실습최종제출 6/8 자정까지

11. 객체지향 언어: 객체와 클래스

숙명여대 창병모

11.1 객체지향 언어

객체지향 언어의 역사

- Simula-67
  - 실세계에 있는 객체들을 표현하고 이들 사이의 상호작용을
  - 시뮬레이션하기 위한 언어로 개발됨.
- Smalltalk
  - 순수한 객체지향 언어로 설계
  - 최초로 GUI를 제공하는 언어
- C++
  - 초기의 C++는 C에 클래스 개념만 도입하여
  - 상속, 가상 함수, 추상 클래스, 예외 처리 등과 같은 다양한 기능 추가
- Java
  - 단순성과 플랫폼 독립성
  - 객체지향 패러다임에 충실하게 고안되어 C++보다 오용의 소지가 적다.
  - 현재 웹 애플리케이션과 모바일 앱 개발 등의 분야에서 가장 많이 사용됨.

객체지향 언어의 역사

Apple
- Objective-C/Swift
  - C++와 마찬가지로 C와 객체지향 언어를 혼합한 언어
  - C++보다는 Smalltalk에 좀 더 가깝게 정의된 언어
  - Swift로 발전함

MS
- C# 자바와 상당히 유사함
  - 마이크로소프트에서 개발한 객체지향 프로그래밍 언어
  - 닷넷 프레임워크의 한 부분으로 만듬.
  - C#은 그 문법적인 특성이 Java와 상당히 유사

- Python 언어
  - 플랫폼에 독립적이며 동적 타이핑을 지원하는 대화형 인터프리터 언어
  - 객체지향 개념을 포함한 다중 패러다임 언어이다.

객체 지향 : 동기

프로그램이란건 뭐지?
→여러관점이 존재

- **질문 1** : 객체-지향 프로그래밍 언어는 왜 시작되었을까?

  프로그램은 실세계 객체들을 표현하고 시뮬레이션 하는 것

  ↳ 그렇 하네!

- 객체지향 프로그래밍(Object-Oriented Programming, OOP)

  오브제(사물)

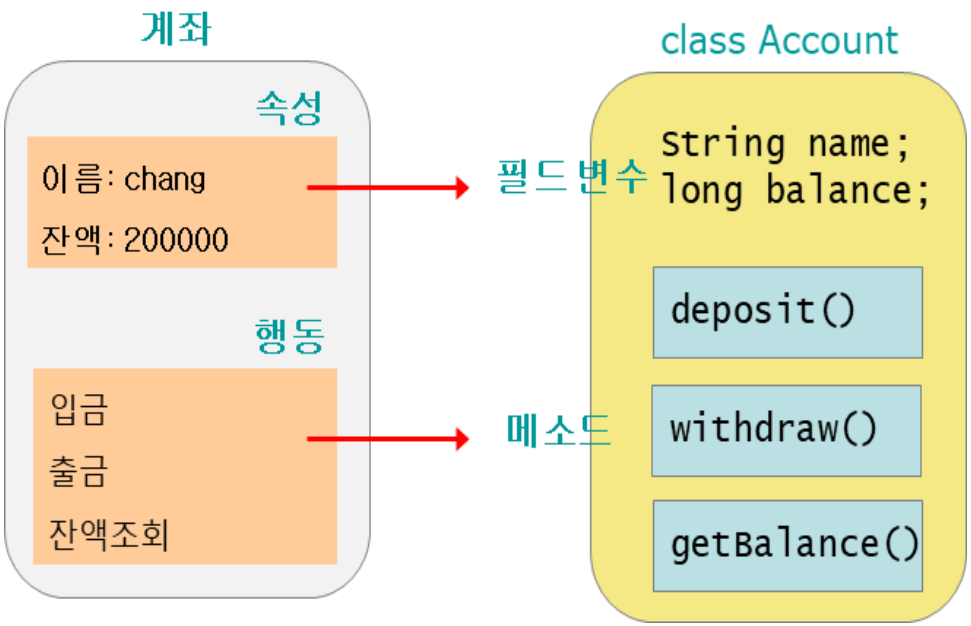
  프로그램을 여러 개의 독립된 단위, 즉 "객체"들의 모음으로 파악하고 각각의 객체는 메시지를 주고받고, 데이터를 처리할 수 있는 개체이다.

객체

- **질문 2** : 실세계의 객체를 프로그램 상에 어떻게 표현할 수 있을까?

- 객체
  - 속성(attribute) - 객체의 데이터 혹은 상태를 나타내는 속성
  - 행동(behavior) - 객체에 취할 수 있는 연산 혹은 동작

- 예: 은행계좌
  - 이름
  - 잔액
  - 입금
  - 출금
  - 잔액조회



객체와 클래스

- 객체와 클래스
  - 클래스는 객체의 정의 혹은 타입(a type of objects)
  - 객체는 어떤 클래스 타입으로 선언
  - 객체는 어떤 클래스의 실체(instance)이다.

- 실체화(Instantiation)
  - 클래스로부터 객체를 생성하는 것
  - new in Java, C++

- 프로그램

  프로그램 = 클래스들의 집합

  프로그램 실행 = 서로 상호작용하는 객체들의 집합

  (a set of objects invoking each other methods)

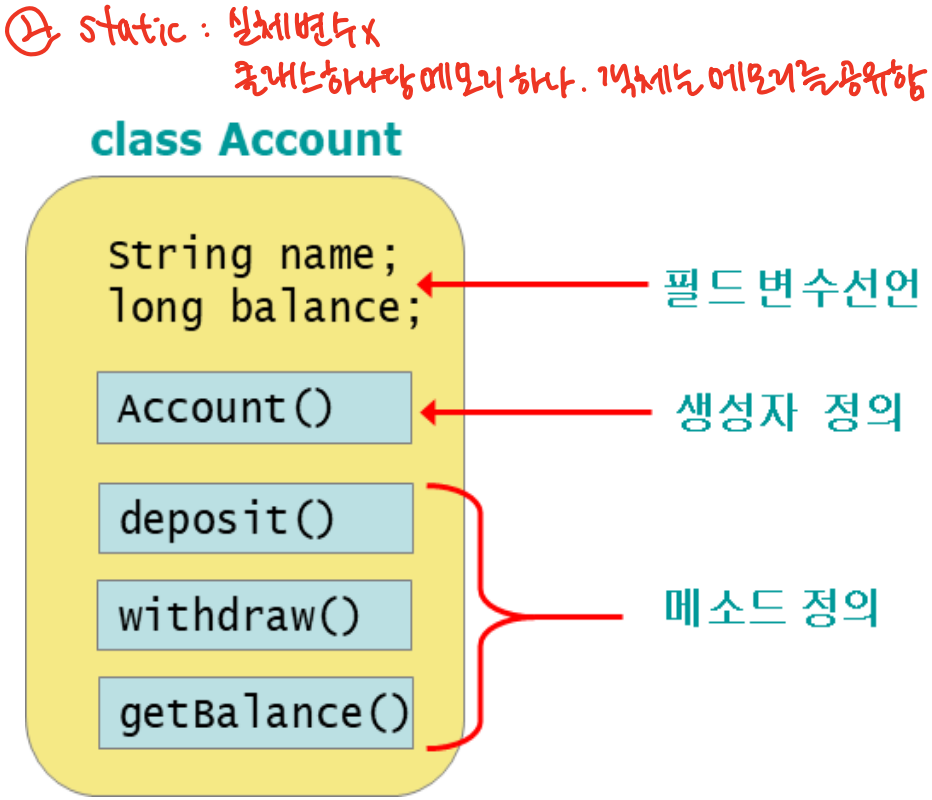
11.2 Java 클래스

클래스 정의

- 클래스 정의 구문

```
class 클래스이름 {
        필드 변수 선언
        생성자 선언
        메소드 선언
}
```

- 클래스 멤버
  - 변수, 생성자, 메소드



실체 변수(instance variable)
- 클래스 내에 정의된 변수
- 각 객체마다 기억공간이 할당됨

[예제 1]

```
class Account
{
    private String name;
    private long balance;
    Account(String name) {
        this.name = name;
        balance = 0;
    }
    public long getBalance() {
        return balance;
    }
    public long deposit(long amount) {
        balance += amount;
        return balance;
    }
    public long withdraw(long amount) {
        if (amount <= balance)
            balance -= amount;
        else System.err.println("잔액 부족");
        return balance;
    }
}
```

생성자

- 생성자
  - 클래스와 이름이 같은 특별한 메소드
  - 생성된 객체를 set up(보통 실체 변수 초기화)하는데 사용된다.
  - 반환 값이 없으며 반환 타입도 없다.
  - 중복 정의하는 것도 가능함.

- 예: Account 생성자
  - 이름을 받아서 초기화한다.
  - 이름뿐만 아니라 초기화 잔액을 받아 초기화하도록 중복정의 가능

```
public Account(String name, long amount) {
    this.name = name;
    balance = amount;
}
```

this. 생략됨

객체 생성

- 객체 참조 변수(object reference variable) 선언

  클래스이름 변수;

  객체를 가리키기 위한 변수로 객체가 생성된 것은 아님!

  Account acc1;

- 객체 생성

  변수 = new 클래스이름(인자);

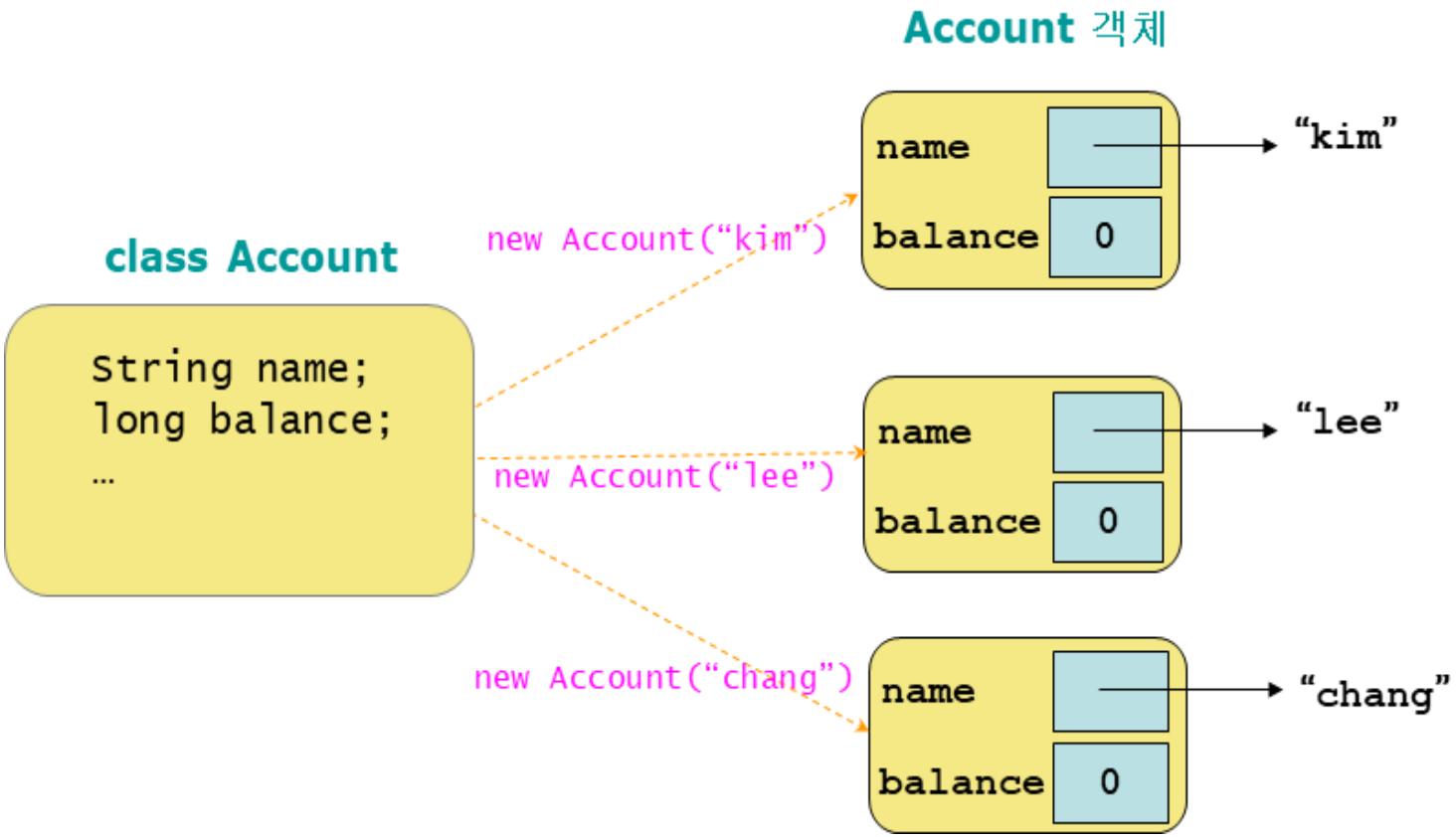
  new 연산자는 클래스로부터 객체를 생성한다.

  Account acc1 = new Account("kim");

객체 생성

- 클래스로부터 객체 생성(instantiation)
  - 각 객체를 위한 기억공간이 할당됨
  - 어디에?
- 객체
  - 클래스의 실체 (an instance of a class)
- 실체 변수(instance variable)
  - 클래스 내에 정의된 변수로 각 객체마다 기억공간이 할당됨
- 한 클래스의 모든 객체들은 메소드를 공유

클래스 및 객체


Account 객체
class Account
String name;
long balance;
…
new Account("kim")
new Account("lee")
new Account("chang")
name
balance
0
"kim"
"lee"
"chang"

클래스 사용

```
/* Milage 클래스의 사용을 테스트한다. */
public class AccountTest
{
    public static void main(String[] args) {
        Account acc1 = new Account( " kim");
        System.out.println(acc1.getBalance());
        System.out.println(acc1.deposit(200000));
        Account acc2 = new Account("lee");
        System.out.println(acc2.getBalance());
        Account acc3 = new Account("chang");
        System.out.println(acc3.deposit(500000));
    }
}
```

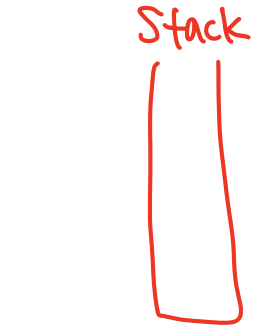
실행결과

```
0
200000
0
500000
```

11.3 캡슐화

추상 자료형

- 추상 자료형(abstract data type)
  - 데이터(필드변수)와 관련된 연산(프러시저)들을 한데 묶어
  - 캡슐화하여 정의한 자료형이다.
  - Ada의 패키지, Modula-2 모듈, 클래스
- 클래스
  - 추상 자료형의 발전된 형태로 볼 수 있음.



구체적으로 어떻게 구현되었는지는
알필요도 없고 알 수도 없음
(관리를 위해 액세스 제한)

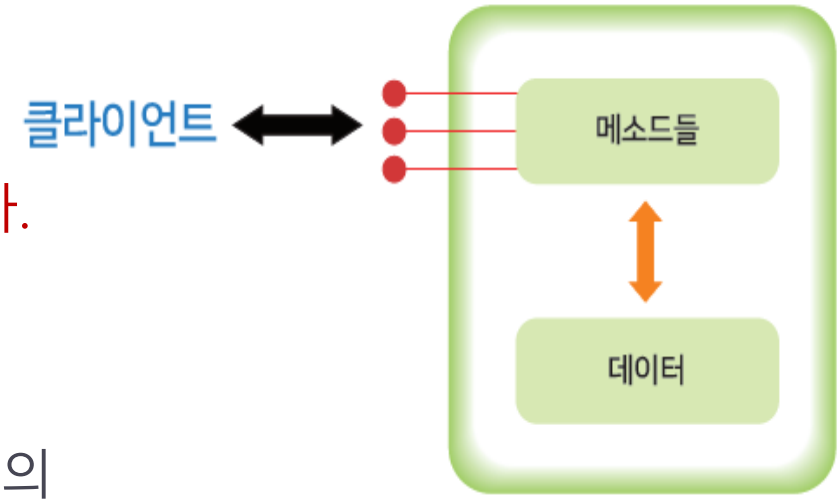
캡슐화(Encapsulation)

- 캡슐화
  - 일반적으로 연관 있는 필드변수와 메소드를 클래스로 묶고
  - 내부 구현 내용을 외부에 감추는 것을 말한다.

- 캡슐화가 왜 필요한가?
  - 객체의 내부 구현 내용을 캡슐화하여
  - 보호하는 이유는 외부의 잘못된 사용으로
  - 객체가 손상되지 않도록 하는 데 있다.

객체와 캡슐화

- 객체의 외부적 관점
  - 객체는 하나의 캡슐로
  - 서비스(즉 객체에 대한 창구)를 제공한다.



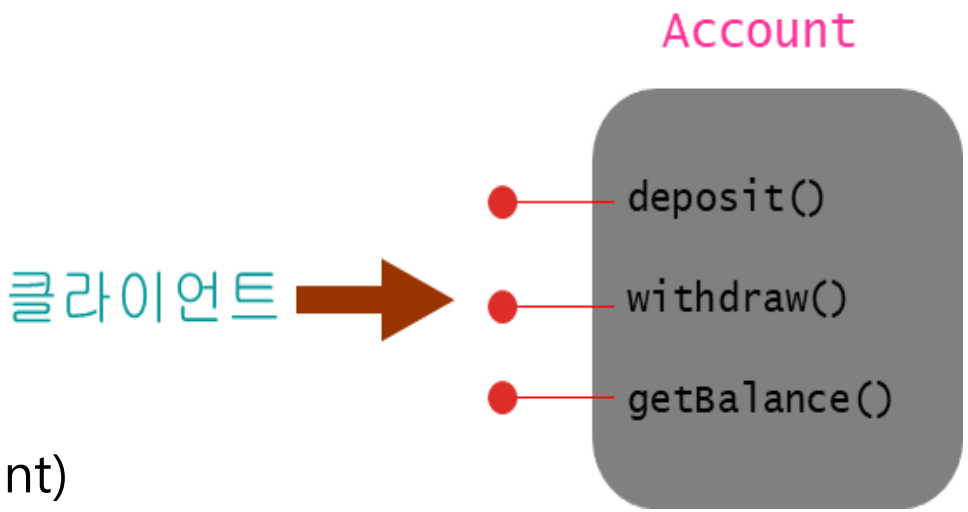
- 객체의 사용자
  - 실제 데이터(필드변수)를 관리하는 객체의
  - 서비스 메소드를 호출하여 객체와 소통한다.

- 객체는 캡슐처럼 자기 관리되어야 한다.
  - 객체의 상태는 메소드에 의해서 변경되게 하고
  - 외부에서 직접 접근하는 것을 어렵게 해야 한다.
  - 캡슐화된 객체는 블랙 박스로 생각할 수 있다.

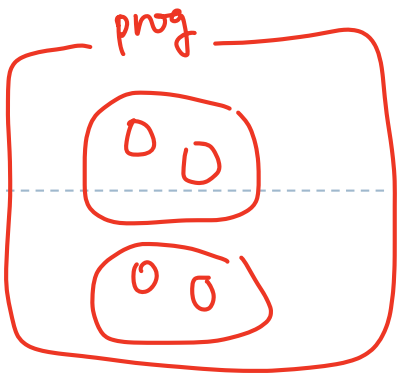
클래스 정의: 예

```
private String name;
private long balance;

public long deposit(long amount)
public long withdraw(long amount)
public long getBalance( )
```



접근 지정자: Java



접근 지정자를 사용하여 캡슐의 접근을 제어한다.

- public

  public으로 선언된 멤버는 클래스 내외 어느 곳에서나 접근 가능

- private

  private으로 선언된 멤버는 클래스 내부에서만 접근 가능

- package

  아무 접근 지정자도 기술하지 않을 때 package 혹은 default 가시성

  같은 패키지 내에서는 자유롭게 접근, 패키지 외부에서는 접근할 수 없음

- protected

  protected 멤버는 클래스 내부와 이 클래스의 자손 클래스에서 접근 가능

  같은 패키지 내에 있는 클래스들은 모두 접근 가능하다.

접근 지정자

- 일반적 규칙
  - 객체의 필드변수는 공용으로 선언하면 안 된다.
- 공용 메소드
  - 객체의 서비스(창구)를 제공하는 메소드로
  - 서비스 메소드라고도 한다.
- 지원 메소드
  - 서비스 메소드를 지원하는 메소드로
  - 공용이 아니다.

11.4 정적 변수 및 정적 메소드

실체 변수

- 실체 변수
  - 각 객체마다 기억공간이 할당됨

- 예

```
/* Account 객체의 개수를 위해 실체 변수 counter 사용 */
class Account
{
    int counter = 0;
    public Account() {
        counter++;
    }
      ...
}
```

정적 변수와 정적 메소드

- 정적 변수
  - static으로 선언된 변수로 클래스 변수(class variable)라고도 함.
  - 객체가 아니라 클래스 내에 변수를 위한 기억공간이 생성됨
  - 한 클래스의 모든 실체(객체)들이 공유하여 사용

- 정적 메소드
  - static으로 선언되 메소드로 **클래스 메소드**라고도 함.
  - 객체를 생성하지 않고도 클래스 이름을 이용하여 호출 가능
  - 이 때문에 실체 변수는 사용할 수 없고 정적 변수는 사용가능

정적 변수와 정적 메소드

```
/* Account 객체의 개수를 위해 정적 변수 counter 사용 */
class Account
{
    static int counter = 0;
    public Account() {
        counter++;
    }
    public static int getCounter() {
       return counter;
    }
    public static void main(String[] args) {
        Account acc1 = new Account();
        Account acc2 = new Account();
        System.out.println(Account.counter);
        System.out.println(Account.getCounter());
    }
}
```

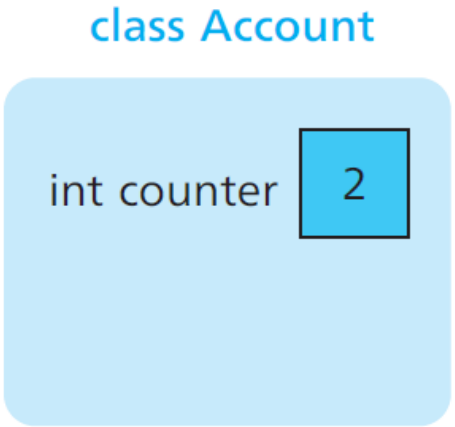


그림 11.6 정적 변수

11.5 제네릭

제네릭 프로그래밍

- 제네릭 프로그래밍(generic programming)
  - 매개변수 다형성(parametric polymorphism)
  - 타입 매개변수를 이용하여 여러 타입에 적용 가능한 포괄적 코드
  - Java 1.5부터 추가되었으며 Java Collection에서 많이 사용됨.
- 제네릭 클래스(generic class)
  - 타입 매개변수(type parameter)를 받는 클래스
  - ArrayList처럼 클래스에서 사용할 타입을 매개변수로 받을 수 있음
- 제네릭 메소드(generic method)
  - 타입 매개변수를 받는 메소드

[예제 4] 제네릭을 사용하지 않은 경우

```
class Drawable { void paint(){} }
class Circle extends Drawable { }
class Rectangle extends Drawable { }

public class Collection1 {
    public static void main(String args[]) {
        ArrayList myList = new ArrayList();
        myList.add(new Circle());
        myList.add(new Rectangle());
        myList.add(new Integer(1));
        for (Iterator i = myList.iterator(); i.hasNext(); ) {
            Drawable obj = (Drawable) i.next();
            obj.paint();
        }
    }
}
```

[예제 5] 제네릭 클래스 사용

```
public class Collection2 {
    public static void main(String args[]) {
        ArrayList<Drawable> myList = new ArrayList<Drawable>();
        myList.add(new Circle());
        myList.add(new Rectangle());
        // myList.add(new Integer(1));  오류

        for (Drawable obj : myList)
            obj.paint();
    }
}
```

- 리스트에 저장하는 내용물은 다르게 사용 가능
  - ArrayList<Drawable> list1 = new ArrayList<Drawable>();
  - ArrayList<String> list2 = new ArrayList<>(); // 타입 생략

제네릭 클래스 정의

- 타입 매개변수를 이용하여 여러 타입에 적용될 수 있는 포괄적인 클래스를 정의한다.

```
public class 클래스명<T> { ... }
```

```
public class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

new Box<String>( );

```
public class Box<String> {
    private String t;
    public void set(String t) {
        this.t = t;
    }
    public String get() {
        return t;
    }
}
```

제네릭 클래스 사용

- 제네릭 클래스 사용할 때 구체적 타입을 지정하여 여러 타입의 Box 객체 생성

```
public static void main(String[] args) {
        Box<String> box1 = new Box<String>();
        Box<Integer> box2 = new Box<Integer>();
        box1.set("test");
        box2.set(100);
        System.out.println(box1.get() + box2.get());
}
```

- 실행 결과

  test 100

제네릭 메소드

- 타입 매개변수를 이용하여 여러 타입에 사용될 수 있는 포괄적 메소드를 정의

```
public <T> 리턴타입 메소드명(매개변수,...) { … }
```

- 예

```
public class TestBox {
    public static <T> Box<T> boxing(T t) {
        Box<T> box = new Box<T>();
        box.set(t);
        return box;
    }
}
```

제네릭 메소드 호출

- 만약 이 메소드를 다음과 같이 호출하면

```
Box<String> box = TestBox.<String>boxing("string");
```

- 이 메소드는 다음과 같이 T가 String으로 대치된 메소드를 호출함.

```
public static Box<String> boxing(String t) {
    Box<String> box = new Box<String>();
    box.set(t);
    return box;
}
```

제네릭 메소드 호출

```
public class TestBox {
    public static <T> Box<T> boxing(T t) {
        Box<T> box = new Box<T>();
        box.set(t);
        return box;
    }
    public static void main(String[] args) {
        Box<String> box1 = TestBox.<String>boxing("test ");
        Box<Integer> box2 = TestBox.<Integer>boxing(100);
        System.out.println(box1.get() + box2.get());
    }
}
```

11.6 Python 클래스

클래스 정의

- 클래스 정의

```
class 클래스명:
    def __init__(self, 매개변수):
        ...
    def 메소드명(self, 매개변수):
        ...
```

- __init__ 메소드
  - 객체 초기화에 이용한다.
- 객체의 실체 변수(속성)
  - 별도로 선언하지 않고 사용한다.
- self
  - Java의 this처럼 자신 객체를 나타내며
  - "self.변수이름"은 객체의 실체 변수를 나타낸다.

Python 클래스

```
class Account:
    def __init__(self, name):
        self.name = name
        self.balance = 0
    def getBalance(self):
        return self.balance
    def deposit(self, amount):
        self.balance += amount
        return self.balance
    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("잔액 부족")
            return self.balance
```

실체 변수를 선언하지 않고 초기화!

객체의 생성 및 사용

```
>>> my = Account(‘kim’)
>>> my.deposit(1000)
1000
>>> my.getBalnace()
1000
>>> my.name
'kim’
>>> my.balance
1000
```

객체 변수와 클래스 변수

- 객체 변수(object variable)
  - 각 객체마다 기억공간이 할당되는 실체 변수(instance variable)
- 클래스 변수(class variable)
  - 클래스에 하나 존재하여 그 클래스의 모든 객체가 공유하는 정적 변수

```
class Account:
    counter = 0
    def __init__(self, myname):
        self.name = myname
        self.balance = 0
        Account.counter += 1
    def __del__(self):
        Account.counter -= 1
    def getCounter(self):
        return Account.counter
    ...
```

초기화한 멤버 변수는 클래스 변수!

객체의 생성 및 사용

```
>>> kim = Account('kim')
>>> lee = Account('lee')
>>> kim.getCounter()
2
>>> Account.counter
2
```

접근제어

- 접근 지정자가 별도로 없고 작명법으로 접근제어를 한다.

public	private	protected
밑줄로 시작하지 않는 이름	두 개의 밑줄로 시작하는 이름	한 개의 밑줄로 시작하는 이름

```
class Account:
    def __init__(self, name):
        self.__name = name
        self.__balance = 0
    def getBalance(self):
        return self.__balance
    def deposit(self, amount):
        self.__balance += amount
        return self.__balance
    ...
```

__name과 __balance는 private 변수

객체의 생성 및 사용

```
>>> my = Account('kim')
>>> my.__balance
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Account' object has no attribute '__balance’
>>> my.deposit(300000)
>>> my.getBalance()
300000
```

11.7 C++ 객체 및 클래스

```
#include <iostream>
#include <string>
using namespace std;

class Account {
private:
    string name;
    long balance;
public:
    Account(string name) {
        this->name = name;
        balance = 0;
    }
    long getBalance() {
        return balance;
    }
    long deposit(long amount) {
         balance += amount;
         return balance;
    }
    long withdraw(long amount) {
        if (amount <= balance)
           balance -= amount;
        else cout << "잔액 부족" << endl;
        return balance;
    }
};
```

객체를 만드는 방법

1. 클래스 타입으로 변수를 선언

   Account acc1; → 객체가 만들어짐

2. new 연산자를 이용한 동적 객체 생성

- 예

```
int main(void) {
    Account acc1("kim"); → acc1 객체가 만들어짐
    acc1.deposit(500000);
    cout << acc1.getBalance() << endl;
    acc1.withdraw(100000);
    cout << acc1.getBalance() << endl;
}
```

[실행결과]

```
500000
400000
```

C++에서 동적 객체 생성

- new T
  - T 타입의 객체를 생성하고 새로 생성된 객체에 대한 포인터 p 반환한다.

- delete p (→ 포인터)
  - p가 가리키는 객체를 제거한다.

- 예

```
Account *p = new Account( );

(*p).deposit()
p->deposit()
```

예제: 동적 객체 생성

```
int main(void) {
    Account *p = new Account("chang");
    p->deposit(400000);
    cout << p->getBalance() << endl;
    p->withdraw(200000);
    cout << p->getBalance() << endl;
}
```

실행결과

400000

200000

C++ 템플릿

- 템플릿(template)
  - C++ 프로그래밍 언어의 제네릭 프로그래밍 기능으로,
  - 함수나 클래스를 Java의 제네릭처럼 포괄적으로 동작하도록 작성

- 템플릿 함수

```
template <typename 변수이름>
함수 정의;
```

포괄적으로정의

- [예제 12]

```
template <typename T>
T sum(T a, T b) {
    return a+b;
}
```

```
sum<int>(1,2);

sum<string>("hello", "world");
```

템플릿 클래스

- 클래스를 타입 매개변수를 사용하여
- 여러 타입에 포괄적으로 적용될 수 있도록 작성

```
template <typename 변수이름>
class 클래스이름 {
        // 클래스 정의
}
```

- [예제 13]

```
template <typename T>
class Box {
private:
    T data;
public:
    Box(T d) {data = d;}
    void set(T d) { data = d;}
    T get() { return data;}
};

Box<int> a(10);
Box<string> b("hello");
```