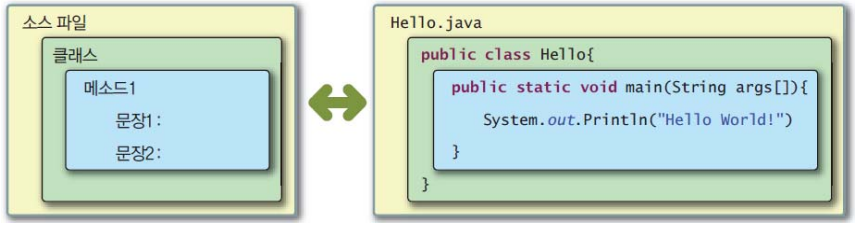
Java Basics, Array
Class, Inheritance, Interface

514770
2018년 가을학기
9/10/2018
박경신

자바 프로그램의 구조

- 클래스 (class): 객체(object)를 만드는 설계도 (템플릿)
- 자바 프로그램은 기본적으로 클래스로 구성됨



  - **public** 키워드는 Hello 클래스가 다른 클래스에서도 사용 가능함을 나타냄
  - 하나의 클래스 안에는 여러 개의 메소드가 포함될 수 있음
  - 하나의 메소드 안에는 여러 개의 문장이 포함될 수 있음

Code Block

- 여러 명령문을 논리적으로 결합해야 할 때 중괄호 ({ })를 사용하여 명령문 그룹을 만들어 표현 - 이러한 명령문 그룹을 코드 블록(code block)이라고 함
- 코드 블록 안에는 변수를 선언할 수 있고, 다른 코드 블록을 포함할 수도 있음

```
public class Example {
    public static void main(String[] args) {
        int outer;
        {
            int inner;
            outer = 1;
            inner = 2;
        }
        outer = 5;
        //inner = 10;        // 오류
    }
}
```

내부 코드블록
main() 메소드 코드블록
Example 클래스 코드블록

Identifier

- 식별자란?
  - 클래스, 변수, 상수, 메소드 등에 붙이는 이름
- 식별자의 원칙
  - '@', '#', '!'와 같은 특수 문자, 공백 또는 탭은 식별자로 사용할 수 없으나 '_', '$'는 사용 가능
  - 유니코드 문자 사용 가능. 한글 사용 가능
  - 자바 언어의 키워드는 식별자로 사용불가
  - 식별자의 첫 번째 문자로 숫자는 사용불가
  - '_' 또는 '$'를 식별자 첫 번째 문자로 사용할 수 있으나 일반적으로 잘 사용하지 않는다.
  - 불린 리터럴 (true, false)과 널 리터럴(null)은 식별자로 사용불가
  - 길이 제한 없음
- 대소문자 구별
  - Test와 test는 별개의 식별자

Keywords

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Variable

- 변수(Variable)
  - 프로그램 실행 중에 값을 임시 저장하기 위한 공간
    - 변수 값은 프로그램 수행 중 변경될 수 있음
  - 데이터 타입에서 정한 크기의 메모리 할당
  - 반드시 변수 선언과 값을 초기화 후 사용
- 변수 선언과 초기화

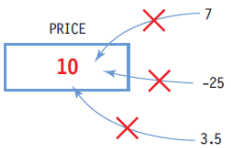
```
int radius;
char c1, c2, c3; // 3 개의 변수를 한 번에 선언
double weight;
```

```
int radius = 10;
char c1 = 'a', c2 = 'b', c3 = 'c';
double weight = 75.56;
```

```
radius = 10 * 5;
c1 = 'r';
weight = weight + 5.0;
```

Constant

- 상수 (Constant)
  - **final** 키워드 사용
  - 변하지 않는 문자나 숫자 값. 값 변경 불가
  - 선언 시 초기값 지정



- 상수 선언 사례

```
final double PI = 3.141592;
final int LENGTH = 20;
```

Data Type

- 자바의 자료형 (Data Type)
  - 기초형 (Primitive Type)
    - boolean *(1 Byte, true or false)*
    - char *(2 Bytes, Unicode)*
    - byte *(1 Byte, -128 ~ 127)*
    - short *(2 Bytes, -32768 ~ 32767)*
    - int *(4 Bytes, $-2^{31} \sim 2^{31}-1$)*
    - long *(8 Bytes, $-2^{63} \sim 2^{63}-1$)*
    - float *(4 Bytes, -3.4E38~3.4E38)*
    - double *(8 Bytes, -1.7E308~1.7E308)*
  - 참조형 (Reference Type)
    - class
    - interface
    - array

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'₩u0000'
String (or any object)	null
boolean	false

Control Statement

- 제어문의 종류
  - 제어문이란 프로그램을 실행할 이러한 문장의 논리적인 흐름
  - 조건문 – 조건식의 값에 따라 수행한다. 예) if 문, switch 문
  - 반복문 – 조건이 만족하는 동안 특정 명령문을 반복적으로 수행한다. 예) while 문, do 문, for 문, foreach 문
  - 점프문 – 제어권을 이동시킬 때 점프문을 사용한다. 예) label 문, break 문, continue 문

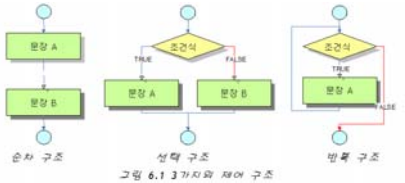
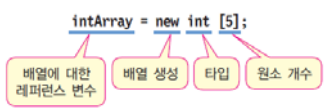
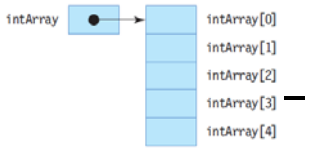


그림 6.1 3가지의 제어 구조

Array





- 배열(array)
  - 여러 개의 데이터를 같은 이름으로 활용할 수 있도록 해주는 자료 구조
    - 인덱스(Index, 순서 번호)와 인덱스에 대응하는 데이터들로 이루어진 자료 구조
    - 배열을 이용하면 한 번에 많은 메모리 공간 선언 가능
  - 배열은 같은 타입의 데이터들이 순차적으로 저장되는 공간
    - 원소 데이터들이 순차적으로 저장됨
    - 인덱스를 이용하여 원소 데이터 접근
    - 반복문을 이용하여 처리하기에 적합한 자료 구조(주로 for 또는 for-each 반복문과 많이 사용됨)
  - 배열 인덱스
    - 0부터 시작
    - 인덱스는 배열의 시작 위치에서부터 데이터가 있는 상대 위치

Array

- 배열 선언과 배열 생성의 두 단계 필요
  - 배열 선언

```
int     intArray[];
char    charArray[];
```
또는
```
int[]   intArray;
char[]  charArray;
```

  - 배열 생성

```
intArray = new int[10];
charArray = new char[20];
```
또는
```
int[] intArray = new int[10];
char[] charArray = new char[20];
```

  - 선언과 초기화
    - 배열 생성과 값 초기화

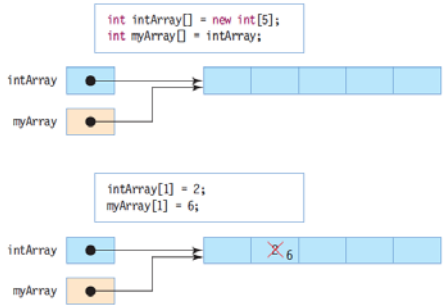
```
// 총 10개의 정수 배열 생성 및 값 초기화
int[] intArray = {0,1,2,3,4,5,6,7,8,9};
```

  - 잘못된 배열 선언

```
//int intArray[10]; // 컴파일 오류. 배열의 크기를 지정할 수 없음
```

Array

- 배열 참조
  - 생성된 1개의 배열을 다수의 레퍼런스가 참조 가능



Array

- 배열 원소 접근
  - 반드시 배열 생성 후 접근

```
int[] intArray; // 배열 선언
intArray[4] = 8; // 오류, intArray 배열의 메모리가 할당되지 않았음
```

  - 배열 변수명과 [] 사이에 원소의 인덱스를 적어 접근
    - 배열의 인덱스는 0부터 시작
    - 배열의 마지막 항목의 인덱스는 (배열 크기 – 1)

```
int[] intArray;
intArray = new int[10];

intArray[3] = 6; // 배열에 값을 저장
int n = intArray[3]; // 배열로부터 값을 읽음
```

Array

- 배열 인덱스
  - 인덱스는 0부터 시작하며 마지막 인덱스는 (배열 크기 -1)
  - 인덱스는 정수 타입만 가능

```
int[] intArray = new int[5]; // 인덱스는 0~4까지 가능
int n = intArray[-2];  // 실행 오류. -2는 인덱스로 적합하지 않음
int m = intArray[5];  // 실행 오류. 5는 인덱스의 범위(0~4)를 넘었음
```

- 배열의 크기
  - 배열의 크기는 배열 레퍼런스 변수를 선언할 때 결정되지 않음
    - 배열의 크기는 배열 생성 시에 결정되며, 나중에 바꿀 수 없음
  - 배열의 크기는 배열의 length 필드에 저장

```
int size = intArray.length;
```

Array & For-each

- For-each 문
  - 배열이나 나열(enumeration)의 각 원소를 순차적으로 접근하는데 유용한 for 문

```
int[] num = { 1,2,3,4,5 };
int sum = 0;
// 반복될 때마다 k는 num[0], num[1], ..., num[4] 값으로 설정
for (int k : num)
  sum += k;
System.out.println("합은 " + sum);
```

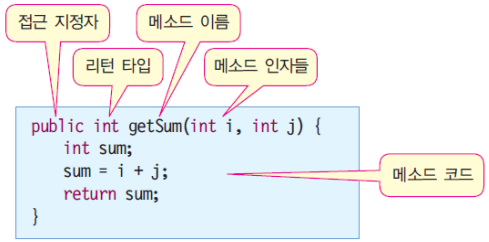
합은 15

```
String names[] = { "사과", "배", "바나나", "체리", "딸기", "포도" } ;
// 반복할 때마다 s는 names[0], names[1], ..., names[5] 로 설정
for (String s : names)
  System.out.print(s + " ");
```

사과 배 바나나 체리 딸기 포도

Method

- 메소드
  - 메소드는 C/C++의 함수와 동일
  - 자바의 모든 메소드는 반드시 클래스 안에 있어야 함(캡슐화 원칙)
- 메소드 구성 형식
  - 접근 지정자
    - public. private, protected, 디폴트(접근 지정자 생략된 경우)
  - 리턴 타입
    - 메소드가 반환하는 값의 데이터 타입



메소드에서 배열 리턴

- 메소드의 배열 리턴
  - 배열의 레퍼런스만 리턴
- 메소드의 리턴 타입
  - 메소드가 리턴하는 배열의 타입은 리턴 받는 배열 타입과 일치
  - 리턴 타입에 배열의 크기를 지정하지 않음

Return type

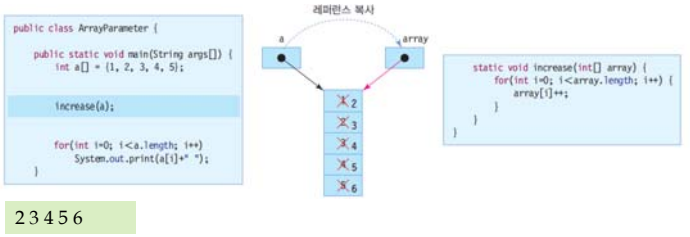
Method name

```
int[] makeArray() {
  int temp[] = new int[4];
  return temp;
}
```

Array return

매개 변수에 배열이 전달되는 경우

- 매개변수에 배열이 전달되는 경우는 배열의 reference가 복사



Parameter Passing – Primitive Type

- 자바의 인자 전달 방식(Parameter Passing)
  - **값에 의한 호출(Pass-by-value)**
  - 기본 타입(Primitive Type: 예를 들어 int, double, char)의 값을 전달하는 경우
    - 값이 복사되어 전달
    - 메소드의 매개 변수가 변경되어도 호출한 실제 인자 값은 변경되지 않음

```
int sum(int m, int n) {
   return m + n;
}

int a = 5;
int b = 7;
int c = sum(a, b);
System.out.println(c);
```

m 5

n 7

a 5

b 7

Parameter Passing – Reference Type

- 자바의 인자 전달(Parameter Passing) 방식
  - 객체(class object) 혹은 배열(array)을 전달하는 경우
    - 객체나 배열의 레퍼런스만 전달**(Pass-by-reference)**
      - 객체 혹은 배열이 통째로 복사되어 전달되는 것이 아님
    - 메소드의 매개 변수와 호출한 실인자가 객체나 배열을 공유하게 됨

```
void assignArray(int[] values) {
   for (int i=0; i<values.length; i++) {
      values[i] = i; // values 배열에 0~4까지 값을 넣어줌
   }
}

int[] intArray = new int[5]; // create 5 integer array
assignArray(intArray);
for (int v : intArray) {
   System.out.println(v);
}
```

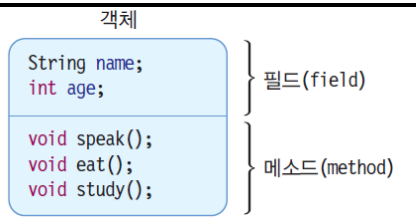
values

intArray

intArray[0]
intArray[1]
intArray[2]
intArray[3]
intArray[4]

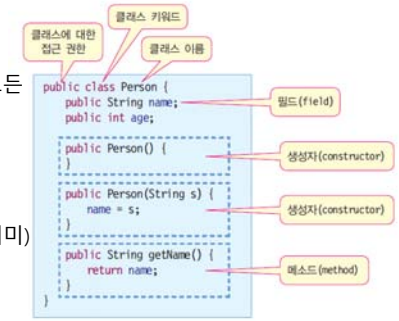
Class

- 클래스(Class)
  - 객체의 속성과 행위 선언
  - 객체의 설계도 혹은 틀
- 객체(Object)
  - 클래스의 틀로 찍어낸 실체
    - 메모리 공간을 갖는 구체적인 실체
    - 클래스를 구체화한 객체를 **인스턴스(instance)**라고 부름
    - 객체와 인스턴스는 같은 뜻으로 사용



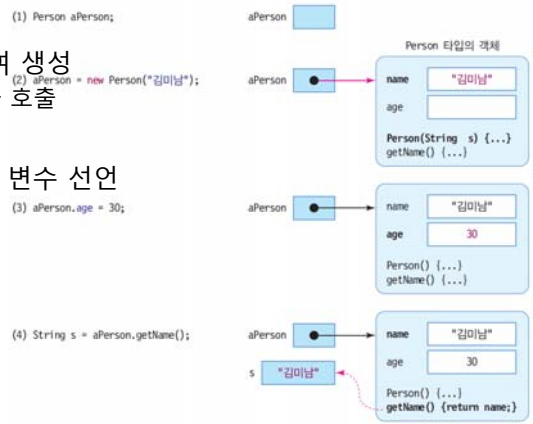
클래스 구조

- 클래스 접근 권한, public
  - 다른 클래스들에서 이 클래스를 사용하거나 접근할 수 있음을 선언
- 클래스(class)
  - Person이라는 이름의 클래스 선언
  - 클래스는 {로 시작하여 }로 닫으며 이곳에 모든 필드와 메소드 구현
- 필드(field)
  - 값을 저장할 멤버 변수 (혹은 필드)
  - 필드의 접근 지정자 public (다른 클래스의 메소드에서 호출할 수 있도록 공개한다는 의미)
- 메소드(method)
  - 메소드는 함수이며 객체의 행위를 구현
  - 메소드의 접근 지정자 public (다른 클래스의 메소드에서 호출할 수 있도록 공개한다는 의미)
- 생성자(constructor)
  - 클래스의 이름과 동일한 메소드
  - 클래스의 객체가 생성될 때만 호출되는 메소드



객체 생성 및 사용 예

- 객체 생성
  - new 키워드를 이용하여 생성
    - new는 객체의 생성자 호출
- 객체 생성 과정
  1. 객체에 대한 레퍼런스 변수 선언
  2. 객체 생성



접근 지정자

- 디폴트(default) 멤버
  - 같은 패키지 내의 다른 클래스만 접근 가능
- public 멤버
  - 패키지에 관계 없이 모든 클래스에서 접근 가능
- private 멤버
  - 클래스 내에서만 접근 가능
  - 상속 받은 하위 클래스에서도 접근 불가
- protected 멤버
  - 같은 패키지 내의 다른 모든 클래스에서 접근 가능
  - 상속 받은 하위 클래스는 다른 패키지에 있어도 접근 가능

멤버에 접근하는 클래스	멤버의 접근 지정자			
	default	private	protected	public
같은 패키지의 클래스	○	×	○	○
다른 패키지의 클래스	×	×	×	○

Constructor

- 생성자(Constructor)
  - 객체가 생성될 때 초기화를 위해 실행되는 메소드
- 생성자의 특징
  - 생성자는 메소드
  - 생성자 이름은 클래스 이름과 동일
  - 생성자는 new를 통해 객체를 생성할 때만 호출됨
  - 생성자도 오버로딩하여 여러개 작성 가능
  - 생성자는 리턴 타입을 지정할 수 없음
  - 생성자는 하나 이상 선언되어야 함
    - 개발자가 생성자를 작성하지 않았으면 컴파일러에 의해 자동으로 기본 생성자가 선언됨
    - 기본 생성자를 디폴트 생성자(default constructor)라고도 함

get & set

- 설정자(set)에서 매개 변수를 통하여 잘못된 값이 넘어오는 경우, 이를 사전에 차단할 수 있음.
- 필요할 때마다 필드값을 계산하여 반환할 수 있음.
- 접근자(get)만을 제공하면 자동적으로 읽기만 가능한 필드를 만들 수 있음.

```
public void setSpeed(int s)
{
    if( s < 0 )
        speed = 0;
    else
        speed = s;
}
```

속도가 음수이면 0으로 만든다.

Inheritance

```
class Animal {
    String name;
    int age;
    void eat() {...}
    void sleep() {...}
    void love() {...}
}
```

상속

```
class Human extends Animal {
    String hobby;
    String job;
    void work() {...}
    void cry() {...}
    void laugh() {...}
}
```

Animal의 객체

```
String name;
int age;
void eat();
void sleep();
void love();
```

Human의 객체

```
String name;
int age;
void eat();
void sleep();
void love();
String hobby;
String job;
void work();
void cry();
void laugh();
```

- 상속(Inheritance)
  - 상위 클래스의 특성을 하위 클래스가 물려받음
    - 상위 클래스 : 수퍼 클래스, 하위 클래스 : 서브 클래스
  - 서브 클래스
    - 수퍼 클래스 코드의 재사용
    - 새로운 특성 추가 가능
  - 자바는 클래스 다중 상속 없음
    - 인터페이스를 통해 다중 상속과 같은 효과 얻음

상속과 접근 지정자

- 자바의 접근 지정자 (public, protected, default, private)
  - 상속 관계에서 주의할 접근 지정자는 private와 protected
- 슈퍼 클래스의 private 멤버
  - 슈퍼 클래스의 private 멤버는 다른 모든 클래스에 접근 불허
- 슈퍼 클래스의 protected 멤버
  - 같은 패키지 내의 모든 클래스 접근 허용
  - 동일 패키지 여부와 상관없이 서브 클래스에서 슈퍼 클래스의 protected 멤버 접근 가능

슈퍼 클래스 멤버에 접근하는 클래스 종류	슈퍼 클래스 멤버의 접근 지정자			
	default	private	protected	public
같은 패키지의 클래스	○	×	○	○
다른 패키지의 클래스	×	×	×	○
같은 패키지의 서브 클래스	○	×	○	○
다른 패키지의 서브 클래스	×	×	○	○

(○는 접근 가능함을, ×는 접근이 불가함을 뜻함)

this

- this 란?
  - 현재 실행되는 메소드가 속한 객체에 대한 레퍼런스
    - 컴파일러에 의해 자동 선언 : 별도로 선언할 필요 없음

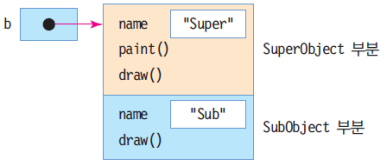
```
class Samp {
  int id;
  public Samp(int x) { id = x; }
  public void set(int x) { id = x; }
  public int get() {return id; }
}
```

=

```
class Samp {
  int id;
  public Samp(int x) { this.id = x; }
  public void set(int x) { this.id = x; }
  public int get() {return id; }
}
```

super

- super 키워드
  - super는 슈퍼 클래스의 멤버를 접근할 때 사용되는 레퍼런스
  - 서브 클래스에서만 사용
  - 슈퍼 클래스의 메소드 호출 시 사용
  - 컴파일러는 super 호출을 정적 바인딩으로 처리




```
class A {
    public A() {
        System.out.println("생성자A");
    }
    public A(int x) {
        System.out.println("매개변수생성자A" + x);
    }
}
```

```
class B extends A {
    public B() {
        System.out.println("생성자B");
    }
    public B(int x) {
        super(x);
        System.out.println("매개변수생성자B" + x);
    }
}
```

```
public class ConstructorEx4 {
    public static void main(String[] args) {
        B b;
        b = new B(5);
    }
}
```

Method Overloading

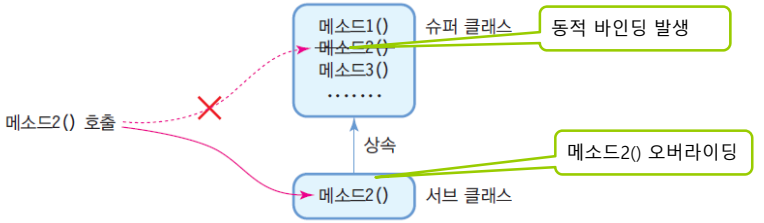
- 메소드 오버로딩(Overloading)
  - 한 클래스 내에서 두 개 이상의 이름이 같은 메소드 작성
    - 메소드 이름이 동일하여야 함
    - 매개 변수의 개수가 서로 다르거나, 타입이 서로 달라야 함
    - 리턴 타입은 오버로딩과 관련 없음

```
// 메소드 오버로딩이 성공한 사례
class MethodOverloading {
  public int getSum(int i, int j) {
    return i + j;
  }
  public int getSum(int i, int j, int k) {
    return i + j + k;
  }
  public double getSum(double i, double j) {
    return i + j;
  }
}
```

```
// 메소드 오버로딩이 실패한 사례
class MethodOverloadingFail {
  public int getSum(int i, int j) {
    return i + j;
  }
  public double getSum(int i, int j) {
    return (double)(i + j);
  }
}
```

Method Overriding

- 메소드 오버라이딩(Method Overriding)
  - 슈퍼 클래스의 메소드를 서브 클래스에서 재정의
    - 슈퍼 클래스의 메소드 이름, 메소드 인자 타입 및 개수, 리턴 타입 등 모든 것 동일하게 작성
      - 이 중 하나라도 다르면 메소드 오버라이딩 실패
  - 동적 바인딩 발생
    - 서브 클래스에 오버라이딩된 메소드가 무조건 실행되도록 동적 바인딩 됨



static vs. non-static

- non-static 멤버의 특성
  - 공간적 - 멤버들은 객체마다 독립적으로 별도 존재
    - 인스턴스 멤버라고도 부름
  - 시간적 - 필드와 메소드는 객체 생성 후 비로소 사용 가능
  - 비공유의 특성 - 멤버들은 여러 객체에 의해 공유되지 않고 배타적
- static 멤버란?
  - 객체를 생성하지 않고 사용가능
  - 클래스당 하나만 생성됨
    - 클래스 멤버라고도 부름
    - 객체마다 생기는 것이 아님
  - 특성
    - 공간적 특성 - static 멤버들은 클래스 당 하나만 생성.
    - 시간적 특성 - static 멤버들은 클래스가 로딩될 때 공간 할당.
    - 공유의 특성 - static 멤버들은 동일한 클래스의 모든 객체에 의해 공유

```
class StaticSample {
  int n; // non-static 필드
  void g() {...} // non-static 메소드

  static int m; // static 필드
  static void f() {...} // static 메소드
}
```

final

- final 클래스 - 더 이상 클래스 상속 불가능

```
final class FinalClass { .....
}
class DerivedClass extends FinalClass { // 컴파일 오류
      .....
}
```

- final 메소드 - 더 이상 오버라이딩 불가능

```
public class SuperClass {
      protected final int finalMethod() { ... }
}
class DerivedClass extends SuperClass {
      protected int finalMethod() { ... } // 컴파일 오류, 오버라이딩 할 수 없음
}
```

- final 필드 – 상수 정의

```
public class FinalFieldClass {
      static final int ROWS = 10; // 상수 정의, 이때 초기 값(10)을 반드시 설정
}
```

객체의 타입 변환

- 업캐스팅(upcasting)
  - 프로그램에서 이루어지는 자동 타입 변환
  - 서브 클래스의 레퍼런스 값을 슈퍼 클래스 레퍼런스에 대입
    - 슈퍼 클래스 레퍼런스가 서브 클래스 객체를 가리키게 되는 현상
    - 객체 내에 있는 모든 멤버를 접근할 수 없고 슈퍼 클래스의 멤버만 접근 가능

```
class Person {
}

class Student extends Person {
}
...

Student s = new Student();
Person p = s; // 업캐스팅, 자동타입변환
```

객체의 타입 변환

- 다운캐스팅(downcasting)
  - 슈퍼 클래스 레퍼런스를 서브 클래스 레퍼런스에 대입
  - 업캐스팅된 것을 다시 원래대로 되돌리는 것
  - 명시적으로 타입 지정

```
class Person {
}
class Student extends Person {
}
...

Student s = (Student)p; // 다운캐스팅, 강제타입변환
```

instanceof

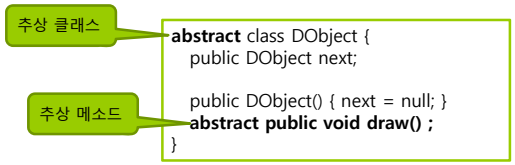
- 업캐스팅된 레퍼런스로는 객체의 진짜 타입을 구분하기 어려움
  - 슈퍼 클래스는 여러 서브 클래스에 상속되기 때문
    - 슈퍼 클래스 레퍼런스로 서브 클래스 객체를 가리킬 수 있음
- **instanceof** 연산자
  - instanceof 연산자
    - 레퍼런스가 가리키는 객체의 진짜 타입 식별
  - 사용법

객체레퍼런스 **instanceof** 클래스타입

연산의 결과 : true/false의 불린 값

Abstract Class & Method

- 추상 메소드(abstract method)
  - 선언되어 있으나 구현되어 있지 않은 메소드
    - **abstract** 키워드로 선언
    - ex) public abstract int getValue();
  - 추상 메소드는 서브 클래스에서 오버라이딩하여 구현
- 추상 클래스(abstract class)
  1. 추상 메소드를 하나라도 가진 클래스
     - 클래스 앞에 반드시 abstract라고 선언해야 함
  2. 추상 메소드가 하나도 없지만 클래스 앞에 abstract로 선언한 경우



추상 클래스의 상속

- 추상 클래스의 상속 2 가지 경우
  - 추상 클래스의 단순 상속
    - 추상 클래스를 상속받아, 추상 메소드를 구현하지 않으면 서브 클래스도 추상 클래스 됨
    - 서브 클래스도 abstract로 선언해야 함

```
abstract class DObject { // 추상 클래스
   public DObject next;
   public DObject() { next = null; }
   abstract public void draw(); // 추상 메소드
}
abstract class Line extends DObject { // draw()를 구현하지 않았기 때문에 추상 클래스
   public String toString() { return "Line";}
}
```

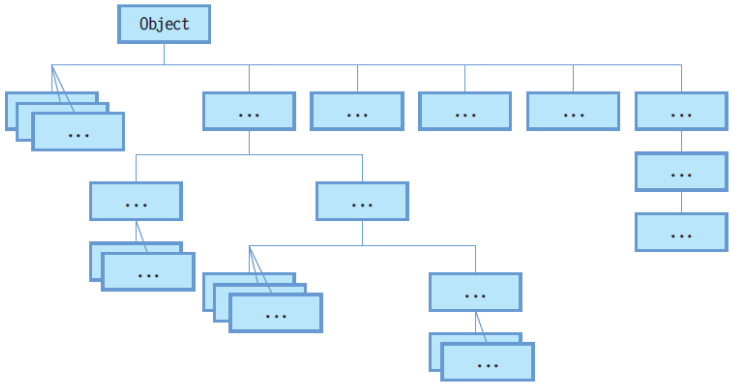
  - 추상 클래스 구현 상속
    - 서브 클래스에서 슈퍼 클래스의 추상 메소드 구현(오버라이딩)
    - 서브 클래스는 추상 클래스 아님

Polymorphism

- 다형성(Polymorphism)
  - 다형성(polymorphism)이란 객체들의 타입이 다르면 똑같은 메시지가 전달되더라도 각 객체의 타입에 따라서 서로 다른 동작을 하는 것(dynamic binding)
  - 자바의 다형성 사례
    - 슈퍼 클래스의 메소드를 서브 클래스마다 다르게 구현하는 메소드 오버라이딩(overriding)

자바의 클래스 계층 구조

자바에서는 모든 클래스는 반드시 java.lang.Object 클래스를 자동으로 상속받는다.

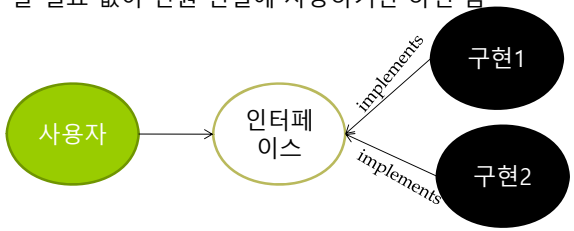


Object의 메소드

메소드	설명
protected Object clone()	현 객체와 똑같은 객체를 만들어 리턴
boolean equals(Object obj)	obj가 가리키는 객체와 현재 객체가 비교하여 같으면 true 리턴
Class getClass()	현 객체의 런타임 클래스를 리턴
int hashCode()	현 객체에 대한 해시 코드 값 리턴
String toString()	현 객체에 대한 스트링 표현을 리턴
void notify()	현 객체에 대해 대기하고 있는 하나의 스레드를 깨운다.
void notifyAll()	현 객체에 대해 대기하고 있는 모든 스레드를 깨운다.
void wait()	다른 스레드가 깨울 때까지 현재 스레드를 대기하게 한다.

인터페이스의 필요성

- 인터페이스를 이용하여 다중 상속 구현
  - 자바에서 클래스 다중 상속 불가
- 인터페이스는 명세서와 같음
  - 인터페이스만 선언하고 구현을 분리하여, 작업자마다 다양한 구현을 할 수 있음
  - 사용자는 구현의 내용은 모르지만, 인터페이스에 선언된 메소드가 구현되어있기 때문에 호출하여 사용하기만 하면 됨
    - 110v 전원 아울렛처럼 규격에 맞기만 하면, 어떻게 만들어졌는지 알 필요 없이 전원 연결에 사용하기만 하면 됨



자바의 인터페이스

- 인터페이스(interface)
  - 모든 메소드가 추상 메소드인 클래스
- 인터페이스 선언
  - **interface** 키워드로 선언
  - ex) public **interface** SerialDriver {...}
- 인터페이스의 특징
  - 인터페이스의 메소드
    - public abstract 타입으로 생략 가능
  - 인터페이스의 상수
    - public static final 타입으로 생략 가능
  - 인터페이스의 객체 생성 불가
  - 인터페이스에 대한 레퍼런스 변수는 선언 가능

인터페이스 선언

- 인터페이스 구성멤버
  - 상수필드 (constant field)
  - 추상메소드 (abstract method)
  - 디폴트메소드 (default method)
  - 정적메소드 (static method)

```
public interface 인터페이스명 {
    //상수(constant fields)
    타입 상수명 = 값;

    //추상 메소드(abstract method)
    리턴타입 메소드명(매개변수,..);

    //디폴트 메소드(default method)
    default 리턴타입 메소드명(매개변수,..) {
        ..내부구현..
    }

    //정적 메소드(static method)
    static 리턴타입 메소드명(매개변수,..) {
        ..내부구현..
    }
}
```

인터페이스 선언

```
public interface RemoteControl {
    int MAX_VOLUME = 10; //상수(constant fields)
    int MIN_VOLUME = 0; //상수(constant fields)

    //추상 메소드(abstract method)
    void turnOn();
    void turnOff();
    void setVolume(int volume);

    //디폴트 메소드(default method)
    default void setMute(boolean mute) {
        if(mute) System.out.println("무음 처리합니다.");
        else     System.out.println("무음 해제합니다.");
    }

    //정적 메소드(static method)
    static void changeBattery() {
        System.out.println("건전지를 교환합니다.");
    }
}
```

인터페이스 상속

- 인터페이스 간에도 상속 가능
  - 인터페이스를 상속하여 확장된 인터페이스 작성 가능
- 인터페이스 다중 상속 허용

```
interface MobilePhone {
    boolean sendCall();
    boolean receiveCall();
    boolean sendSMS();
    boolean receiveSMS();
}
interface MP3 {
    void play();
    void stop();
}
interface MusicPhone extends MobilePhone, MP3 {
    void playMP3RingTone();
}
```

```
public class MyPhone implements MusicPhone {
    public boolean sendCall() { ……; }
    public boolean receiveCall() { ……; }
    public boolean sendSMS() { ……; }
    public boolean receiveSMS() { ……; }
    public void play() { ……; }
    public void stop() { ……; }
    void playMP3RingTone() { ……; }
}
```

인터페이스 구현

- 구현 클래스 정의
  - 자신의 객체가 인터페이스 타입으로 사용할 수 있음
  - **implements** 키워드 사용
  - 여러 개의 인터페이스 동시 구현 가능
  - 상속과 구현이 동시에 가능

```
public class 클래스명 implements 인터페이스명 {
    //인터페이스에 선언된 추상 메소드의 실체 메소드 구현

}
```

- 추상 메소드의 실체 메소드를 작성하는 방법
  - 메소드의 선언부가 정확히 일치해야 함
  - 인터페이스의 모든 추상 메소드를 재정의하는 실체 메소드를 작성해야 함
    - 일부 추상메소드만 재정의할 경우, 구현 클래스는 추상 클래스

인터페이스 구현 및 사용

```
public class Television implements RemoteControl {
    private int volume; //필드
    //turnOn() 추상 메소드의 구현
    public void turnOn() {
        System.out.println( " TV를 켭니다. " );
    }
    //turnOff() 추상 메소드의 구현
    public void turnOff() {
        System.out.println( " TV를 끕니다. " );
    }
    //setVolume() 추상 메소드의 구현
    public void setVolume(int volume) {
        if(volume>RemoteControl.MAX_VOLUME) {
            this.volume = RemoteControl.MAX_VOLUME;
        } else if(volume<RemoteControl.MIN_VOLUME) {
            this.volume = RemoteControl.MIN_VOLUME;
        } else {
            this.volume = volume;
            }
        System.out.println("현재 TV 볼륨: " + volume);
    }
}
```

```
public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc;
        rc = new Television();      // 구현 객체
        rc = new Audio();           // 구현 객체
    }
}
```

인터페이스 구현 및 사용

```
public class Audio implements RemoteControl {
    private boolean mute; //필드
    @Override // 필요시 디폴트메소드 재정의
    public void setMute(boolean mute) {
        this.mute = mute;
        if(mute) {
            System.out.println("Audio 무음
처리합니다.");
        } else {
            System.out.println("Audio 무음
해제합니다.");
        }
    }
}
```

```
public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc = null;
        rc = new Television();
        rc.turnOn(); // use abstract
method
        rc.setMute(true); // use default
method
        rc = new Audio();
        rc.turnOn(); // use abstract
method
        rc.setMute(true); // use default
method
        RemoteControl.changeBattery();
// use static method
    }
}
```

인터페이스 구현

- **익명 구현 객체**
  - 명시적인 구현 클래스 작성 생략하고 바로 구현 객체를 얻는 방법
    - 이름 없는 구현 클래스 선언과 동시에 객체 생성

```
인터페이스명 변수 = new 인터페이스명() {
    //인터페이스에 선언된 추상 메소드의 실체 메소드 구현

}
```

    - 인터페이스의 추상 메소드들을 모두 재정의하는 실체 메소드가 있어야
    - 추가적으로 필드와 메소드 선언 가능하나 익명 객체 안에서만 사용
      - 인터페이스 변수로 접근 불가

```
public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc = new RemoteControl() { // anonymous class
            public void turnOn() { /*실행문*/ }
            public void turnOff() { /*실행문*/ }
            public void setVolume(int volume) { /*실행문*/ }
        };
    }
}
```

인터페이스의 다중 구현

- **다중 인터페이스(multi-interface) 구현 클래스**
  - 구현 클래스는 다수의 인터페이스를 모두 구현
  - 객체는 다수의 인터페이스 타입으로 사용

```
public class 클래스명 implements 인터페이스명A, 인터페이스명B {
    //인터페이스A에 선언된 추상 메소드의 실체 메소드 구현
    //인터페이스B에 선언된 추상 메소드의 실체 메소드 구현
}
```

인터페이스의 다중 구현

```
interface USBMouseInterface {
  void mouseMove();
  void mouseClick();
}
interface RollMouseInterface {
  void roll();
}
public class MouseDriver implements RollMouseInterface, USBMouseInterface
{
  public void mouseMove() { .... }
  public void mouseClick() { ... }
  public void roll() { ... }

  // 추가적으로 다른 메소드를 작성할 수 있다.
  int getStatus() { ... }
  int getButton() { ... }
}
```

인터페이스 사용

- 인터페이스의 사용
  - 클래스의 **필드(field)**
  - 생성자 또는 메소드의 **매개변수(parameter)**
  - 생성자 또는 메소드의 **로컬 변수(local variable)**

MyClass mc = new MyClass(**new Television()**);

```
public class MyClass {
      // field
      RemoteControl rc = new Television();
      // constructor – parameter
      MyClass(RemoteControl rc) {
            this.rc = rc;
      }
      // method
      public method() {
            // local variable
            RemoteControl rc = new Audio();
      }
}
```

Comparable 인터페이스

- Comparable 인터페이스는 객체의 비교를 위한 인터페이스로 객체 간의 순서나 정렬을 하기 위해서 사용

```
public interface Comparable {
      // 이 객체가 다른 객체보다 크면 1, 같으면 0, 작으면 -1을 반환한다.
      int compareTo(Object other);
}
```

```
class Person implements Comparable {
    public int compareTo(Object other) {
        Person p = (Person)other;
        if (this.age == p.age) return 0;
        else if (this.age > p.age) return 1;
        else   return -1;
    }
}
```

Ex : Comparable 인터페이스

```
public class Rectangle implements Comparable {
  public int width = 0;      public int height = 0;
  @Override
  public String toString() { return "Rect [w=" + width + ", h=" + height + "]"; }
  public Rectangle(int w, int h) { width = w; height = h; System.out.println(this); }
  public int getArea() { return width * height; }
  @Override
  public int compareTo(Object other) {
     Rectangle otherRect = (Rectangle)other;
     if (this.getArea() < otherRect.getArea())
        return -1;
     else if (this.getArea() > otherRect.getArea())
        return 1;
     else
        return 0;
  }
}
```

Ex : Comparable 인터페이스

```
public class RectangleTest {
  public static void main(String[] args) {
  Rectangle r1 = new Rectangle(100, 30);
  Rectangle r2 = new Rectangle(200, 10);
  int result = r1.compareTo(r2);
  if (result == 1)
        System.out.println(r1 + "가 더 큽니다.");
  else if (result == 0)
        System.out.println("같습니다");
  else
        System.out.println(r2 + "가 더 큽니다.");
  }
}
```

Comparator 인터페이스

- Comparator 인터페이스는 다른 두 개의 객체를 비교하기 위한 인터페이스

```
public interface Comparator {
      // o1가 o2보다 크면 1, 같으면 0, 작으면 -1을 반환한다.
      int compare(Object o1, Object o2);
}
```

```
class AgeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Person p1 = (Person)o1;
        Person p2 = (Person)o2;
        if (s1.age == s2.age) return 0;
        else if (s1.age > s2.age) return 1;
        else   return -1;
    }
}
```