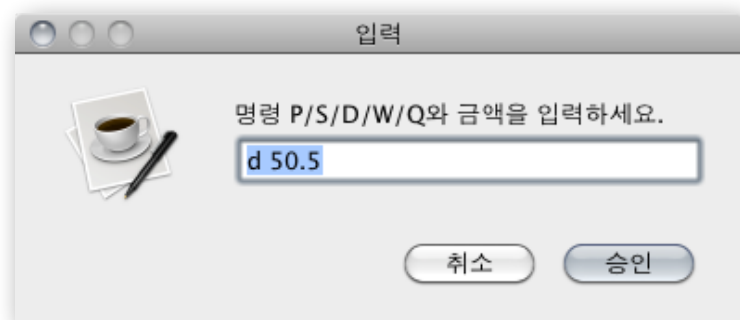
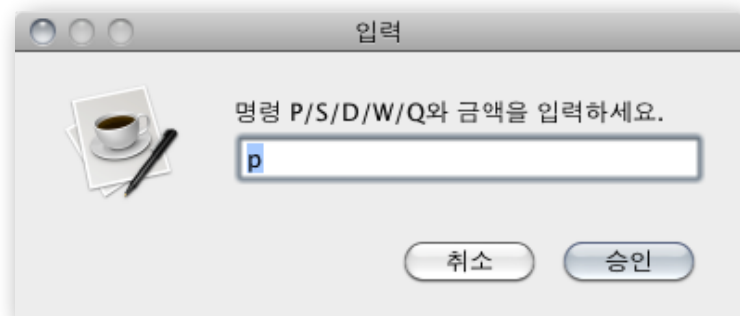
# 구동 class AccountManger

```
public class AccountManager {
 public static void main(String[] args)
 { // create the application's objects:
 BankReader reader = new BankReader();
 BankAccount account = new BankAccount("myaccount",0);
 BankWriter writer = new BankWriter(account);
 AccountController controller =
 new AccountController(reader, writer, account);
 // start the controller:
 controller.processTransactions();
 }
}
```

# 확장: 두 계좌를 관리해 보자

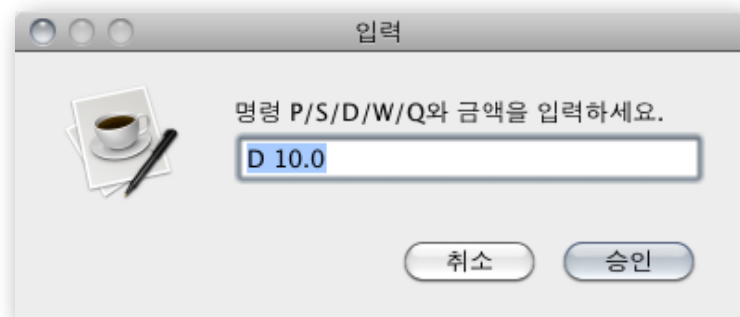
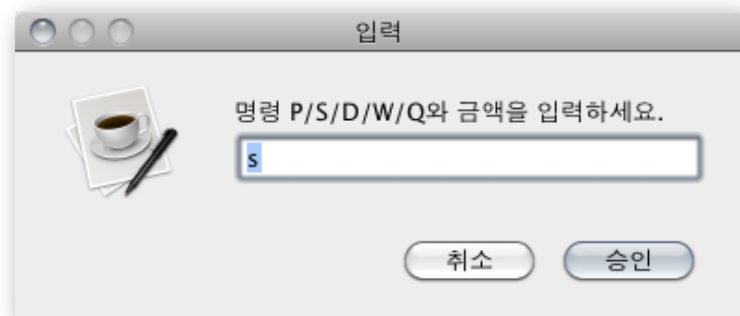


# 사용 행태



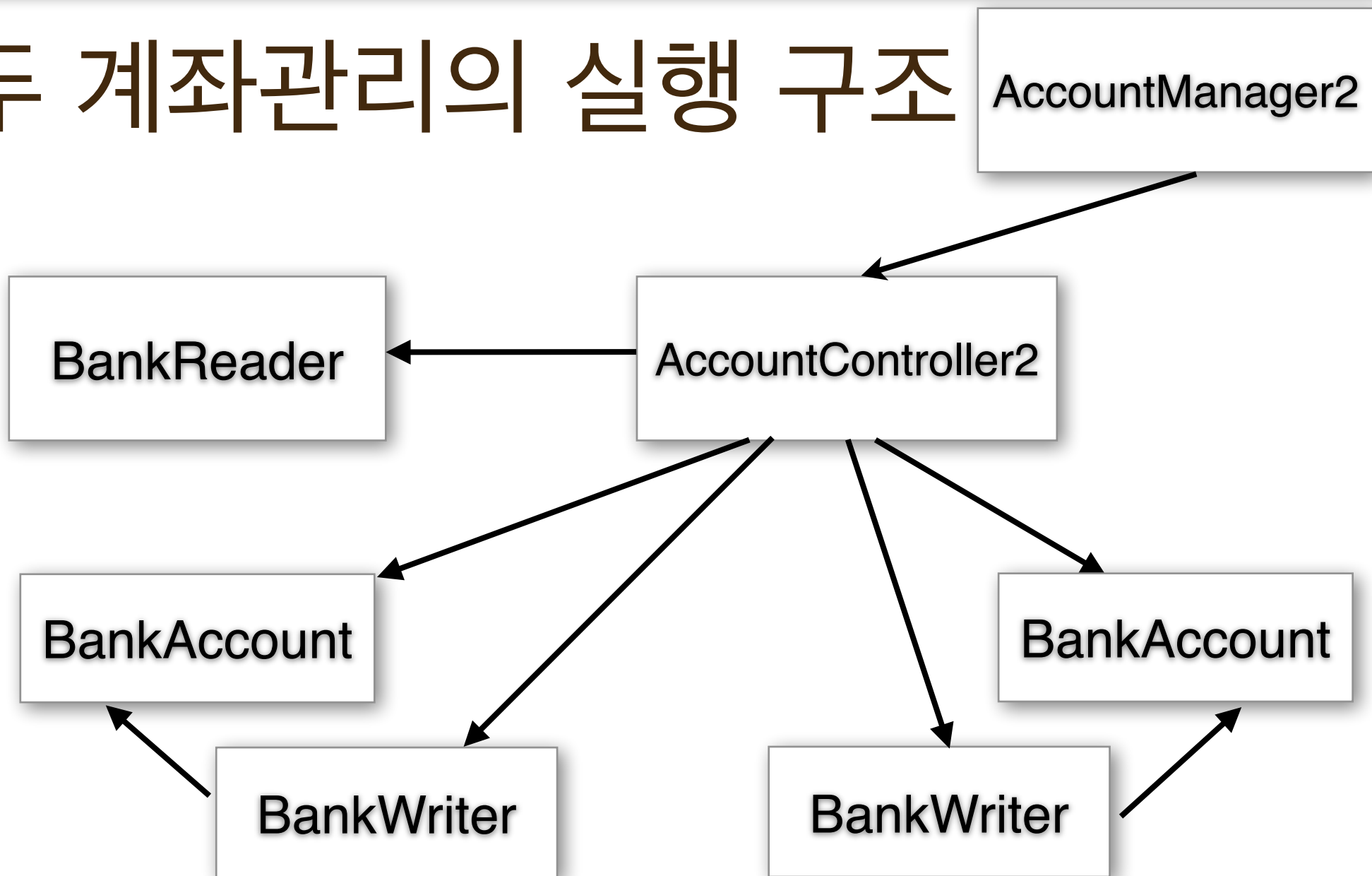
Secondary account: Inactive  
Primary account: Active  
Primary account: Deposit of \$ 50.50  
Current balance = \$50.50

# 사용 행태

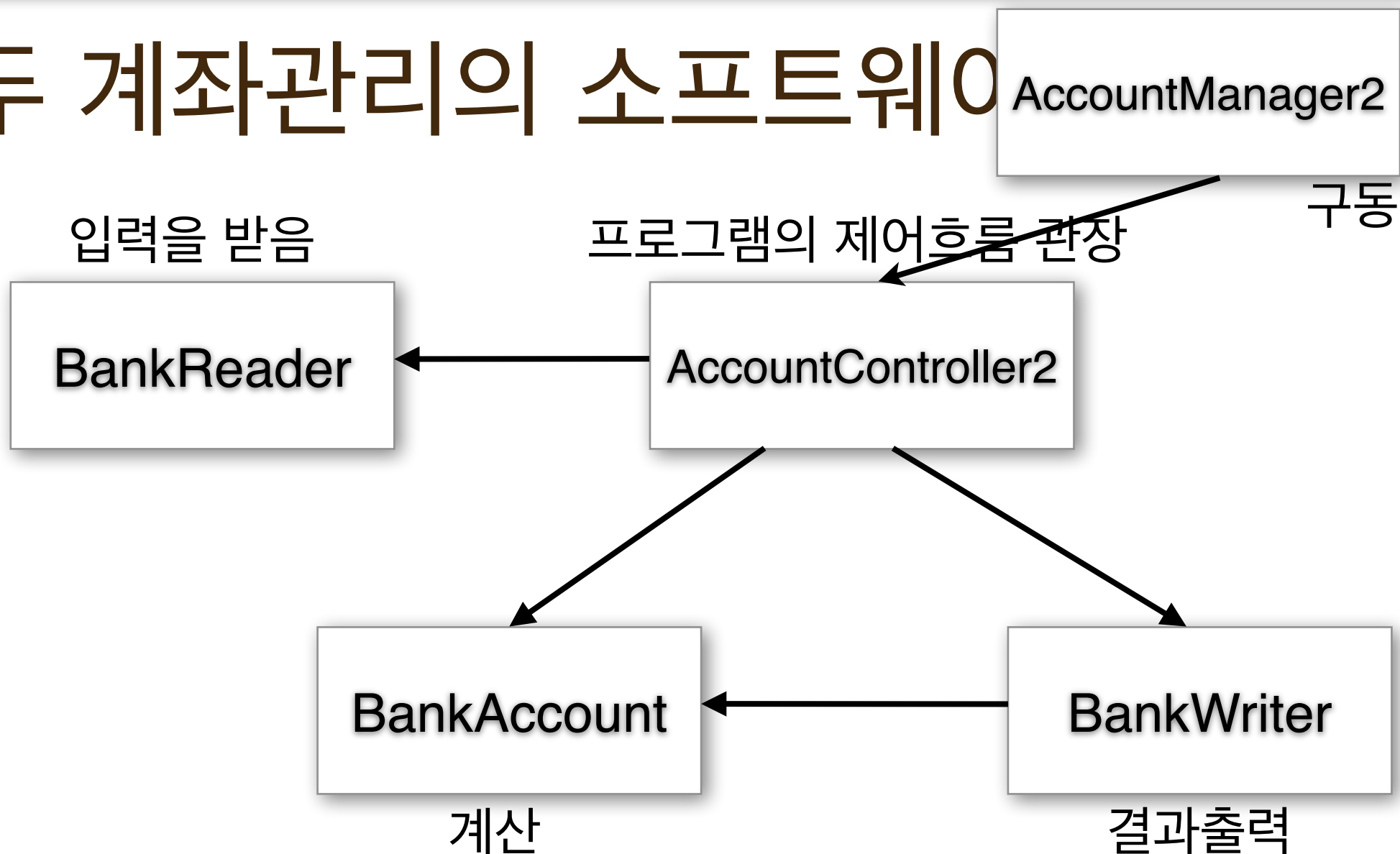


Primary account: Inactive  
Secondary account: Active  
Secondary account: Deposit of \$ 10.00  
Current balance = \$10.00

# 두 계좌관리의 실행 구조



# 두 계좌관리의 소프트웨어



# AccountController2

```
public class AccountController2 {
 private BankReader reader; // input view
 private BankAccount primary_account, secondary_account, account;
 private BankWriter primary_writer, secondary_writer, writer;

 public AccountController2 (BankReader r, BankAccount a1,
 BankWriter w1, BankAccount a2, BankWriter w2) {
 reader = r;
 primary_account = a1;
 primary_writer = w1;
 secondary_account = a2;
 secondary_writer = w2;
 account = primary_account;
 writer = primary_writer;
 }
}
```



# AccountController2.resetAccount

```
public void resetAccount (BankAccount new_account,
 BankWriter new_writer) {
 writer.showTransaction("비활성");
 account = new_account;
 writer = new_writer;
 writer.showTransaction("활성");
}
```

# AccountController2.processTransactions

```
public void processTransactions()
{
 char command =
 reader.readCommand("명령 P/S/D/W/Q와 금액을 입력하세요.");
 switch (command) {
 case 'P':
 resetAccount(primary_account, primary_writer);
 break;
 case 'S':
 resetAccount(secondary_account, secondary_writer);
 break;
 ...
 }
 this.processTransactions();
}
```

# 구동 class AccountManager2

```
public class AccountManager2 {
 public static void main(String[] args)
 { // create the application's objects:
 BankReader reader = new BankReader();
 BankAccount primary_account = new BankAccount(0);
 BankWriter primary_writer =
 new BankWriter("계좌#1", primary_account);
 BankAccount secondary_account = new BankAccount(0);
 BankWriter secondary_writer =
 new BankWriter("계좌#2", secondary_account);
 AccountController controller =
 new AccountController(reader, primary_account, primary_writer,
 secondary_account, secondary_writer);
 controller.processTransactions();
 }
}
```

# 요약

- 조건문
  - `if(질문?) { 명령문1 } else { 명령문2 }`
- 흐름을 확 바꾸는 구조
  - 예외 (exception), 반환 (return), `System.exit`
- 모델-뷰-제어기 (MVC) 구조
  - 클래스의 재사용 원할
  - 문제를 관리 가능한 작은 부품으로 분해
  - 각 부품은 다른 부품과 연관이 적어야 함