기본 자료형 클래스

2017-08-17

이승진

**학습목표**

업캐스팅과 다운캐스팅을 이해한다.

배열 객체 메모리 구조를 이해한다.

기본 자료형 클래스를 사용하는 이유를 이해한다.

autoboxing, unboxing 개념을 이해한다.

정적 멤버(static member) 개념과 구현 방법을 이해한다.

**목차**

[1. 객체 배열 2](#_Toc490726682)

[1) up-casting & down-casting 2](#_Toc490726683)

[2) 배열 클래스 5](#_Toc490726684)

[3) Object[] 7](#_Toc490726685)

[2. 기본 자료형 클래스 9](#_Toc490726686)

[1) 기본 자료형 클래스 9](#_Toc490726687)

[2) autoboxing & unboxing 11](#_Toc490726688)

[3) nullable integer 14](#_Toc490726689)

[3. static member 15](#_Toc490726690)

[1) static member variable and static method 15](#_Toc490726691)

[2) static 메소드의 제약 17](#_Toc490726692)

[3) static constructor 18](#_Toc490726693)

[4. 구현 실습 24](#_Toc490726694)

[1) 객체 배열 (Exercise1.java) 24](#_Toc490726695)

[2) Properties (Exercise2.java) 24](#_Toc490726696)

# 객체 배열

## up-casting & down-casting

**Casting.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | package net.skhu.lecture02;  class Parent {  String name;  Parent(String name) {  this.name = name;  }    @Override  public String toString() {  return name;  }  }  class Child extends Parent {  int age;  Child(String name, int age) {  super(name); // 부모 클래스 생성자 호출  this.age = age;  }    @Override  public String toString() {  return name + "(" + age + ")";  }  }  public class Casting {    public static void main(String[] args) {  Parent p;  Child c;    p = new Parent("부모1");  c = new Child("아들1", 18);  p = new Child("아들2", 16);    //c = new Parent();  //c = p;  System.out.println(p);  System.out.println(c);  }  } |

|  |
| --- |
| class Parent {  String name;  Parent(String name) {  this.name = name;  }    @Override  public String toString() {  return name;  }  } |

Parent 클래스를 정의하였다.

### Object 클래스

Parent 클래스의 부모 클래스를 명시하지 않았지만, Object 클래스가 Parent 클래스의 부모 클래스이다.

Object 클래스는 java의 모든 클래스의 부모 클래스이기 때문이다.

### toString 메소드 재정의

toString 메소드는 Object 메소드에 정의되어 있는 메소드이다.

부모 클래스의 메소드를 자식 클래스에서 또 구현하는 것을 재정의(override)라고 부른다.

부모 클래스의 메소드를 자식 클래스에서 재정의하기 위한 규칙은 다음과 같다.

- 메소드 이름이 정확히 일치해야 한다.

- 파라미터 변수의 수와 타입이 일치해야 한다. (정확히 일치할 필요는 없다.)

- 리턴 타입이 일치해야 한다. (정확히 일치할 필요는 없다.)

### @Override

재정의된 메소드에는 @Override 어노테이션을 붙여주는 것이 바람직하다.

붙여주지 않아도 에러가 발생하는 것은 아니다.

그렇지만 @Override 어노테이션을 붙여주면, 실수를 막을 수 있어서 좋다.

오타 등의 실수 때문에, 재정의 규칙이 깨지면, 컴파일 에러가 발생하기 때문에, 실수를 막을 수 있다.

@Override 어노테이션을 붙여주지 않으면, 재정의 규칙이 깨져도, 컴파일 에러가 발생하지 않는다. 그래서 실수를 모르고 지나갈 수 있다.

|  |
| --- |
| class Child extends Parent {    int age;  Child(String name, int age) {  super(name); // 부모 클래스 생성자 호출  this.age = age;  }    @Override  public String toString() {  return name + "(" + age + ")";  }  } |

Parent 클래스를 상속하여 Child 클래스를 구현하였다.

Child 클래스 생성자 메소드의 첫 문장, 부모 클래스 생성자 호출하는 문장에 주목하자.

|  |  |
| --- | --- |
| 35  36  37  38  39  40 | Parent p;  Child c;    p = new Parent("부모1");  c = new Child("아들1", 18);  p = new Child("아들2", 16); |

(줄35) Parent 객체를 가르키기 위한 참조 변수 p를 선언함. p의 값은 null

(줄36) Child 객체를 가르키기 위한 참조 변수 c를 선언함. c의 값은 null

(줄38) Parent 타입의 참조 변수에 Parent 객체를 대입하는 것은 아무 문제 없다.

(줄39) Child 타입의 참조 변수에 Child 객체를 대입하는 것은 아무 문제 없다.

(줄40) Parent 타입의 참조 변수에 Child 객체를 대입하는 것은 허용된다. up-casting 이라고 부른다.

|  |  |
| --- | --- |
| 42  43 | //c = new Parent();  //c = p; |

(줄42) Child 타입의 참조 변수에 Parent 객체를 대입하는 것은 컴파일 오류가 발생한다.

(줄43) 참조 변수 p가 가르키는 것은 줄40에서 생성된 Child 객체이다.

이 Child 객체에 대한 참조를, Child 타입의 참조 변수 c에 대입하는 것인데도 컴파일 오류가 발생한다.

줄43에서 변수 p의 값을 변수 c에 대입하고 있다.

참조 변수 p가 가르키는 실제 객체는 Child 객체이다.

그렇지만 참조 변수 p의 타입은 Parent 타입이다.

Java 컴파일러가 컴파일 할 때, 참조 변수의 타입만 비교할 뿐, 그 참조 변수가 가르키는 것이 실제로 어떤 객체인지는 무시한다.

참조 값을 참조 변수에 대입하는 문장을 Java 컴파일러가 컴파일 할 때,

대입되는 값의 타입과 변수의 타입만 비교한다.

실제로 그 참조가 가르키는 객체가 누구인지 상관하지 않는다.

즉 줄43에서 컴파일 에러가 발생하는 이유는,

Parent 객체 타입의 참조 변수의 값을 Child 객체 타입의 참조 변수에 대입하기 때문이다.

이것은 컴파일 에러이다.

참조 변수에 동일한 타입의 참조 값을 대입하는 것은 당연히 아무 문제 없다.

자식 객체 타입의 참조 값을 부모 객체 타입의 참조 변수에 대입하는 것은 문제 없다. (up-casting)

부모 객체 타입의 참조 값을 자식 객체 타입의 참조 변수에 대입하는 것은 컴파일 에러다. (down-casting)

7

## 배열 클래스

ObjArray1.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package net.skhu.lecture02;  public class ObjArray1 {    public static void main(String[] args) {  int[] a = new int[] { 1, 2, 3 };  int[] b = new int[] { 1, 2, 3 };  String[] c = new String[] { "one", "two", "three" };    System.out.println(a.getClass().getName());  System.out.println(b.getClass().getName());  System.out.println(c.getClass().getName());  System.out.println(a.equals(b));  }  } |



위 그림에는 6개의 객체가 있다. 빨간색으로 그려진 것들이 객체이다.

int[] 객체 2개, String[] 객체 1개, String 객체 3개

하늘색으로 칠한 main은 객체가 아니고, main 메소드가 차지하는 메모리 영역이다.

메소드가 호출되자마자 메소드 메모리 영역이 할당되고, 메소드가 리턴하자마자 그 영역이 해제된다.

obj.getClass()

위 코드는 obj가 속한 클래스에 대한 정보를 담고 있는 객체를 리턴한다.

getClass() 메소드는 Object 클래스의 메소드이다.

이 메소드는, 객체가 속한 클래스에 대한 정보를 담고 있는 객체를 리턴한다.

이 메소드가 리턴하는 객체, 즉 클래스 정보 객체의 getName() 메소드는, 클래스 정보 중에서 클래스 이름을 리턴한다.

obj.getClass().getName()

위 코드는 obj가 속한 클래스의 이름을 리턴한다.

Java 언어에서 배열도 어떤 클래스의 객체이다.

배열 클래스 이름은 다음과 같다.

|  |  |
| --- | --- |
| 배열 종류 | 배열 클래스 이름 |
| int 배열 | [I |
| double 배열 | [D |
| char 배열 | [C |
| long 배열 | [L |
| float 배열 | [F |
| String 배열 | [Ljava.lang.String |
| Date 배열 | [Ljava.util.Date |

이 배열 클래스 이름을 기억할 필요는 없다.

그냥 배열도 어떤 클래스의 객체라는 것만 기억하자.

Java에서 Object 클래스는 모든 클래스의 부모 클래스이다.

배열 클래스도 Object 클래스의 자식 클래스이고, Object 클래스의 메소드를 상속 받는다.

따라서 배열 객체에도 equals, toString, hashValue 메소드를 사용할 수 있다.

그런데 배열 클래스는 Object 클래스에서 상속 받은 메소드를 자신에 맞게 재정의하지 않았다.

equals 메소드도 배열 클래스에서 재정의 (override) 되지 않았기 때문에,

배열 객체의 equals 메소드를 호출하면, 상속된 Object 클래스의 equals 메소드가 호출된다.

Object 클래스의 equals 메소드는 equality를 비교하지 않고, identity를 비교한다.

위 예제 코드에서 a 배열과 b 배열의 내용이 동일하다.

그래서 equality 비교는 true 이다.

그런데 a 배열과 b 배열은 내용이 같을 뿐, 동일 객체는 아니다.

그래서 identity 비교는 false 이다.

그래서 위 코드의 줄13은 false를 출력한다.

### 실습

줄13에서 true가 출력되게 하려면

줄6~7을 어떻게 수정하면 될까? -> 불가능

## Object[]

ObjArray2.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | package net.skhu.lecture02;  import java.util.Date;  import net.skhu.lecture01.Person;  public class ObjArray2 {    public static void main(String[] args) {  Object[] a = new Object[7];  Person p = new Person("전우치", 24);    a[0] = "hello";  a[1] = new String("world");  a[2] = new Date();  a[3] = new Person("홍길동", 18);  a[4] = new int[] { 1, 2, 3 };  a[5] = new String[] { "one", "two", "three" };  a[6] = new Person[] { new Person("임꺽정",22), p };    for (Object o : a) {  System.out.println(o);  }  }  } |



Java에서 Object 클래스는 모든 클래스의 부모 클래스이다.

Object를 부모 클래스로 선언하지 않아도 자동으로 부모 클래스가 된다.

up-casting은 아무 문제 없다.

즉 자식 클래스 객체에 대한 참조를 부모 클래스 타입의 참조 변수에 대입하는 것은 아무 문제 없다.

줄9에서 생성된 a 배열은 Object 타입의 참조를 저장하는 배열이다.

Java에서 모든 객체는 Object의 자식 클래스의 객체이므로,

어떤 객체에 대한 참조를 Object 타입의 참조 변수에 저장하는 것은 up-casting 이므로 가능하다.

String 객체, Date 객체, Person 객체, int[] 객체, String[] 객체 모두 Object 클래스의 자식 클래스 객체이다.

그래서 이 객체들에 대한 참조를 a 배열의 원소에 대입할 수 있다.

우리는 보통, 변수에 객체를 대입한다고 표현하지만,

정확하게 표현하자면, 객체를 대입하는 것이 아니고, 객체에 대한 참조만 대입하는 것이다.

기본 자료형은 객체가 아니다.

기본 자료형 int는 클래스가 아니다. int는 Object 클래스의 자식 클래스가 아니다.

a 배열은 객체에 대한 참조만 저장할 수 있기 때문에, 123 같은 정수 값을 a 배열에 그냥 넣을 수는 없다.

a 배열은 객체에 대한 참조만 저장할 수 있기 때문에, 123 같은 정수를 a 배열에 넣으려면,

먼저 123 정수를 객체로 변환해야 한다.

# 기본 자료형 클래스

## 기본 자료형 클래스

Java의 기본 자료형은 모두 값 타입 (value type) 이다.

Java 언어의 기본 자료형은 다음과 같다.

byte, short, int, long, float, double, boolean, char

|  |  |
| --- | --- |
| 1  2 | int i = 123  double d = 3.14159; |

줄1에서 변수 i에 123 값이 대입된다.

줄2에서 변수 d에 3.14159 값이 대입된다



그런데 기본 자료형을 참조형 객체로 변환해야 하는 경우가 있다.

바로 아래와 같은 경우이다.

**ObjArray3.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | package net.skhu.lecture02;  import java.util.Date;  import net.skhu.lecture01.Person;  public class ObjArray3 {    public static void main(String[] args) {  Object[] a = new Object[8];    a[0] = "hello";  a[1] = new String("world");  a[2] = new Date();  a[3] = new Person("홍길동", 13);  a[4] = new int[] { 1, 2, 3 };  a[5] = new String[] { "one", "two", "three" };  a[6] = new Integer(3);  a[7] = new Double(3.14);    for (Object o : a) {  System.out.println(o);  }  }  } |

위 코드에서 배열 a는 객체만 저장할 수 있다.

즉 참조형만 저장할 수 있고, 기본 자료형은 객체가 아니기 때문에 저장할 수 없다.

그래서 줄17, 줄18에서 3과 3.14를 직접 배열 a에 저장할 수 없어서,

기본 자료형 값을 내부에 가지고 있는 객체를 생성하여 저장했다.

즉 기본 자료형을 객체로 변화하여 저장했다.



기본 자료형을 객체로 변환하기 위해서, 기본 자료형 각각에 대한 클래스가 Java 표준 라이브러리에 포함되어 있다.

|  |  |
| --- | --- |
| 기본 자료형 | 클래스 이름 |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

앞의 예제 코드의 줄17, 줄18에서 Integer 객체, Double 객체를 생성하고 있다.

당연히 Integer, Double 클래스는 Object 클래스의 자식 클래스이다.

그래서 배열 a에 Integer, Double 객체를 넣을 수 있다. (up-casting 이므로)

위 클래스들을 기본 자료형에 대한 껍데기 클래스 (wrapper clas)라고 부른다.

이 클래스들은 기본 자료형을 객체 껍데기로 씌워서 어디에 대입하는 용도로만 사용된다.

즉 참조형을 대입해야 하는 곳에 기본 자료형을 대입해야 하는 경우에만 사용된다.

### 실습

기본 자료형 클래스에 equals 메소드가 재정의 되어 있는지 간단한 예제를 구현해서 확인하라.

재정의하지 않아서 Object 클래스의 equals가 상속된다면, equality가 아니고 identitiy가 비교될 것이다.

## autoboxing & unboxing

**ObjArray4.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | package net.skhu.lecture02;  import java.util.Date;  import net.skhu.lecture01.Person;  public class ObjArray4 {    public static void main(String[] args) {  Object[] a = new Object[10];    a[0] = "hello";  a[1] = new String("world");  a[2] = new Date();  a[3] = new Person("홍길동", 13);  a[4] = new int[] { 1, 2, 3 };  a[5] = new String[] { "one", "two", "three" };  a[6] = new Integer(3);  a[7] = new Double(3.14);  a[8] = 3;  a[9] = 3.14;    for (Object o : a) {  System.out.println(o);  }  }  } |



앞에서 다음과 같이 얘기했다.

기본 자료형은 객체가 아니다.

a 배열은 객체에 대한 참조만 저장할 수 있기 때문에, 123 같은 정수 값을 a 배열에 그냥 넣을 수는 없다.

먼저 123 정수를 객체로 변환해야 한다.

그래서 줄17~18 에서, int 값으로 Integer 객체를 생성하고,

double 값으로 Double 객체를 생성해서 a 배열에 넣았다.

그런데 줄19, 줄20 에서는 int, double을 a 배열에 그냥 넣고있다. 에러 아닌가?

에러가 아니다.

줄19, 줄20 에서도 배열에 넣어지는 것은 Integer 객체와 Double 객체이다.

줄17과 줄19는 실행될 때 같은 일이 일어난다. 줄19에서도 Integer 객체가 생성되어 배열에 넣어진다.

줄18과 줄21도 실행될 때 같은 일이 일어난다. 줄21에서도 Double 객체가 생성되어 배열에 넣어진다.

Java 컴파일러가 줄19, 줄20을 컴파일할 때, Integer 객체, Double 객체가 생성되는 코드로 변환해서 컴파일 해준다.

즉 줄19은 줄17과 동일한 바이트코드(bytecode)로 컴파일되고, 줄20은 줄18과 동일한 바이트코드로 컴파일된다.

줄19, 줄20처럼 간단하게 코딩하면, 컴파일러가 줄17, 줄18처럼 컴파일해주는 기능을 autoboxing 이라고 부른다.

기본 자료형 값이 들어있는 껍데기 객체(wrapper)를 자동으로 생성해 주는 기능이라서,

껍데기 객체를 상자(box)라고 보고 autoboxing 이라고 이름을 붙였다.

**Unboxing.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package net.skhu.lecture02;  public class Unboxing {    public static void main(String[] args) {    int i = 3;  Integer obj1 = i;  int j = obj1;    double d = 3.14;  Double obj2 = d;  double f = d;    System.out.printf("%d %d %d\n", i, obj1, j);  System.out.printf("%f %f %f\n", d, obj2, f);  }  } |



(줄8) i 변수의 값 3 이 obj1 참조 변수에 대입되는 것이 아니다.

참조 변수에 int 정수 값이 대입되는 것은 가능하지 않다.

참조 변수에는 객체에 대한 참조만 저장될 수 있다.

컴파일 결과 줄8에서 생성되는 바이트코드는 다음과 같다.

Integer obj1 = new Integer(1);

이렇게 기본 자료형 객체가 생성되는 코드를 컴파일러가 자동으로 생성해주는 기능을

autoboxing 이라고 부른다.

(줄9)에서 obj1 객체에 대한 참조가 int형 변수 j에 대입되는 것이 아니다.

컴파일 결과 줄9에서 생성되는 바이트코드는 다음과 같다.

int j = obj1.intValue();

Integer 클래스의 intValue 메소드는 int 값을 리턴한다.

이렇게 기본 자료형 객체에서 기본 자료형 값을 리턴하는 메소드를 호출하는 코드를

컴파일러가 자동으로 생성해주는 기능을 auto unboxing 이라고 부른다.

autoboxing 이름은, 기본 자료형 값이 들어간 상자 객체를 자동으로 생성해 준다는 뜻이고,

unboxing 이름은, 상자 객체에서 기본 자료형 값을 자동으로 꺼내 준다는 뜻이다.

줄12, 줄13 에서도 autoboxing, unboxing 이 일어난다.

## nullable integer

null 값은, 아무것도 참조하지 않는다는 뜻이다.

그래서 null 값은 참조형 변수에만 대입될 수 있는 값이다.

Java의 기본 자료형 int 변수에는 null을 대입할 수 없지만,

데이터베이스의 int 타입 필드에는 null을 저장할 수 있다.

null 값이 가능한 int 값을 Java 변수에 대입하려면

기본 자료형 int 변수를 사용할 수 없고

Integer 객체 참조 변수를 사용해야 한다.

즉 데이터베이스 조회 결과를 Java 변수에 저장할 때,

null 값이 가능한 int 필드 값은 Integer 객체 참조 변수에 대입해야 한다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package net.skhu.lecture02;  public class NullableInteger {    public static void main(String[] args) {  Integer intObj1 = 23, intObj2 = null;  int intValue1 = intObj1.intValue();  int intValue2 = intObj1;    int intValue3 = (intObj2 == null) ? 0 : intObj2.intValue();  int intValue4 = (intObj2 == null) ? 0 : intObj2;    System.out.printf("%d %d %d %d\n", intValue1, intValue2, intValue3, intValue4);  }  } |

(줄6) Integer intObj1 = 23;

정수 23 값이 intObj1 변수에 대입되는 것이 아니고,

autoboxing 기능에 의해서, 자동으로 new Intger(23) 코드가 실행되어 Integer 객체가 생성되고,

그 객체에 대한 참조가 intObj1 참조 변수에 대입된다.

(줄9) int intValue2 = intObj1;

intObj1 객체에 대한 참조가 intValue2 변수에 대입되는 것이 아니고,

unboxing 기능에 의해서, 자동으로 intObj1.intValue() 메소드가 호출되고

이 메소드가 리턴하는 정수 값이 intValue1 정수 변수에 대입된다.

(줄8,9) 만약 intObj1 참조 변수 값이 null 이라면, null exception error가 발생한다.

(줄11,12) intObj2 참조 변수 값이 null 이라고 해도 에러가 발생하지 않는다.

이렇게 코딩하는 것이 바람직하다.

# static member

## static member variable and static method

Fraction.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73 | package net.skhu.lecture02;  class Fraction {  private static int instanceCount = 0;  private int numerator;  private int denominator;  Fraction(int numerator, int denominator) {  ++instanceCount;  int gcd = greatestCommonDivisor(numerator, denominator);  this.numerator = numerator / gcd;  this.denominator = denominator / gcd;  }    @Override  public String toString() {  return (numerator + "/" + denominator);  }    @Override  public boolean equals(Object obj) {  if ((obj instanceof Fraction) == false) return false;  Fraction f = (Fraction)obj;  return this.numerator == f.numerator && this.denominator == f.denominator;    }  public static int getInstancenCount() {  return instanceCount;  }  public static Fraction add(Fraction f1, Fraction f2) {  int numerator = (f1.numerator \* f2.denominator) + (f2.numerator \* f1.denominator);  int denominator = f1.denominator \* f2.denominator;  return new Fraction(numerator, denominator);  }  public static Fraction subtract(Fraction f1, Fraction f2) {  int numerator = (f1.numerator \* f2.denominator) - (f2.numerator \* f1.denominator);  int denominator = f1.denominator \* f2.denominator;  return new Fraction(numerator, denominator);  }  public static Fraction multiply(Fraction f1, Fraction f2) {  int numerator = f1.numerator \* f2.numerator;  int denominator = f1.denominator \* f2.denominator;  return new Fraction(numerator, denominator);  }  public static Fraction divide(Fraction f1, Fraction f2) {  int numerator = f1.numerator \* f2.denominator;  int denominator = f1.denominator \* f2.numerator;  return new Fraction(numerator, denominator);  }  public static int greatestCommonDivisor(int x, int y) {  return y == 0 ? x : greatestCommonDivisor(y, x % y);  }  public static void main(String[] argv) {  Fraction f1 = new Fraction(1, 2);  Fraction f2 = new Fraction(1, 3);    System.out.printf("%s ＋ %s = %s\n", f1, f2, Fraction.add(f1, f2));  System.out.printf("%s － %s = %s\n", f1, f2, Fraction.subtract(f1, f2));  System.out.printf("%s × %s = %s\n", f1, f2, Fraction.multiply(f1, f2));  System.out.printf("%s ÷ %s = %s\n", f1, f2, Fraction.divide(f1, f2));    System.out.println(Fraction.add(f1, f2).equals(new Fraction(5, 6)));  System.out.printf("number of Fraction objects is %d.\n", Fraction.getInstancenCount());  }  } |



(줄62~63)

위 그림은 main 메소드의 줄 62~63을 실행한 상황에서 객체 구조 그림이다.

클래스의 객체를 객체 인스턴스(object instance)라고 부르기도 한다.

줄여서 그냥 인스턴스(instance) 라고도 부른다.

위 그림에서 Heap Segment에 Fraction 객체 인스턴스 두 개가 생성되어 있다.

(줄6~7)

numerator, denominator 멤버 변수는 인스턴스 멤버 변수이다.

인스턴스 멤버 변수는, 객체 인스턴스 메모리 안에 만들어진다.

즉 객체 인스턴스마다 인스턴스 멤버 변수를 따로 소유한다.

위 그림에서 Heap Segment 영역의 인스턴스 객체 내부에 인스턴스 멤버 변수가 들어있음을 볼 수 있다.

(줄5)

instanceCount 멤버 변수는 static 멤버 변수이다.

static 멤버 변수를 클래스 멤버 변수라고 부른다.

static 멤버 변수는 클래스에 한 개 만들어져서, 그 클래스의 모든 객체 인스턴스들에게 공유된다.

instanceCount 멤버 변수는 Fraction 클래스의 객체 인스턴스의 총 수를 세기 위한 변수이다.

정수 하나만 저장하면 되므로 이 변수를 한 개 이상 만들 필요가 없다.

그래서 인스턴스 멤버 변수가 아니고 클래스 멤버 변수로 만들었다.

객체 인스턴스는 메모리에서 heap segment 영역에 생성된다.

인스턴스 멤버 변수는 객체 내부에 들어있으므로, 인스턴스 멤버 변수도 heap segment 영역에 위치한다.

static 멤버 변수는 data segment 영역에 생성된다.

C 언어의 전역 변수와 static 지역 변수도 data segment 영역에 생성된다.

(줄10)

Fraction 객체가 생성될 때, 줄9의 생성하자 호출된다.

생성자에서 instanceCount 변수 값을 1 증가시킨다.

따라서 instanceCount 변수 값은 생성된 Fraction 객체 인스턴스 수와 같다.

## static 메소드의 제약

static 키워드가 붙은 메소드는 static 메소드이다.

static 키워드가 붙지 않은 메소드는 인스턴스 메소드 (instance method) 이다.

**static 메소드와 instance 메소드의 차이는 다음과 같다.**

\* instance 메소드에서는 this 키워드를 사용할 수 있고,

\* static 메소드에서는 this 키워드를 사용할 수 없다.

\* instance 메소드를 호출할 때에는 this 객체를 명시해 주어야 한다.

\* static 메소드를 호출할 때에는 this 객체를 명시하지 않아도 된다.

예:

|  |  |
| --- | --- |
| 1  2  3  4 | Fraction f1 = new Fraction(1, 2);  String s = f1.toString();  Fraction f2 = Fraction.add(f1, f1);  Fraction f3 = f1.add(f1, f1); |

(줄2) toString 메소드는 인스턴스 메소드이기 때문에, this 객체를 명시하였다. f1 이 this 객체이다.

(줄3) add 메소드는 static 메소드이기 때문에, this 객체를 명시하지 않았다. Fraction 은 클래스 이름이다.

(줄4) add 메소드가 static 메소드이지만, f1 객체 인스턴스로 add 메소드를 호출할 수도 있다.

f1 객체 인스턴스가 Fraction 클래스 타입이므로, Fraction 클래스 이름을 명시하는 것과 차이가 없기 때문이다.

static 메소드에서 this를 사용할 수 없기 때문에, **this.멤버변수** 형태의 코드로 멤버 변수에 접근할 수 없고,

**this.메소드()** 형태의 코드로 메소드를 호출할 수 없다. this를 생략해도 마찬가지이다.

예를 들어서 Fraction 클래스의 static 메소드의 본문에서 아래의 코드는 컴파일 에러이다.

int numerator = (this.numerator \* this.denominator) + (this.numerator \* this.denominator);

this를 생략해도 마찬가지로 컴파일 에러이다.

int numerator = (numerator \* denominator) + (numerator \* denominator);

static 메소드의 본문인 줄34의 코드를 살펴보자.

int numerator = (f1.numerator \* f2.denominator) + (f2.numerator \* f1.denominator);

위 코드는 this를 사용하고 있지 않기 때문에 컴파일 에러가 아니다.

static 메소드에서는 사용할 수 없는 것은 this 키워드 뿐이다.

this 키워드를 사용할 수 없기 때문에,

**this.멤버변수** 형태의 코드나 **this.메소드** 형태의 코드를 사용할 수 없다.

this가 아닌 다른 객체 인스턴트의 멤버 변수나 메소드는 사용할 수 있다.

static 멤버 변수나 static 메소드를 사용할 때는 this 키워드가 필요 없기 때문에

static 메소드에서 이들을 사용할 수 있다.

## static constructor

static 멤버 변수를 초기화하는 코드는 static constructor에 구현한다.

instance constructor는 객체 인스턴스가 생성될 때마다 호출되지만,

static constructor는 한 번만 호출된다.

static constructor는 main 메소드보다 먼저 호출된다.

static constructor가 포함된 클래스가 사용되기 전에 먼저 static constructor가 호출된다.

static constructor는 다음과 같은 형태이다.

|  |
| --- |
| public class Person {  static int staticVariable;  static {  staticVariable = 0;  }  } |

변수에 0을 대입하기 위해 static constructor를 구현할 필요는 없다.

아래와 같이 구현해도 되기 때문이다.

|  |
| --- |
| public class Person {  static int staticVariable = 0;  } |

static constructor 구현이 필요한 사례를 살펴보자.

### DB1.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package net.skhu.lecture02;  import java.sql.Connection;  import java.sql.DriverManager;  public class DB1 {    static String JDBC\_DRIVER\_NAME = "com.microsoft.sqlserver.jdbc.SQLServerDriver";  static String DB\_URL = "jdbc:sqlserver://127.0.0.1;databaseName=lecture2;";  static String USER\_ID = "sa";  static String USER\_PASSWORD = "test123";    public static Connection getConnection() throws Exception {  Class.forName(JDBC\_DRIVER\_NAME);  return DriverManager.getConnection(DB\_URL, USER\_ID, USER\_PASSWORD);  }  } |

위 코드에서 getConnection 메소드는, 데이터베이스 서버와 통신을 연결하고, 그 결과 객체를 리턴한다.

위 코드에서 줄8~11은 데이터베이스 서버에 연결하기 위한 정보이다.

이 연결 정보는 종종 수정되어야 한다.

연결 정보를 수정할 때마다, 위 코드를 수정하고, 다시 컴파일하고, 컴파일 결과를 다시 설치하는 일은 매우 번거롭다.

어떤 정보를 소스 코드에 직접 입력하는 것을 하드 코딩(hard coding) 이라고 부른다.

종종 수정되어야 할 사항을 하드 코딩하는 것은 바람직하지 않다.

데이터베이스 연결 정보를 별도의 텍스트 파일에 저장하고,

그 텍스트 파일의 내용을 읽어서, 줄8~11의 static 멤버 변수에 대입하는 형태로 구현하는 것이 바람직하다.

데이터베이스 연결 정보가 수정되어도 위 코드를 수정할 필요가 없어지기 때문이다.

### DB2.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | package net.skhu.lecture02;  import java.io.IOException;  import java.io.InputStream;  import java.sql.Connection;  import java.sql.DriverManager;  import java.util.Properties;  public class DB2 {    static String JDBC\_DRIVER\_NAME;  static String DB\_URL;  static String USER\_ID;  static String USER\_PASSWORD;    static {  Properties properties = new Properties();  try {  InputStream in = DB2.class.getClassLoader().getResourceAsStream("res/DB.properties");  properties.load(in);  } catch (IOException e) {  e.printStackTrace();  }  JDBC\_DRIVER\_NAME = properties.getProperty("JDBC\_DRIVER\_NAME");  DB\_URL = properties.getProperty("DB\_URL");  USER\_ID = properties.getProperty("USER\_ID");  USER\_PASSWORD = properties.getProperty("USER\_PASSWORD");  }    public static Connection getConnection() throws Exception {  Class.forName(JDBC\_DRIVER\_NAME);  return DriverManager.getConnection(DB\_URL, USER\_ID, USER\_PASSWORD);  }    public static void main(String[] args) {  System.out.println(DB2.JDBC\_DRIVER\_NAME);  System.out.println(DB2.DB\_URL);  System.out.println(DB2.USER\_ID);  System.out.println(DB2.USER\_PASSWORD);  }  } |

(줄16~28) static construcor 이다.

"res/DB.properties" 파일의 내용을 읽어서 static 멤버 변수에 값을 대입한다.

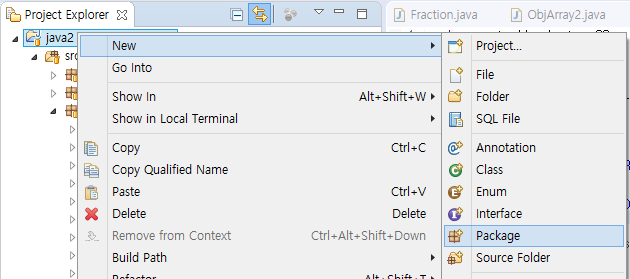
(줄19) "res/DB.properties" 파일의 내용을 읽기 위한 InputStream 객체 생성.

(줄20) InputStream 객체를 사용하여, 파일의 내용을 읽어서, 파일의 내용이 채워진 Properties 객체 생성.

(줄24~27) Properties 객체에 채워진 값을 꺼내서, static 멤버 변수에 대입.

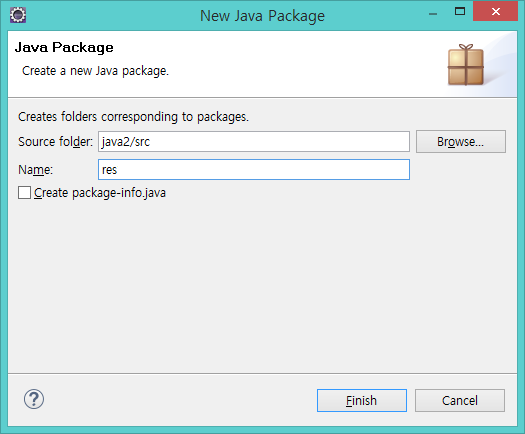
### res 패키지 생성

res 패키지 생성



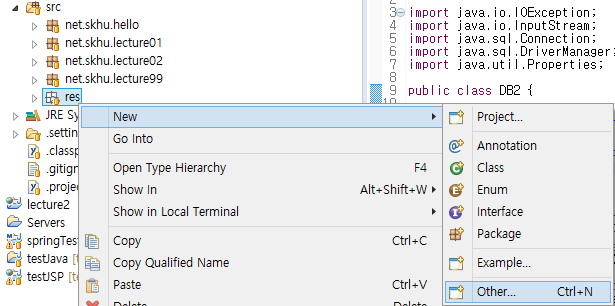
화면 왼쪽 Project Explorer 창에서 현재 프로젝트를 마우스 오른쪽 버튼으로 클릭하고,

메뉴에서 New - Package 클릭



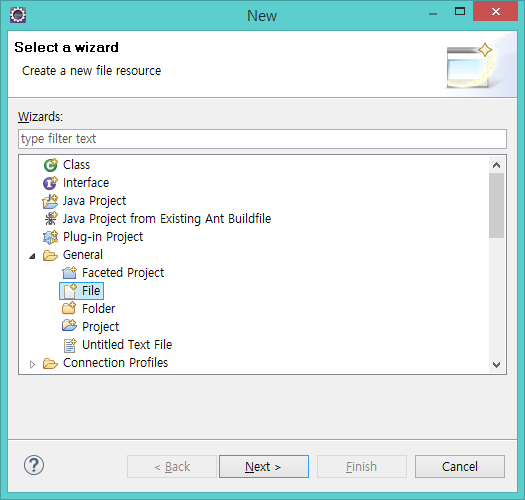
Package Name: res

### res/DB.properties 파일 생성

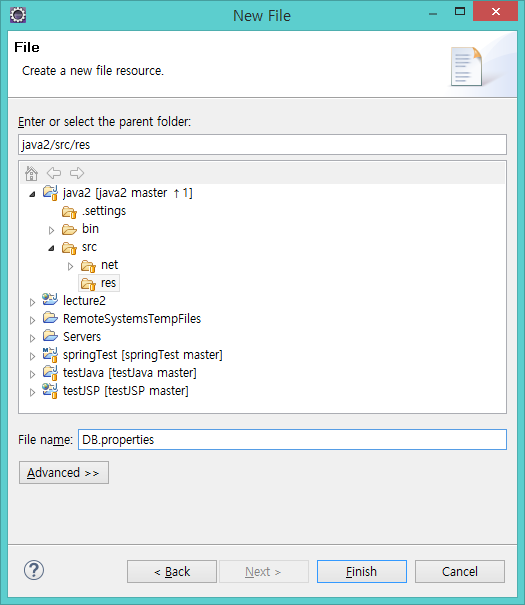


res 패키지를 마우스 오른쪽 버튼으로 클릭하고,

메뉴에서 New - Other - General - File 선택



Next 버튼 클릭



File Name: DB.properties

res/DB.properties

|  |  |
| --- | --- |
| 1  2  3  4 | JDBC\_DRIVER\_NAME=com.microsoft.sqlserver.jdbc.SQLServerDriver  DB\_URL=jdbc:sqlserver://127.0.0.1;databaseName=lecture2;  USER\_ID=sa  USER\_PASSWORD=test123 |

이 파일의 내용은, 한 줄에 데이터 하나이다.

한 줄의 내용은, **데이터\_이름=데이터\_값** 형태이다.

Java 코드에서 이 파일의 내용을 읽는 방법은 다음과 같다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Properties properties = new Properties();  InputStream in = DB2.class.getClassLoader().getResourceAsStream("res/DB.properties");  properties.load(in);  JDBC\_DRIVER\_NAME = properties.getProperty("JDBC\_DRIVER\_NAME");  DB\_URL = properties.getProperty("DB\_URL"); |

(줄1) 읽어들인 속성을 채울 Properties 객체를 생성한다.

(줄2) 파일의 내용을 읽기 위한 InputStream 객체를 생성한다.

(줄3) 파일의 내용을 읽어서 Properties 객체에 채운다.

(줄4) Properties 객체에 채워진 내용에서, 데이터 이름이 "JDBC\_DRIVER\_NAME" 인 데이터 값을 찾아서

변수에 대입한다.