JAVA 입문 : 이론과 실습

제 5장 클래스

목차

클래스와 객체 [1/5]

- 클래스(Class)
  - 자바 프로그램의 기본 단위
    - 재사용성(reusability), 이식성, 유연성 증가
  - 객체를 정의하는 템플릿
    - 객체 자료형(object type)
    - 하나의 사용자 정의 자료형(User-defined data type)
  - 자료 추상화(data abstraction)의 방법
- 객체(Object)
  - 클래스의 인스턴스로 변수와 같은 역할
  - 객체를 정의하기 위해서는 해당하는 클래스를 정의

클래스와 객체 [2/5]

- 클래스 선언 형태

```
[modifier] class ClassName {
        // field declarations
        // method declarations
```

public, final, abstract

객체의 구조를 기술하는 자료 부분(변수, 상수)

객체의 행위를 정의하는 메소드 부분

- public
  - 다른 패키지에서 사용 가능
  - 하나의 소스 파일에는 한 개 이하의 public 클래스
    - 소스 파일의 이름은 반드시 public 클래스 이름과 동일

클래스와 객체 [3/5]

- 클래스 선언 예

```
public  class Fraction {                    // 분수 클래스
        int numerator;                      // 분자 필드
        int denominator;                    // 분모 필드

        public Fraction add(Fraction f)  /* ... */   // 덧셈 메소드
        public Fraction mul(Fraction f)  /* ... */   // 곱셈 메소드
        public void printFraction()      /* ... */   // 프린트 메소드
}
```

- 객체 선언
  - 객체를 참조(reference)하는 변수를 선언
  - 예 : **Fraction f1, f2;**

클래스와 객체 [4/5]

- 객체 생성
  - f1 = **new** Fraction();
  - **Fraction f1 = new Fraction();**

생성자

생성자 : 객체를 생성할 때 초기화를 위해 컴파일러에 의해 자동으로 호출되는 루틴

- 객체 참조
  - 필드의 참조 형태 : **objectName.fieldName**
  - 메소드의 참조 형태 : **objectName.methodName**

```
f1.numerator
f1.add(f2)
```

클래스와 객체 [5/5]

[예제 5.1- SubPartOfFraction.java]

```
class Fraction {
   int numerator;                    // 분자 필드
   int denominator;                  // 분모 필드

   Fraction(int num, int denom) { // 생성자
      numerator = num;
      denominator = denom;
   }
   public void printFraction() {
       System.out.println("numerator + "/" + denominator);
   }
}
public class SubPartOfFraction {
   public static void main(String[] args) {
      Fraction f = new Fraction(1,2);
      f. printFraction();
   }
}
```

실행 결과 :

```
1/2
```

필드

- 객체의 구조를 기술하는 자료 부분
  - 객체의 상태를 표시
- 필드 선언
  - 형태 : **[qualifier] DataType fieldNames;**

    where, qualifier : ❶ access modifier
    ❷ static, final, volatile
  - 예

```
int  anInteger, anotherIntegrer;
public String usage;
static long idNum = 0;
public static final double earthWeight = 5.97e24;
```

접근 수정자 [1/3]

- 접근 수정자(access modifier)
  - 다른 클래스에서 필드의 **접근 허용 정도**를 나타내는 부분
  - **public, protected, private**

접근 수정자	클래스	서브 클래스	같은 패키지	모든 클래스
private	O	X	X	X
package	O	X	O	X
protected	O	O	O	X
public	O	O	O	O

  - 선언 예

```
private int i;          // private
int j;                  // package
protected int k;        // protected
public int sum;         // public
```

접근 수정자 [2/3]

- **private**
  - 정의된 클래스 내에서만 필드 접근이 허용
  - 필드에 저장된 값을 외부에서 변경하는 경우를 제한
  - 예)

```
class PrivateAccess {
    private int iamPrivate;
    // …
}
```

```
class AnotherClass {
    void accessMethod() {
        PrivateAccess pa = new PrivateAccess();
        pa.iamPrivate = 10;  // 에러
    }
}
```

접근 수정자 [3/3]

- **public**
  - 모든 클래스 및 패키지에서 자유롭게 접근

```
class PublicAccess {
    public int iamPublic;
    // …
}
```

```
class AnotherClass {
    void accessMethod() {
        PublicAccess pa = new PublicAccess();
        pa.iamPublic = 10;  // OK
    }
}
```

- **package**
  - 접근 수정자 없이 선언된 필드
  - 동일 패키지 내에서 자유롭게 접근
- **protected**
  - 동일 패키지 및 서브클래스에서 필드의 접근 (6장 참고)

자격자 – static

- **static**
  - 정적 변수, 클래스 변수(class variable)
  - 클래스 단위로 존재하여 그 클래스로부터 만들어진 모든 객체가 **공유**할 수 있는 변수

```
class StaticVariable {
        public static int numOfItems = 0;
        public int value;
        //…
}
//…
StaticVariable p = new StaticVariable();
```

  - 정적 변수 참조
    - **StaticVariable.numOfItems 가능 (객체 생성 없이)**

자격자 – final, volatile

- **final**
  - 값이 변할 수 없다는 속성
  - static + final ⇨ 상수

```
class QualifierExample {
        int anInteger;
        static int staticInteger = 0;
        static final double  π = 3.14159265358979323846;
 }
```

- **volatile**
  - 특정한 필드에 대하여 다른 곳에서 값이 변할 수 있기 때문
  - **상수 전파**와 같은 최적화를 방지하고자 할 때 사용

메소드 [1/6]

- 객체의 행위를 기술하는 방법
  - 프로그램 코드를 포함하고 있는 함수의 형태
  - 객체는 메소드 호출을 통하여 객체에 대한 작업을 수행
- 선언 형태

```
[qualifier] returnType methodName(parameterList) {
        // method body
}
```

  - **qualifier** : 접근 수정자, **static, final, native, synchronized**
  - returnType : 반환 값이 없으면 void
  - 메소드 이름의 첫 글자는 일반적으로 소문자로 씀.
    - 참고 : 클래스 이름의 첫 글자는 일반적으로 대문자로 씀.

메소드 [2/6]

- 메소드 선언 예
  - Simple method

```
class MethodExample {
        int simpleMethod() {
                //...
        }
        public void emptyMethod()  {  }
}
```

  - toString method

```
public String toString() {
        String form = numerator + "/" + denominator;
        return form;
}
```

    - 객체의 외부 표현(external representation)을 위해 객체를 스트링으로 변환하는 메소드
    - 객체를 +나 +=과 같은 스트링 연산자의 피연산자로 사용하면, toString 메소드가 묵시적으로 호출

```
"string value = " + obj
```

메소드 [3/6]

[예제 5.2- ExampleOftoString.java]

```
class Fraction {
   int numerator;                    // 분자
   int denominator;                  // 분모

   Fraction(int num, int denom) { // 생성자
      numerator = num;
      denominator = denom;
   }
   public String toString() {
      String form = numerator + "/" + denominator;
      return form;
   }
}
public class ExampleOftoString {
   public static void main(String[] args) {
      Fraction f = new Fraction(1,2);
      System.out.println("Implicit call = " + f);
      System.out.println("Explicit call = " + f.toString());
   }
}
```

실행 결과 :

Implicit call = 1/2 **Explicit call = 1/2**

메소드 [4/6]

- 메소드 자격자
  - 접근 수정자
    - 다른 클래스에서 메소드의 접근 허용 정도
    - 필드의 접근 수정자와 의미가 동등
  - **static**
    - 정적 메소드(static method), 클래스 메소드(class method)
    - 전역 함수(global function)와 같은 역할
    - 해당 클래스의 정적 필드 혹은 다른 정적 메소드만을 사용
    - 클래스 이름만으로도 참조 가능

```
ClassName.methodName;
```

메소드 [5/6]

- **final**
  - 최종 메소드(final method)
  - 서브클래스에서 재 정의할 수 없는 속성
- **synchronized**
  - 동기화 메소드
  - 항상 하나의 스레드만이 접근할 수 있도록 제어하는 기능
- **native**
  - C 언어와 같은 다른 프로그래밍언어로 쓰여진 구현 부분을 이용

메소드 [6/6]

[예제 5.3- StaticMethod.java]

```
class Count {
   public static int scount = 0;
   public int count = 0;
   public static void sIncrement() {
      scount ++
   }
   public void increment() {
      count ++;
   }
}
public class StaticMethod {
   public static void main(String[] args) {
      Count c = new Count();
      Count d = new Count();
      c.increment(); Count.sIncrement();
      d.increment(); d.sIncrement();

      System.out.print("Instance Value: c.count = " + c.count );
      System.out.println(", Static Value: c.scount = " + c.scount );
      System.out.print("Instance Value: d.count = " + d.count );
      System.out.println(", Static Value: Count.scount = " + Count.scount );
      System.out.print("Shared Value ? " + ( c.scount == d.scount ));
   }
}
```

실행 결과 :

```
Instance Value: c.count = 1, Static Value: c.scount = 2
Instance Value: d.count = 1, Static Value: Count.scount = 2
Shared Value ? true
```

매개변수 [1/4]

- 메소드 내에서만 참조될 수 있는 **지역 변수**

```
class Fraction {
        int numerator, denominator;                              // 필드
        public Fraction(int numerator, int denominator) {  // 매개 변수
              // ...
        }
  }
```

- 매개변수 전달
  - **값 호출(call by value)**
  - **참조 호출(call by reference)**

```
void parameterPass(int i, Fraction f) {
        // ...
  }
```

매개변수 [2/4]

[예제 5.4- CallByValue.java]

```
public class CallByValue {
   public static void swap(int x, int y) {
      int temp;
      temp = x; x = y; y = temp;
      System.out.println("  swap: x = " + x + ", y = " + y);
   }
   public static void main(String[] args) {
      int x = 1, y = 2;
      System.out.println("before: x = " + x + ", y = " + y);
      swap(x, y);
      System.out.println("  after: x = " + x + ", y = " + y);
   }
}
```

실행 결과 :

```
before: x = 1, y = 2
  swap: x = 2, y = 1
  after: x = 1, y = 2
```

매개변수 [3/4]

[예제 5.5- CallByReference.java]

```
class Swap {
    public int x, y;
    public static void swap(Swap obj) {
        int temp;
        temp = obj.x; obj.x = obj.y; obj.y = temp;
        System.out.println("  swap: x = " + obj.x + ", y = " + obj.y);
    }
}
public class CallByReference {
        public static void main(String[] args) {
        Swap a = new Swap();
        a.x = 1; a.y = 2;
        System.out.println("before: x = " + a.x + ", y = " + a.y);
        Swap.swap(a);
        System.out.println("  after: x = " + a.x + ", y = " + a.y);
    }
}
```

실행 결과 :

```
before: x = 1, y = 2
swap: x = 2, y = 1
after: x = 2, y = 1
```

매개변수 [4/4]

- main 메소드의 매개변수
  - main 메소드의 형태

```
public static void main(String[] args) {
        // …
  }
```

    - 명령어 라인에서 전달

  - 명령어 라인 : **args[0] args[1] args[2]**

```
java  ClassName     args1     args2     args3
```

메소드 중복 [1/3]

- **메소드 중복(method overloading)**
  - 메소드 이름은 같은데 매개 변수의 개수와 형이 다른 경우

```
void  methodOver(int i)  {  /* . . . */  }        // 첫 번째 형태
void  methodOver(int i, int j)  {  /* . . . */ }  // 두 번째 형태
```

  - 메소드가 중복된 경우, 호출 시 구별은 컴파일러가 행함.

- **시그네춰(signature)**
  - 메소드를 구별하는데 쓰이는 정보
    1. 메소드의 이름
    2. 매개 변수의 개수
    3. 매개 변수의 형

메소드 중복 [2/3]

[예제 5.8- MethodOverloading.java]

```
public class MethodOverloading {
  void someThing() {
    System.out.println("someThing() is called.");
  }
  void someThing(int i) {
    System.out.println("someThing(int) is called.");
  }
  void someThing(int i, int j) {
    System.out.println("someThing(int,int) is called.");
  }
  public static void main(String[] args) {
    MethodOverloading m = new MethodOverloading();
    m.someThing();
    m.someThing(526);
    m.someThing(54, 526);
  }
}
```

실행 결과 :

```
someThing() is called.
someThing(int) is called.
someThing(int,int) is called.
```

메소드 중복 [3/3]

[예제 5.9- OverloadingByArgType.java]

```
class MethodOverloading {
   void someThing(int i) {
      System.out.println("someThing(int) is called.");
   }
   void someThing(double d) {
      System.out.println("someThing(double) is called.");
   }
}
public class OverloadingByArgType {
   public static void main(String[] args) {
      MethodOverloading m = new MethodOverloading();
      m.someThing(526);
      m.someThing('c');
      m.someThing(5.26f);
      m.someThing(5.26);
   }
}
```

실행 결과 :

```
someThing(int) is called.
someThing(int) is called.
someThing(double) is called.
someThing(double) is called.
```

생성자(Constructor) [1/5]

- 객체가 new 연산자에 의해 생성될 때 자동으로 불려지는 메소드
  - **이름** : 클래스 이름과 동일
  - **복귀형** : 명시하지 않음
  - **기능** : 주로 객체를 초기화하는 작업

```
class Fraction {
        // ...
        Fraction(int a, int b) {
                numerator = a;
                denominator = b;
        }
    }
```

  - **Fraction f = new Fraction(1,2);**

생성자(Constructor) [2/5]

- 실행 시점
  - 클래스 내의 필드들이 모두 초기화된 후
  - [예제 5.10] 테스트 – 교과서 195쪽
- 디폴트 생성자(default constructor)
  - 매개 변수를 갖지 않는 생성자를 의미
  - 생성자를 정의하지 않았을 때
- **this()**
  - 명시적으로 클래스 내에 있는 다른 생성자를 호출
  - 반드시 생성자의 첫 문장

생성자(Constructor) [3/5]

[예제 5.10 - OverloadedConstructor.java]

```
class Fraction {
  int numerator;                    // 분자 필드
  int denominator;                  // 분모 필드
  Fraction() {                      // 디폴트 생성자
    numerator = 0;
    denominator = 1;
  }
  Fraction(int num) {               // 생성자
    numerator = num;
    denominator = 1;
  }
  Fraction(int num, int denom) {    // 생성자
    numerator = num;
    denominator = denom;
  }
  public String toString() {
    String form = numerator + "/" + denominator;
    return form;
  }
}
public class OverloadedConstructor {
  public static void main(String[] args) {
    Fraction f1 = new Fraction();
    Fraction f2 = new Fraction(2);
    Fraction f3 = new Fraction(1,2);
    System.out.println("f1 = " + f1 + ", f2 = " + f2 + ", f3 = " + f3);
  }
}
```

실행 결과 :

f1 = 0/1, f2 = 2/1, f3 = 1/2

생성자(Constructor) [4/5]

[예제 5.11 - ExecConstructor.java]

```
public class ExecConstructor {
  int a = 1;
  ExecConstructor() {
        System.out.println("a = " + a);
        a = 0;
  }
   public static void main(String[] args) {
        ExecConstructor obj = new ExecConstructor();
        System.out.println("a = " + obj.a);
   }
}
```

실행 결과 :

```
a = 1
a = 0
```

생성자(Constructor) [5/5]

[예제 5.12 - ThisConstructor.java]

```
public class ThisConstructor {
  ThisConstructor() {
        System.out.println("Default Constructor");
  }
  ThisConstructor(int a) {
        this();
        System.out.println("Constructor with one parameter");
  }
  public static void main(String[] args) {
        ThisConstructor  obj = new ThisConstructor(10);
        System.out.println("End of main");
  }
}
```

실행 결과 :

```
Default Constructor
Constructor with one parameter
End of main
```

정적 초기화문 [1/3]

- 클래스 안에 선언된 정적 변수를 초기화할 때 함께 실행되는 문장
- 형식

```
static {  <문장>  }
```

정적 초기화문 [2/3]

- 실행 순서
  - 정적 초기화문과 정적 변수를 초기화하는 순서는 소스 프로그램에 존재하는 순서

```
class Initializers {
        static {   i = j + 2; }      // 에러
        static int i, j;
        static {
            j = 4;
        }
        //...
}
```

  - 생성자보다 먼저 실행
    - [예제 5.13] 테스트 – 교과서 199쪽

정적 초기화문 [3/3]

[예제 5.13 - StaticInitializer.java]

```
public class StaticInitailizer {
  StaticInitailizer() {
        System.out.println("in Constructor");
  }
  static {
        System.out.println("in static initializer");
  }
   public static void main(String[] args) {
        System.out.println("Start of main");
        StaticInitailizer s = new StaticInitailizer();
        System.out.println("End of main");
   }
}
```

실행 결과 :

```
in static initializer
Start of main
in Constructor
End of main
```

finalize 메소드 [1/2]

- Garbage Collector
  - 자동적인 메모리 관리
- finalize 메소드
  - 가비지 콜렉터가 메모리를 회수하기 전에 그 객체의 finalize 메소드를 호출
  - 자원을 해제할 수 있는 방법 제공
  - 프로그래머는 finalize 메소드를 통하여 가비지 콜렉터가 회수할 수 없는 자원을 직접 제거

```
protected void finalize() throws Throwable {
        // ...
}
```

[예제 5.14] 테스트 – 가비지 콜렉터의 호출이 이루어지지 않음

finalize 메소드 [2/2]

[예제 5.14 - BeyondMain.java]

```
public class BeyondMain {
  static {
        System.out.println("Initialize");
  }
  protected void finalize() throws Throwable {
        System.out.println("Clean Up");
  }
   public static void main(String[] args) {
        System.out.println("in main …");
   }
}
```

실행 결과 :

```
Initialize
in main …
```

중첩 클래스 [1/3]

- 클래스의 내부에서 정의된 클래스
  - 클래스 내부에서 사용하고자 하는 객체형을 정의할 수 있는 방법을 제공
  - 클래스의 참조 범위를 제한하여 클래스의 이름 충돌 문제를 해결
  - 정보 은닉화(information hiding)

```
class OuterClass {
        // ...
        class InnerClass {
                // ...
        }
}
```

중첩 클래스 [2/3]

- 이름 참조
  - OuterClass 내부 : InnerClass 단순명을 사용
  - OuterClass 외부 : **OuterClass.InnerClass**

```
public static void main(String[] args) {
        OuterClass  outObj = new  OuterClass();
        OuterClass.InnerClass   inObj = outObj.new  InnerClass();
}
```

[예제 5.15] 테스트

- 접근 수정자
  - public, private, protected

중첩 클래스는 정적 변수를 가질 수 없음

중첩 클래스 [3/3]

[예제 5.14 - NestedClass.java]

```
class OuterClass {
 class InnerClass {
        private int value;
        InnerClass(int i) {
             value = i;
        }
        void print() {
        System.out.println("value of Inner class = " + value);
        }
 }
 public void link(int i) {
        InnerClass inObj = new InnerClass(i);
        inObj.print();
 }
} //end of OuterClass
public class NestedClass {
    public static void main(String[] args) {
        OuterClass outObj = new OuterClass();
        outObj.link(1);
        OuterClass.InnerClass inObj = outObj.new Inner Class(2);
        inObj.print();
    }
}
```

실행 결과 :

```
value of Inner class = 1
value of Inner class = 2
```

정적 중첩 클래스 [1/2]

- 정적 중첩 클래스
  - 지정어 static을 사용하여 정의
  - 정적 변수 사용 가능

```
class OuterClass {
        // ...
        static class InnerClass {
                static  int staticVariable;
                // ...
         }
}
```

OuterClass.InnerClass

- 객체 생성없이 참조 가능

정적 중첩 클래스 [2/2]

[예제 5.15 - StaticInnerClass.java]

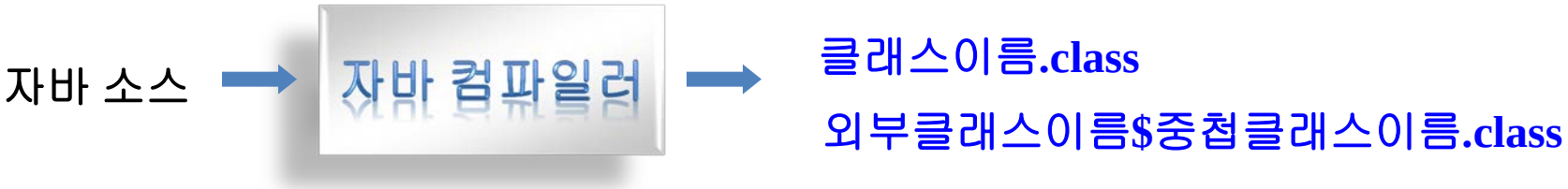
```
class OuterClass {
 static class InnerClass {
        static String str;
        InnerClass(String s) {
            str = s;
        }
        void print() {
            staticPrint(str);
        }
        static void staticPrint(String s) {
            str = s;
            System.out.println(s);
        }
   } //end of InnerClass
} //end of OuterClass
public class StaticInnerClass {
    public static void main(String[] args) {
        String s = "… without creating Outer-class object";
        OuterClass.InnerClass p = new OuterClass.InnerClass(s);
        p.print();
        OuterClass.InnerClass.staticPrint("call static method");
        p.print();
    }
}
```

실행 결과 :

```
… without creating Outer-class object
call static method
call static method
```

중첩 클래스의 명칭

- 모든 클래스에 대해 ***.class** 파일을 생성
  - 명칭의 형태: **외부클래스이름$중첩클래스이름**




```
class Outer {
        class Inner1 {
            class Inner2 {    // …        }
            // ...
        }
        // ...
 }
```

Outer.class
Outer$Inner1.class
Outer$Inner1$Inner2.class

자료 추상화 [1/2]

- **Data Abstraction**
  - 자료 구조와 함께 그에 해당하는 연산들을 정의
    - **Abstract Data Type ::= Data structures + Operations**
  - 기본형처럼 프로그래머가 사용
  - 재 사용할 수 있는 소프트웨어 부품화
  - 정보 은닉화(Information hiding)
  - 언어의 확장성 증가
- 자바는 클래스를 제공하여 추상 자료형을 정의

자료 추상화 [2/2]

- 클래스를 이용하여 분수를 다루는 방법을 구현
  - 분자, 분모를 가진 클래스 정의

```
class Fraction {
        int numerator;      // 분자
        int denominator;    // 분모
 }
```

  - 분수를 다루는 연산(덧셈, 뺄셈, 곱셈,나눗셈)과 출력

```
public Fraction add(Fraction);
public Fraction sub(Fraction);
public Fraction mul(Fraction);
public Fraction div(Fraction);
public String toString();
```

단원 요약 [1/2]

- 클래스
  - 자바 프로그램의 기본 단위로 객체를 정의하는 템플릿
  - 자료 추상화(data abstraction)의 방법임
- 객체
  - 클래스의 인스턴스로 변수와 같은 역할을 함
- 필드
  - 객체의 구조를 기술하는 자료 부분으로 객체의 상태를 표시
- 메소드
  - 객체의 행위를 기술하는 방법으로 프로그램 코드를 포함하고 있는 함수의 형태임

단원 요약 [2/2]

- 접근 수정자
  - 프로그램 단위에 접근할 수 있는 정도를 나타냄
  - 캡슐화(encapsulation)의 방법임
- 메소드 중복
  - 메소드 이름은 같은데 매개 변수의 개수와 형이 다른 경우로, 호출 시 구별은 컴파일러가 함
- 생성자
  - 객체 생성시 자동으로 불려지는 메소드로, 주로 객체를 초기화하는 작업을 함