Java 용법 마스터 1

객체의 개념과 기본 메소드 익히기

글 | 이해일 <컴포넌트비전(주) 책임 컨설턴트> jongidal@hitel.net

영어 단어를 아는 사람은 많지만 영어를 제대로 구사하는 사람은 드물듯이,
현대 프로그래밍의 기본 언어인 Java를 '유창하게' 쓰는 사람은 의외로 많지 않은 것 같다.
이 글은 Java 언어의 자연스러운 용법 몇 가지를 통해
Java를 '유창하게' 쓸 수 있도록 하기 위해 기획된 것이다.
Java의 실전 적용을 목표로 꼭 익혀야 할 핵심 용법을 다룰 것이다.
이번호에서는 객체의 개념부터 시작해 주요 기본 메소드를 살펴본다.


JAVA
if (in! = null) try {
( ). error
length = = 0
return storries; }
in.close
= = null

Java 할 줄 아세요?

우리가 외국어를 배울 때 꼭 익혀야 하는 세 가지는 문법, 어휘, 용법이다. 언어가 어떤 구조를 가지는지(문법), 구성요소는 무엇인지(어휘), 어떻게 쓰는 것이 자연스러운지(용법)를 익혀야 유창하게 외국어를 구사할 수 있다. 문법과 어휘는 책만 봐도 익힐 수 있지만 용법은 직접 그 나라에 가서 살아 보거나 그 말을 모국어로 쓰는 외국인과 부딪혀 보기 전에는 익히기 어렵다. 프로그래밍 언어도 마찬가지다. 우선 배우려는 프로그래밍 언어의 핵심 문법을 이해해야 한다. 절차형 언어인지, 객체지향 언어인지 알아야 하고 어떤 특징을 가지는지 이해해야 한다. 또, 자료구조, 연산자, 키워드, 표준 라이브러리 같은 프로그래밍 언어 어휘들도 배워야 한다. 여기까지는 책을 읽거나 강의를 듣는 것만으로 쉽게(?) 익힐 수 있다. 하지만, 정확한 용법을 벗어난 외국어가 어색한 것처럼 프로그래밍 언어의 정확한 용법을 모르고서는 자연스럽고 우아한 코드를 만들어 낼 수 없다.

자연스럽고 우아한 코드를 만드는 것은 미학의 문제만은 아니다. 중요한 외교석상에서 어설픈 외국어를 쓴다면 망신을 당하는 차원을 넘어 국가 대사를 그릇칠 수도 있는 것처럼 어설프게 만든 코드는 전체 시스템을 죽일 수도 있다. 특히, 요즘같이 정보 시스템 없이 돌아가는 업무를 상상하기 힘든 상황에서 제대로 프로그래밍 언어를 구사하는 것은 정말 중요한 일이다.

"Java 할 줄 아세요?"라는 질문은 "영어 할 줄 아세요?"라는 질문과 비슷하다. 우리의 "예"라는 대답은 과연 어떤 "예"일까? 하지만, 미리 기죽을 필요는 없다. 우리가 태어났을 때 외국어를 몰랐지만 계속 배우고 익히면 유창하게 외국어를 구사할 수 있는 것처럼 프로그래밍 언어도 배우고 익혀 유창하고 우아하게 구사하면 된다.

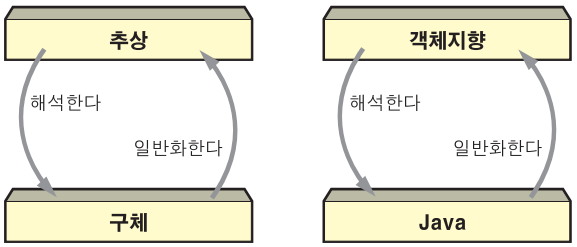
언어는 사고와 서로 상호작용하면서 같이 커가는 것이다. 언어를 통해 표현하면 생각이 자라고 생각이 자라면 표현도 더 우아해진다. 이 기사에서는 Java라는 프로그래밍 언어의 자연스러운 용법을 몇 가지 소개하여 우리가 'Java스럽게' 생각하고 Java를 모국어처럼 유창하게 말하는 데 작은 도움을 주고자 한다.

이번 호에서는 Java 프로그래밍 언어의 용법을 객체라는 일반론 관점에서 살펴보고, 아주 중요한 기본 메소드인 equals, hashCode, copareTo, compare의 정확한 용법을 살펴볼 것이다. 그리고, 다음 호에서는 프로그래밍 일반론으로 불변규칙을 지키는 방법, API 만드는 방법, Java의 중요한 특징 중 하나인 예외처리, 유용한 구현 패턴들을 살펴볼 계획이다.

객체를 이해하면 Java가 쉬워진다

고대로부터 귀납법과 연역법(요즘은 Bottom-up(상향식)과 Top-down(하향식)이라고도 한다)이라는 사물을 이해하는 방법이 전해 내려온다.. 상향식 방법은 구체적인 사실로부터 추상적인 개념을 이해하는 방식이고, 하향식 방법은 추상적인 개념으로부터 구체적인 사실들을 이해하는 것이다. 이 둘은 서로 양립하는 관계가 아니라 서로 보완하는 관계이다. Java와 객체지향의 관계도 마찬가지다. Java 언어를 쓰면서 객체지향 사고를 단단히 할 수 있고 객체지향 사고가 단단해질수록 Java라는 언어를 우아하고 유창하게 말할 수 있는 것이다 <그림 1>.

그러면, Java와 객체지향 개념이 어떻게 서로 영향을 미치는지 살펴보자.

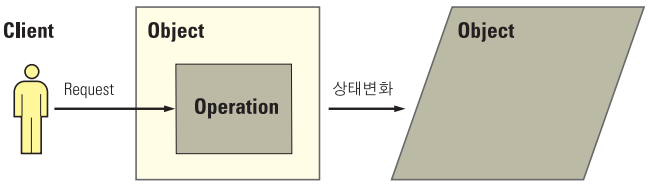


<그림 1> Java와 객체지향의 관계

모든 것은 객체다

객체지향 언어에서 모든 것은 객체다. 객체란 프로그램이 해결해야 하는 문제 공간의 구성요소와 이 문제를 해결하는 해결책들을 추상화한 것이다. 객체지향 언어는 문제를 기계나 컴퓨터의 관점에서 기술하고 해결하는 것이 아니라 문제의 관점에서 기술하고 해결하려고 노력한다. 컴퓨터가 해결해야 하는 복잡한 문제를 우리가 매일 만나는 세상사를 해결하듯이 접근할 수 있다는 것이 객체지향 언어의 가장 큰 장점일 것이다. 그렇다면 객체는 어떻게 동작할까?

객체는 클라이언트로부터 요청을 받았을 때만 자신의 오퍼레이션을 수행할 수 있다. 이렇게 오퍼레이션을 수행해야만 객체 상태가 바뀌고 이런 상태 변화를 통해 객체는 문제를 해결한다<그림 2>.



<그림 2> 객체가 동작하는 원리

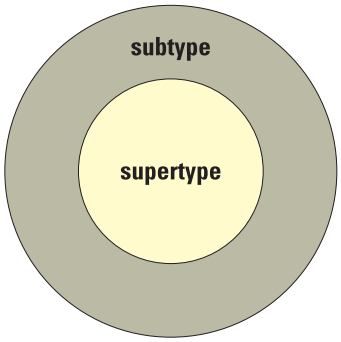
객체의 목적은 자신의 클라이언트에게 서비스를 제공하는 것이다. 객체 혼자서는 아무런 의미가 없다. 한 객체가 받아들일 수 있는 요청이 무엇인지 모아 놓은 것이 바로 인터페이스이다. 클라이언트는 객체가 요청을 받아들여서 어떻게 처리하는지 알고 싶지도 않을 것이고 알 필요도 없을 것이다. 궁금한 것은 인터페이스뿐이다. 객체지향의 핵심은 바로 인터페이스이다.

객체지향의 핵심은 인터페이스이다

어떤 객체가 A 타입이라는 것은 이 객체가 A라는 인터페이스가 받아들일 수 있는 모든 요청을 받아서 서비스를 제공한다는 뜻이다. 한 객체는 여러 타입일 수 있고 전혀 관련이 없는 객체들도 같은 타입일 수 있다.

<그림 3>처럼 어떤 인터페이스가 다른 인터페이스를 포함할 수도 있다. 이 경우 포함된 인터페이스를 '슈퍼타입(supertype)'이라고 하고 포함하는

인터페이스를 '서브타입(subtype)'이라고 한다. 서브타입은 슈퍼타입이 받아들일 수 있는 모든 요청을 받아들일 수 있고 자신만의 요청도 받아들일 수 있다. 이런 포함 관계가 있을 때 서브타입 인터페이스가 슈퍼타입 인터페이스를 상속받는다고 말한다.



<그림 3> 인터페이스의 상속 관계

앞에서도 이야기했듯이 객체는 인터페이스를 통해서만 자신을 외부에 드러낼 수 있다. 외부에서 객체에 대해 알 수 있는 것은 인터페이스밖에 없고 사실 인터페이스만 알면 된다. 이 말은 타입만 같으면 요청을 받아서 처리하는 객체가 어떻게 구현되었든 아무런 상관없다는 뜻이다. 이렇게 클라이언트의 요청을 처리할 객체를 바꿀 수 있는 기법을 '동적 결합(dynamic binding)'이라고 한다.

동적 결합은 타입만 같으면 언제든지 다른 객체로 교체할 수 있는 다형성(polymorphism) 때문에 가능하다. 이 개념은 상당히 중요하다. 인터페이스, 동적 결합, 다형성 때문에 객체지향 언어로 개발한 프로그램이 뛰어난 유연성과 재사용성을 제공할 수 있는 것이다. 이 사실은 요청에도 똑같이 적용된다. 요청을 받아서 처리하는 객체는 요청이 어떻게 구현되었는지 알 필요가 없다. 요청이 제공하는 인터페이스만 알면 된다.

다음 코드를 살펴보자.

```
// Good - 인터페이스로 선언한다.
List subscribers = new Vector();

// Bad - 구현체로 선언한다.
Vector subscribers = new Vector();
```

만약 Vector가 아닌 ArrayList로 구현체를 바꿔야 한다면, 인터페이스로 선언한 경우에는 subscribers를 초기화하는 부분을 다음과 같이 바꿔주기만 하면 나머지 코드는 손 댈 필요가 전혀 없다. 하지만, 구현체로 선언했고 나머지 코드에서 구현체만 제공하는 기능을 쓰고 있다면 구현체를 바꾸려 할 때 많은 부분을 고쳐야 한다.

```
List subscribers = new ArrayList();
```

인자를 정의할 때도 인터페이스로 정의하는 것이 좋다. 예를 들어, 인터페이스가 있는데도 Hashtable과 같은 구현체를 인자로 받는 메소드를 만들면 안 된다. 인터페이스인 Map을 써야 한다. Map을 인자로 받으면 Hashtable, HashMap, TreeMap과 같이 현재 구현된 모든 Map 구현체뿐만 아니라, 앞으로 구현될 모든 Map 구현체들을 인자로 받아 처리할 수 있다. Hashtable로 인자를 정의했다면 이 오퍼레이션을 호출할 때 Hashtable이 아닌 Map 객체는 모두 Hashtable로 바꿔야 할 것이다. 이것은 필요하지도 않고 오류가 발생하기도 쉽다. 이렇게 구현체가 아닌 인터페이스로 프로그램을 만드는 것은 객체지향 프로그래밍의 중요한 원칙이다.

그렇다면 클래스는 무엇인가? 클래스는 객체의 구현을 정의한 것이다. 한 객체는 여러 타입일 수 있고 여러 클래스로부터 생성된 객체라도 같은 타입일 수 있지만, 한 객체의 클래스는 단 하나이다. 클래스는 객체의 정적인 모습을 표현한다.

그런데, 객체지향 세계를 객체 사이의 관계가 아닌 클래스 사이의 관계로 이해하려는 오류를 범하는 경우가 많다. 원래 시간과 상황에 따라 바뀌는 것보다는 고정된 것을 이해하는 것이 더 쉽고, 객체가 클래스로부터 생성되기 때문에 이런 오류를 범하는 것 같다. 하지만, 세상의 모든 사물이 시간과 상황에 따라 상태가 변하는 것처럼 시스템의 상태도 계속 변한다. 이렇게 계속 변하는 상태와 관계를 고정된 클래스로 파악하기는 힘들다. 또, 인터페이스, 동적 결합, 다형성으로 실제로 동작하는 객체가 실행 시점에 바뀔 수 있기 때문에 항상 어떤 객체가 어떤 상태에 있는지 파악해야 하고 객체들 사이의 관계도 어떻게 변하고 있는지 파악해야 시스템을 이해할 수 있다.

클래스 상속과 인터페이스 상속은 구분해야 한다. 클래스 상속은 코드 재사용이 목적이지만 인터페이스 상속은 다른 객체로 대체하는 것이 목적이다. 많은 언어가 문법 차원에서 클래스와 인터페이스를 구분하지 않아서 혼란스러울 수 있지만, 이 개념만은 확실히 구분해야 한다(Java에는 'interface'라는 키워드가 있어서 인터페이스와 클래스를 구분할 수는 있지만 클래스로도 타입을 정의할 수 있기 때문에 역시 혼동이 생길 수 있다. 언어 자체가 인터페이스와 클래스를 구분하는 것을 강제하지는 못하지만 타입을 정의할 때는 인터페이스를 쓰는 것이 좋다는 개념을 꼭 명심하고 가능하면 인터페이스로 타입을 정의해야 한다.)

클래스 상속은 코드를 재사용하는 것이기 때문에 부모 클래스가 어떻게 구현되었는지 자세하게 알아야 하고 부모 클래스의 구현이 변하면 바로 이 부모 클래스를 상속받은 모든 서브클래스들이 영향을 받는다. 이런 구현 종속성은 인터페이스 상속에선 나타나지 않는다. 인터페이스에는 아무런 구현이 없기 때문이다. <그림 3>에서 보았듯이 서브타입은 슈퍼타입을 단순히 포함한다.

그렇다면, 안전하게 코드를 재사용할 방법은 없는 것일까? 다행히도 '컴포지션'이라는 방법이 있다(많은 책이나 교육과정에서 이 방법을 가르치지 않아 안타깝다.) 컴포지션 방법은 기존 클래스를 상속받아 새로운 클래스를 만드는 것이 아니라 재사용하려는 객체에 대한 참조를 'private' 필드로 갖는 새로운 클래스를 만드는 것이다. 재사용하려는 기존 클래스가 새로운 클래스의 한 구성요소가 되므로 이런 방식을 '컴포지션(composition)'이라고 한

다. 컴포지션에서는 새로운 클래스가 기존 클래스의 인터페이스만 알면 되지 자세한 구현 방식을 알 필요는 없다. 이것은 서브타입이 단순히 슈퍼타입의 인터페이스를 포함하는 인터페이스 구현과 비슷한 개념이다.

새로운 클래스의 객체는 전달받은 요청을 자신이 포함하고 있는 객체에게 위임한다. 이런 방식을 '포워딩(forwarding)'이라 하고 새로운 클래스의 메소드를 '포워딩 메소드(forwarding method)'라고 한다. 이렇게 만든 클래스는 기존 클래스의 세부 구현 방법에 의존하지 않기 때문에 기존 클래스의 구현이 바뀐다 해도 영향을 받지 않는다. 구성요소를 몇 개나 저장했는지 알 수 있는 Set를 만들 때 HashSet와 같은 기존 Set 구현체를 상속받는 것이 아니라 다음과 같이 컴포지션과 포워딩 메소드를 쓰는 것이 좋다. 이 때, 재사용하려는 클래스가 인터페이스를 구현하고 있다면 새로 만드는 클래스도 같은 인터페이스를 구현하는 것이 좋다.

```
public class InstrumentedSet implements Set {
    // 선언은 항상 interface로.
    // 재사용하려는 Set 객체
    private final Set s;
    private int addCount = 0;

    public InstrumentedSet(Set s) {
        this.s = s;
    }

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }

    // 포워딩 메소드들
    public void clear() { s.clear(); }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty() { return s.isEmpty(); }
    public int size() { return s.size(); }
    public Iterator iterator() { return s.iterator(); }
    public boolean remove(Object o) { return s.remove(o); }
    public boolean containsAll(Collection c) { return s.containsAll(c); }
    public boolean removeAll(Collection c) { return s.removeAll(c); }
    public boolean retainAll(Collection c) { return s.retainAll(c); }
    public Object[] toArray() { return s.toArray(); }
    public Object[] toArray(Object[] a) { return s.toArray(a); }
    public boolean equals(Object o) { return s.equals(o); }
    public int hashCode() { return s.hashCode(); }
    public String toString() { return s.toString(); }
}
```

어떻게 객체를 생성할 것인가?

객체지향 세계에서 가장 중요한 객체는 어떻게 생성하는 것이 좋을까? 상속보다는 컴포지션이 권장되면서 객체를 생성하는 방법은 더욱 더 중요해지고 있다.

객체를 생성하는 가장 흔한 방법은 public 생성자를 쓰는 것이다. 이 방법을 쓰면 반드시 생성하려는 객체의 클래스 이름이 명시적으로 나오게 되므로 객체 생성이 특정 구현에 완전히 얽매이게 된다. 따라서 이렇게 객체를 생성하면 나중에 다른 구현체로 바꾸려 할 때 문제가 생긴다. 또, 싱글톤(singleton)처럼 객체 수를 제한할 필요가 있다면 public 생성자로 객체를 생성하지 못하게 막아야 한다.

객체 생성은 어떻게 제어할까? 우선 팩토리 메소드라는 방법을 살펴보자. 팩토리 메소드는 단순하게 객체 생성을 책임지는 메소드이다. 이 메소드는 생성하는 객체를 정의한 클래스의 스태틱 메소드일 수도 있고 객체 생성 전담 클래스에 정의된 메소드일 수도 있다. 팩토리 메소드를 쓰면 public 생성자보다 다음과 같은 몇 가지 좋은 점이 있다.

생성자는 클래스와 같은 이름만 가질 수 있지만 팩토리 메소드는 의미있는 이름을 가질 수 있어서 코드를 이해하기 쉬워진다. 예를 들어 소수(素數)일 가능성이 큰 BigInteger 객체를 생성할 때, 생성자인 BigInteger(int, int, Random)보다 팩토리 메소드인 BigInteger.probablePrime를 쓰는 쪽이 훨씬 더 이해하기 쉽다. 또, 생성자를 중복 정의할 때 어쩔 수 없이 같은 시그니처를 가질 수밖에 없어 곤란한 경우가 있지만, 팩토리 메소드를 쓰면 이름만 바꾸면 된다.

팩토리 메소드를 쓰면 특정 시점에 존재하는 객체의 수를 엄격하게 관리할 수 있다. 싱글톤처럼 객체 수가 제한되거나 내용이 동등한 불변 클래스의 객체가 단 하나만 존재하게 하려면, 팩토리 메소드로 객체 생성을 제어해야 한다. 예를 들어, Boolean에는 기본타입 boolean 값을 받아 이에 해당하는 Boolean 객체를 만들어 내는 valueOf라는 팩토리 메소드가 있다. Boolean은 true아니면 false에 해당하는 객체 두 개만 있으면 된다. 따라서, public 생성자로 새로운 객체를 매번 생성할 필요가 없이 객체 두 개를 미리 만들어 놓고 필요할 때마다 이 객체를 제공하면 된다.

```
public static Boolean valueOf(boolean b) {
```

```
        return (b ? Boolean.TRUE : Boolean.FALSE);
    }
```

생성자는 반드시 자신을 정의한 클래스의 객체를 생성해야 하지만, 팩토리 메소드는 리턴타입과 그 하위타입에 해당하는 어떤 객체라도 생성할 수 있다. 다시 말하면, 타입만 맞으면 언제라도 구현체를 마음대로 바꿀 수 있다. 이것은 아주 중요한 특성으로, 팩토리 메소드를 쓰는 가장 큰 이유라고 할 수 있다. 예를 들어, 컬렉션 프레임워크에는 수정할 수 없는 컬렉션(unmodifiable collection), 동기화 컬렉션(synchronized collection)과 같은 20여 개의 편리한 구현 클래스들이 있다. 이 클래스들의 객체는 모두 java.util.Collections에 있는 팩토리 메소드로만 얻을 수 있다. 만약 이 20개 클래스가 모두 외부에 드러났다면 컬렉션 프레임워크는 지금보다 훨씬 복잡했을 것이다. 다행히 팩토리 메소드가 이런 복잡성을 감췄기 때문에 오로지 인터페이스만 알면 모든 구현 클래스를 마음대로 쓸 수 있다. 또, 팩토리 메소드를 쓰면 클라이언트가 리턴받은 객체를 실제 구현 클래스 타입이 아닌 인터페이스 타입으로만 참조하도록 강제할 수 있다는 장점도 있다.

클래스를 만들 때 습관처럼 public 생성자를 만들지 말고 자유롭게 객체가 생성될 필요가 있는지 다시 한 번 생각해 보라. 객체 생성을 제어하고 싶다면 팩토리 메소드를 쓰는 것이 좋다.

모든 생성자를 private로 만들어 외부에서 해당 클래스의 객체를 만들지 못하게 막아야 하는 경우가 있다. 상태가 없는 유틸리티 클래스(java.util.Collections, java.util.Arrays 같은 것들)는 객체를 만들 필요가 없다. 따라서, 이런 클래스의 생성자는 모두 private이고 메소드는 모두 static이다.

싱글톤은 정확히 하나의 객체만 존재하는 클래스로 생성자를 private으로 정의하고 클라이언트가 이 클래스의 유일한 인스턴스에 접근할 수 있도록 public static 필드나 메소드를 제공하는 방식으로 구현한다. 우선 public static 필드를 쓰는 방식을 살펴보자.

```
public class JNDIService implements IService
{
    ...
    public static final JNDIService SINGLETON = new JNDIService();

    private JNDIService() {
        super();
        caches = new HashMap();
    }
    ...
}
```

JNDIService의 private 생성자는 이 클래스가 로딩되는 시점에 단 한 번 호출된다. 클라이언트는 접근할 수 있는 JNDIService 생성자가 없기 때문에 더 이상 이 클래스의 객체를 만들 수 없다. JNDIService 객체는 SINGLETON이 참조하는 객체 하나만 존재한다.

다음으로 public static 메소드를 쓰는 방식을 살펴보자.

```
public class JNDIService implements IService
{
    ...
    private static JNDIService singleton = null;

    private JNDIService() {
        super();
        caches = new HashMap();
    }

    public synchronized static JNDIService getInstance() {
        if (singleton == null) {
            singleton = new JNDIService();
        }
            return singleton;
    }
    ...
}
```

public static 필드를 쓰면 자동으로 JNDIService 객체가 하나 생성된다. 만약 싱글톤을 생성하는 비용이 크다면 이것은 낭비다. 이런 경우에는 public static 메소드 방식을 써서 늦은 초기화(lazily initialization)를 해야 한다. 하지만, 다중 스레드 환경에서 싱글톤을 보장하려면 앞의 코드처럼 반드시 동기화해야 한다. 늦은 초기화도 필요하고 동기화도 피해야 한다면 다음과 같이 보유자 클래스(Initiate-on-demand holder class) 구현 패턴을 써야 한다.

```
public class JNDIService implements IService
{
    ...
    // 보유자 클래스
    private static class SingletonHolder {
        static final JNDIService SINGLETON = new JNDIService();
```

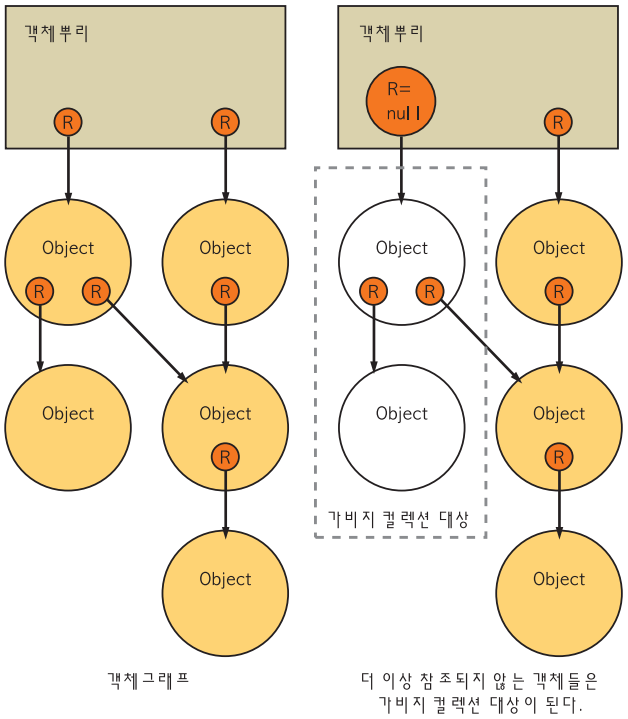
```
    }

    private JNDIService() {
        super();
        caches = new HashMap();
        ...
    }

    // 동기화도 필요 없고 비교도 필요 없다.
    public static JNDIService getInstance() {
            return SingletonHolder.SINGLETON;
    }
    ...
}
```

이 구현 패턴은 모든 클래스의 초기화는 이 클래스가 처음으로 쓰이는 순간 이루어진다는 사실에 기초한 것이다. getInstance 메소드를 호출하여 SingletonHolder 클래스의 SINGLETON 필드를 처음으로 읽는 순간 SingletonHolder 클래스가 초기화된다. 따라서, 동기화가 필요 없고 비교도 필요 없다. 아무런 추가비용 없이 늦은 초기화의 장점까지 제공할 수 있는 아주 유용한 구현 패턴이다.

객체를 생성할 때 상황에 따라 여러 가지 전략이 필요하다. 무심코 public 생성자를 만들고 있는 자신을 한 번 돌아보라. 객체 생성은 많은 비용이 들어가는 작업일 수도 있고 시스템의 유연성에 큰 영향을 미치는 작업일 수도 있기 때문에 신중해야 한다.

어떻게 객체를 파괴할 것인가?

Java 플랫폼의 모든 배열과 객체들은 '힙(heap)'이라는 메모리 공간에 저장된다. new 키워드를 쓸 때마다 힙의 새로운 메모리가 객체에 할당된다. 하지만, Java는 C++ 같은 언어와 달리 할당한 메모리를 명시적으로 반환하는 방법이 없다. 사실, 이 작업은 가비지 컬렉터가 담당한다. 가비지 컬렉터는 아주 낮은 우선순위를 가진 백그라운드 스레드로 동작하면서 어떤 객체의 메모리를 반환해야 하는지 계속 검사한다. 만약, 메모리를 반환해야 하는 객체를 찾았고 시간도 충분하다면 가비지 컬렉터는 종료자를 수행하는 것과 같이 몇 가지 필요한 작업을 처리하고 객체를 파괴한 다음에 이 객체의 메모리를 힙으로 반환한다. 프로그래머가 무엇을 하든 가비지 컬렉터에게 이 작업을 강제로 시킬 수는 없다. 하지만, 객체 참조에 null을 대입하고 System.gc()를 호출하면 <그림 4>처럼 객체 그래프에서 더 이상 참조되지 않는 객체들이 생기고 JVM이 한가하다면 가비지 컬렉션이 일어나면서(반드시 가비지 컬렉션이 일어난다는 것을 보장할 수는 없다.) 더 이상 참조되지 않는 객체들을 파괴한다(실제로 이것보다 훨씬 복잡한 알고리즘을 쓰고 있지만, 이 정도만 알아도 충분할 것이다.)



<그림 4> 가비지 컬렉션 대상을 고르는 방법

이런 방식으로 메모리를 할당하고 반환하면 아주 안전하기는 하지만 프로그래머가 메모리를 직접 관리할 수 없어서 불편할 수 있다. 이 문제를 해결하기 위해 JDK 1.2 배포판부터 java.lang.ref 패키지를 제공해서 프로그램에서 가비지 컬렉터에 접근할 수 있는 방법을 제공하고 있다. 자세한 것은 Java 명세 문서를 참조하기 바란다.

그런데, Java에서는 가비지 컬렉터가 객체 파괴를 처리해 주므로 메모리에 대해 신경 쓰지 않아도 될 것 같지만, 몇 가지 주의할 점이 있다.

다음과 같이 구현한 간단한 스택을 한 번 살펴보자.

```
public class stack {
    private object [] elements;
    private int size = 0;
    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
```

```
    public Object pop() {
        if(size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    private void ensureCapacity() {
        if (elements.length == size) {
            Object[] oldElements = elements;
            elements = new Object[2*elements.length+1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

이 스택은 별 문제가 없어 보이지만 스택에서 팝(pop)된 객체들에 대한 참조를 스택이 계속 쥐고 있기 때문에 이 객체들은 가비지 컬렉션 대상이 되지 않고 끝까지 남는다. 가비지 컬렉터가 있는 언어에서도 메모리 누수 현상('의도하지 않은 객체 유지(unintentional object retention)'가 더 적절한 표현이다.)이 일어날 수 있다! 이런 문제는 간단하게 해결할 수 있다. 쓸모 없어진 참조에 다음과 같이 null을 대입해 버리면 된다.

```
public Object pop() {
    if(size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null;    // 쓸모 없는 참조를 없앤다.
    return result;
}
```

Stack처럼 자신만의 메모리 공간을 가지는 클래스는 언제나 메모리 누수가 일어날 가능성이 있다. 어떤 객체를 보관할 필요가 없는지는 프로그래머만 알 수 있다. 따라서, 프로그래머가 직접 필요 없는 객체 참조를 null로 만들지 않으면 가비지 컬렉터는 어떤 객체를 가비지 컬렉션할지 알 수가 없다. 따라서, 자신만의 메모리 공간을 가지는 클래스를 쓸 때 보관할 필요가 없는 객체가 생기면 이 객체에 대한 참조 변수에 null을 대입하여 가비지 컬렉션 대상으로 만들어야 한다.

객체 파괴에서 또 하나 주의할 점은, finalize, System.runFinalization, System.runFinalizersOnExit, Runtime.runFinalizersOnExit와 같은 종료자는 쓰지 않는 것이 좋다는 것이다. 종료자들은 정확히 실행된다는 보장도 없고 종료자 메소드에서 처리하지 않는 예외가 발생하면 예외도 무시되고 종료자도 끝나버린다. 이런 여러가지 문제 때문에 종료자에 의존하는 코드는 되도록 만들지 않도록 한다.

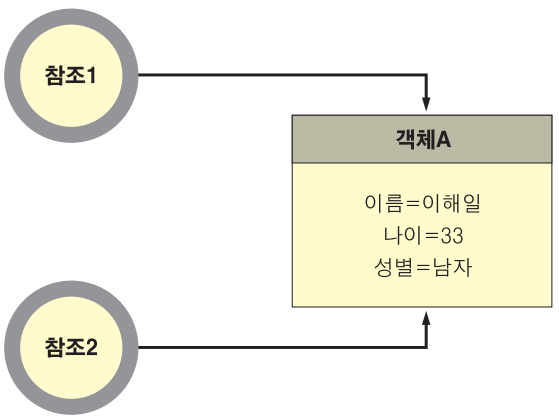
이 메소드들은 꼭 알아두자

Java의 모든 클래스는 java.lang.Object를 상속받는다. java.lang.Object에는 equals, hashCode, clone, toString, finalize와 같이 하위 클래스에서 재정의할 수 있는 final이 아닌 public이나 protected 메소드들이 있다. 이 메소드들을 재정의할 때 꼭 지켜야 하는 규칙들이 정해져 있다. 여러분이 만든 클래스가 이 메소드들을 재정의하면서 규칙을 지키지 않는다면 커다란 혼란이 일어난다. 이런 혼란을 막기 위해서 간단하지만 아주 중요한 java.lang.Object의 기본 메소드인 equals와 hashCode를 정확히 구현하는 방법을 살펴보자. 또, java.lang.Object의 메소드는 아니지만 꼭 알아두어야 할 Comparable과 Comparator 인터페이스에 있는 compareTo 메소드와 compare 메소드도 살펴보겠다.

equals 메소드

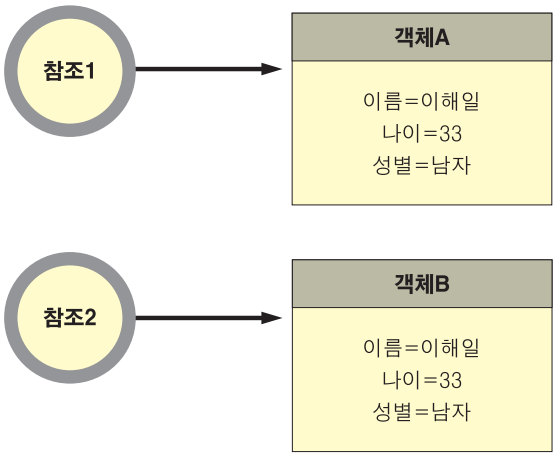
equals 메소드를 살펴보기 전에 우선 '같다'라는 것을 구분해 보자. same, equal, equivalent, identical은 모두 우리말로 '같다'로 바꿀 수 있지만 이것들을 구분할 필요가 있다.

<그림 5>에서 참조 1과 참조 2는 '같은' 객체를 참조하고 있다. 이 경우 '같은' 객체를 참조한다는 것은 한 메모리 주소에 있는 '동일한(identical)' 객체를 참조한다는 뜻이다. 이 경우 참조1 == 참조2의 결과는 true이다. 당연히 두 참조가 가리키는 객체의 내용은 같다.



<그림 5> 동일한 객체. 참조 1 == 참조 2, 참조 1.equals(참조 2)는 모두 true

그러나, <그림 6>에서 참조 1과 참조 2는 동일하지 않은 객체를 참조하고 있다. 다시 말하면, 참조 1 == 참조 2의 결과는 false이다. 하지만, 참조 1과 참조 2가 가리키는 두 객체의 상태는 같다(필드에 저장된 값이 객체 상태를 결정한다.) 이 때 '같다'는 것은 두 객체의 내용이 '동등하다(equivalent)'는 뜻이다. equals 메소드는 이런 내용이 동등한지 검사하여 같다고 판단하면 true를 리턴해야 한다.



<그림 6> 동등한 객체. 참조1 == 참조2는 false, 참조1.equals(참조2)는 true

하지만, java.lang.Object의 equals 메소드는 다음과 같다. 따라서, equals 메소드를 재정의하지 않으면 객체 내용이 동등한지 검사할 수 없다.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

따라서, 객체 내용의 동등성이 중요하지 않은 몇가지 경우(예를 들면, ◉Thread처럼 객체 내용이라는 것이 아예 없는 경우 ◉Random처럼 객체 내용은 있지만 의미가 없는 경우 ◉외부에 드러나지 않는 객체라서 equals 메소드가 호출되는 일이 없는 경우 ◉불변객체나 싱글톤처럼 내용이 같은 객체가 중복으로 존재할 수 없는 경우 ◉상위 클래스에서 이미 적절한 equals 메소드를 정의해 놓은 경우)가 아니라면 항상 equals 메소드를 재정의해야 한다. equals 메소드는 아주 간단해 보이지만 지켜야 하는 구현 계약은 생각보다 복잡하고 조심하지 않으면 문제가 생기기 쉽다. equals 메소드의 구현 계약은 다음과 같다*(java.lang.Object의 equals 메소드 명세)*

equals 메소드는 동등 관계(equivalence relation)를 구현한다.

1. 반사적(reflexive)이다 : 모든 참조 값 x에 대해 x.equals(x)는 true를 리턴해야 한다.
2. 대칭적(symmetric)이다 : 모든 참조 값 x와 y에 대해 y.equals(x)가 true를 리턴할 때만 x.equals(y)는 true를 리턴해야 한다.
3. 추이적(transitive)이다 : 모든 참조 값 x, y, z에 대해 만약 x.equals(y)와

y.equals(z)가 true를 리턴한다면 x.equals(z)도 true를 리턴해야 한다.

4. 일관적(consistent)이다 : 모든 참조 값 x,y에 대해, 만약, equals 메소드가 비교할 때 쓰는 정보가 변하지 않는다면 x.equals(y)의 결과는 항상 일관성이 있어야 한다. 즉, 한 번 true면 계속 true를, 한 번 false면 계속 false를 리턴해야 한다.
5. null이 아닌 모든 참조 x에 대해, x.equls(null)는 반드시 false를 리턴한다.

이 구현 계약들 중에서 2번, 3번은 특히 어기기 쉽다. 대/소문자를 구분하지 않는 문자열을 표현하는 클래스인 CaseInsensitiveString를 예로 들어 대칭성을 어기는 경우를 살펴보자(참고로, 여기서 다루는 예제는 Joshua Bloch의 *[Effective Java Programming Language Guide]*(Addison-Wesley)에 나온 것을 인용했음을 밝혀 둔다.)

```
public final class CaseInsensitiveString {
    ...
    public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(((CaseInsensitiveString)o).s);
        // 대칭성에 문제가 생긴다.
        if (o instanceof String)
            return s.equalsIgnoreCase((String)o);
        return false;
    }
    ... // 이하 생략
}
```

위 코드에서는 대/소문자를 구분하지 않는 문자열과 보통 문자열(String 객체)의 동등성을 비교하고 있다. 하지만 String의 equals 메소드는 대/소문자를 구분하지 않는 문자열을 처리할 수 없기 때문에 대칭성이 깨진다. 다음과 같이 대/소문자를 구분하는 문자열 객체와 일반 문자열 객체가 있다고 하자.

```
CaseInsensitiveString cis = new CaseInsensitiveString("KoReA");
String s = "Korea";
```

예상한 대로 cis.equals 메소드는 true를 리턴하지만 String.equals 메소드는 false를 리턴한다. 이것이 왜 문제일까? CaseInsensitiveString 객체를 컬렉션에 넣어 보자.

```
List list = new ArrayList();
list.add(new CaseInsensitiveString("KoReA"));
```

list.contains("Korea")의 결과는 무엇일까? 이 결과는 정해져 있지 않다. 심지어 예외가 발생할 수도 있다. 이렇게 어떤 클래스가 구현 계약을 어기면 다른 클래스가 예측할 수 없는 행동을 한다. CaseInsensitiveString의 equals 메소드가 String까지 비교하려고 한 것이 문제이다. 다른 타입의 객체와 동등성을 비교하려 하지 말아야 한다.

그렇다면, 추이성은 언제 깨질까? 2차원 평면 위의 한 점을 표현하는 간단한 클래스를 만들어 보자.

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }
    ... // 이하 생략
}
```

이 클래스를 상속받아 '색깔' 정보를 추가해 보자.

```
public class ColorPoint extends Point {
    private Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    ... // 이하 생략
}
```

이 클래스는 equals 메소드를 재정의하지 않았기 때문에 Point 클래스의 equals 메소드를 그대로 쓴다. ColorPoint가 색깔을 비교하지 않는 것은 분명히 이상하기 때문에 다음과 같이 위치도 같고 색깔도 같을 때만 true를 리턴하도록 equals 메소드를 재정의해 보자.

```
public boolean equals(Object o) {
```

```
        if (!(o instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint)o;
        return super.equals(o) && cp.color == color;
    }
```

이번에는 Point의 equals 메소드가 ColorPoint 객체를 받아들여 비교할 수 있기 때문에 대칭성이 깨진다. ColorPoint의 equals 메소드는 항상 false를 리턴하고 Point의 equals 메소드는 위치만 같다면 true를 리턴한다.

그렇다면, 다음과 같이 ColorPoint.equals가 Point 객체도 비교할 수 있게 고치면 되지 않을까?

```
public boolean equals(Object o) {
    if(!(o instanceof Point))
        return false;
    // o가 그냥 Point일 때는 색깔은 비교하지 않고 위치만 비교한다.
    // 추이성이 깨지는 부분이다.
    if (!(o instanceof ColorPoint))
        return o.equals(this);
    // o가 ColorPoint인 경우 색깔과 위치를 모두 비교한다.
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
}
```

안타깝게도 이번에는 추이성이 깨진다. 다음과 같이 Point의 인스턴스와 ColorPoint의 인스턴스를 만들어 이 사실을 확인해 보자.

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

p1.equals(p2)와 p2.equals(p3)는 true를 리턴하지만 p1.equals(p3)는 false를 리턴한다. p1.equals(p2)와 p2.equals(p3)는 색깔을 뺀 위치만 비교하고 p1.equals(p3)는 색깔과 위치를 모두 비교하기 때문에 이런 문제가 생긴 것이다. 그렇다면 도대체 이 문제는 어떻게 해결해야 할까? 사실 이런 문제는 객체지향 언어에서 나타나는 객체 동등성에 대한 근본 문제다. 객체를 만들 수 있는 클래스를 상속받아 새로운 필드를 추가하면서 표준 구현 계약을 준수하는 equals 메소드를 만들 수 없다. 이 문제는 정면 돌파가 거의 불가능하기 때문에 피해가야만 한다. 상속 대신 컴포지션을 써야 한다.

```
// equals 구현 계약을 지키면서 새로운 부분을 추가하기.
public class ColorPoint {
    // Point 객체와 컴포지션 관계를 가진다.
    private Point point;
    private Color color;

    public ColorPoint(int x, int y, Color color) {
        point = new Point(x, y);
        this.color = color;
    }

    /**
     * ColorPoint 인스턴스의 Point로서의 모습을 리턴한다.
     */
    public Point asPoint() {
        return point;
    }

    // ColorPoint 객체만 비교한다.
        public boolean equals(Object o) {
        if(!(o instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint)o;
        return cp.point.equals(point) && cp.color.equals(color);
    }
    ... // 이하 생략
}
```

ColorPoint는 Point 타입이 아니기 때문에 Point의 equals 메소드에 ColorPoint 객체가 전달되면 항상 false를 리턴한다. ColorPoint와 Point의 상속관계를 끊음으로써 ColorPoint와 Point의 equals 메소드는 정확한 타입의 객체를 비교할 수 있다. 정확한 타입의 객체를 비교하게 하는 것이 핵심이다. 또, ColorPoint가 Point로 어떻게 표현되는지 알아야 한다면 asPoint와 같은 뷰 메소드를 제공하면 된다.

이 모든 구현 계약을 지키는 equals 메소드를 만드는 비법은 다음과 같다(이 비법은 Joshua Bloch의 *[Effective Java Programming Language Guide]*(Addison-Wesley)에 나온 것을 인용했다.)

1. 연산자를 써서 인자가 this 객체를 참조하는지 검사한다. 만약 그렇다면, true를 리턴한다. 이것은 성능을 최적화하기 위해 수행하는 작업이다. 비교 작업이 복잡하다면 이 검사를 하는 것이 좋다.
2. instanceof 연산자를 써서 인자의 타입이 올바른지 검사한다. 만약, 타입이 틀리다면 false를 리턴한다. 이때, null이 인자로 넘어오면 instanceof 연산자는 항상 false를 리턴하므로 따로 null 검사를 하지 않아도 된다. 보통, 올바른 타입은 호출하는 equals 메소드를 정의한 클래스 타입이다. 하지만, 이 클래스가 인터페이스를 구현한다면 이 인터페이스도 올바른 타입이 될 수 있

다. 즉, 같은 인터페이스를 구현한 클래스들은 서로 비교할 수 있다. 컬렉션 프레임워크의 Set, List, Map, Map.Entry와 같은 인터페이스를 구현한 클래스들은 이 인터페이스의 타입인지 비교한다.

3. 인자를 정확한 타입으로 변환한다. 이 타입 변환은 이미 instanceof로 타입을 검사했기 때문에 항상 성공한다.
4. 주요 필드(significant field)에 대해 인자의 필드와 this 객체의 해당 필드의 값이 동등한지 검사한다. 모든 필드가 동등하다면 true를 리턴하고 하나라도 동등하지 않다면 false를 리턴한다. 해당 필드가 float나 double이 아닌 기본 타입이라면 == 연산자로 비교한다. Float와 double 타입은 각각 Float.floatToIntBits 메소드와 Double.doubleToLongBits 메소드로 int와 long 값으로 변환한 다음 == 연산자로 비교한다. 객체 참조 필드의 경우에는 그 객체의 equals 메소드로 비교한다. 배열 필드의 경우 모든 각 구성요소에 대해 지금까지 설명한 작업을 수행한다. null에 대한 참조가 허용된 객체 참조 필드에 대한 비교는 NullPointerException을 막기 위해 다음과 같은 구현 패턴을 써서 비교한다.

```
(field == null ? o.field == null : field.equals(o.field))
```

만약, this 객체의 필드와 인자가 가리키는 객체의 필드가 동일한(identical) 객체를 참조하는 경우라면 다음과 같이 비교하는 것이 더 빠르다.

```
(field == o.field || (field != null && field.equals(o.field)))
```

앞에서 살펴본 CaseInsensitiveString과 같은 클래스는 단순히 필드의 동등성을 검사하는 것만으로 equals 메소드를 구현할 수 없다. 이런 경우, 무엇을 어떻게 비교하고 검사하는지가 클래스 명세에서부터 확실하게 드러나야 한다. 이 경우 표준 형식(canonical form)을 정해 놓고 여기에 각 객체의 정보를 저장한 다음, equals 메소드에서 이 표준 형식을 써서 정확하고 빠른 비교작업을 수행하게 만들 수도 있다. 그러나, 객체의 내용이 변할 때마다 표준형식에 저장한 내용도 계속 바꿔 주어야 하기 때문에 표준형식은 불변 클래스에 쓰는 것이 가장 좋다.

이 비법에 따라 전화번호를 표현하는 PhoneNumber 클래스의 equals 메소드를 다음과 같이 구현할 수 있다.

```
public final class PhoneNumber {
    private final short areaCode;
    private final short exchange;
    private final short extension;

    public PhoneNumber(int areaCode, int exchange, int extension) {
        rangeCheck(areaCode, 999, "area code");
        rangeCheck(exchange, 999, "exchange");
        rangeCheck(extension, 9999, "extension");
        this.areaCode = (short) areaCode;
        this.exchange = (short) exchange;
        this.extension = (short) extension;
    }

    private static void rangeCheck(int arg, int max, String name) {
        if (arg < 0 || arg > max)
            throw new IllegalArgumentException(name + ":" + arg);
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;

        // extention, exchange, areaCode 필드가 equals의 비교 대상이다.
```

표준 형식에 대하여

java.math.BigDecimal의 경우를 예로 들어 보자. 다음과 같이 BigDecimal 인스턴스를 생성해 보자.

```
BigDecimal n = new BigDecimal ( "-12345.6789" )
```

이 때, BigDecimal 내부에서는 -12345.6789라는 수를 어떻게 저장하고 있을까? BigDecimal의 소스를 보면 다음과 같은 필드를 정의하고 있다.

```
private BigInteger intVal;

private int scale = 0;
```

BigDecimal은 -12345.6789라는 숫자를 그대로 보관하는 것이 아니라, 이 수의 숫자 부분(-123456789)은 BigInteger intValue 필드에, 10의 제곱수 부분(-4)은 int scale 필드에 보관한다. -12345.6789를 -123456789와 -4로 나누어 $-123456789 \times 10^{-4}$로 표현하는 것이 10진수를 표현하는 표준 형식(canonical form)이다. 이렇게 표준 형식으로 저장하면 어떤 숫자라도 정확하게 표현할 수 있다.

```
            return pn.extension == extension && pn.exchange == exchange &&
                    pn.areaCode == areaCode;
        }
    ...
    }
```

equals 메소드를 제대로 만드는 것은 생각보다 어렵고 중요하다. equals 메소드의 구현 계약을 지키지 않으면 이 클래스를 쓰는 다른 클래스들이 문제를 일으킨다는 점과 equals 메소드 구현 계약을 지키려면 자기와 같은 타입만 비교해야 한다는 것은 꼭 기억해야 한다.

hashCode 메소드

hashCode 메소드의 구현 계약은 다음과 같다.*(java.lang.Object의 hashCode 메소드 명세)*

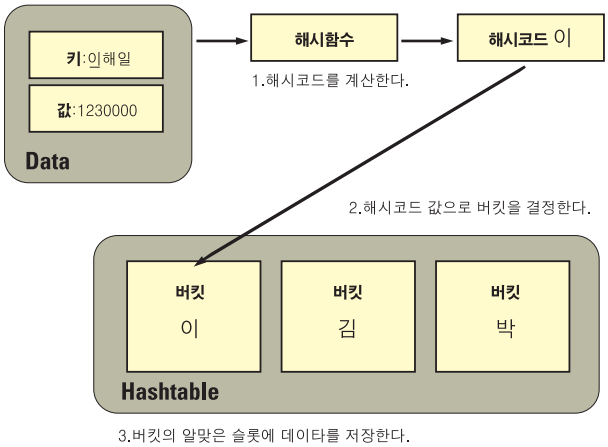
1. 애플리케이션이 일단 실행된 다음 동일한 객체의 hashCode를 여러 번 호출하더라도 equals 메소드에서 비교하는 필드의 값을 바꾸지 않는다면, 항상 같은 정수 값을 리턴해야 한다. 하지만, 이 정수 값은 애플리케이션이 다시 실행되면 바뀔 수도 있다.
2. equals(Object) 메소드의 리턴 값이 true인 두 객체의 hashCode 메소드는 같은 정수 값을 리턴해야 한다.
3. equals(Object) 메소드의 리턴 값이 flase인 두 객체의 hashCode 메소드가 반드시 다른 정수 값을 리턴할 필요는 없다. 하지만, 해시(hash) 알고리즘을 쓰는 컬렉션의 성능을 향상시키려면 동등하지 않은 객체는 다른 정수 값을 리턴하는 것이 좋다.

이 조항들 중에서 특히 두 번째 조항을 어기기 쉽다. equals 메소드를 재정의했으면 꼭 hashCode 메소드를 재정의해야 한다. 그렇지 않으면 해시 알고리즘을 기반으로 동작하는 HashMap, HashSet, Hashtable과 같은 컬렉션에서 문제가 생긴다. 문제가 생기는 이유를 알려면 해시가 동작하는 원리를 이해해야 한다.

해시가 동작하는 원리는 아주 간단하고 일상 생활에서 많이 볼 수 있다. 우리가 많이 쓰는 주소록이 바로 해시를 이용한 것이다. 이 주소록은 이름의 첫 글자에 따라 페이지를 구분하고 설명을 쉽게 하기 위해 이 주소록에 기록한 이름은 유일하다고 가정하자.

이 주소록에 이름이 '이해일'이고 연락처가 '1230000'인 정보를 다음과 같은 순서로 기록한다<그림 7>.

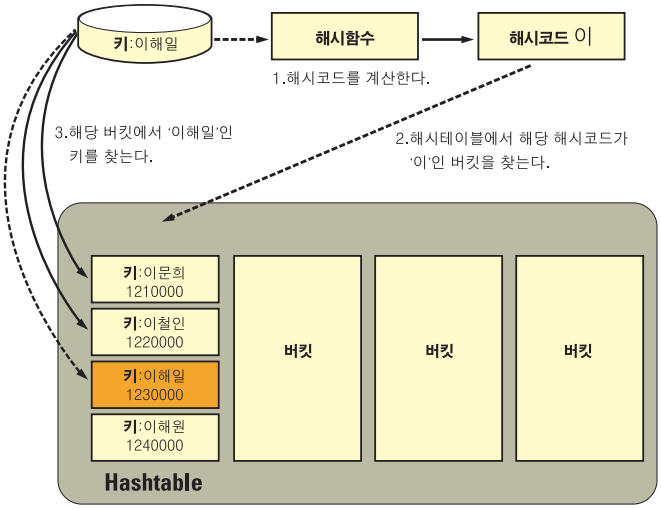
1. 이름의 첫 글자가 무엇인지 알아낸다(첫 글자는 '이'다.)
2. 이 첫 글자에 해당하는 주소록 페이지를 펼친다('이'로 시작하는 이름을 모아 놓은 페이지를 펼친다.)
3. 주소록에 이미 같은 이름이 있는지 확인한다. 같은 이름이 있다면 고쳐 쓰고 같은 이름이 없다면 연락처를 추가한다('이해일'이라는 이름이 없으니까 새로 추가한다.)



<그림 7> 해시에 데이타를 저장하는 과정

이렇게 기록한 주소록에서 원하는 연락처를 찾는 순서는 다음과 같을 것이다<그림 8>.

1. 찾으려는 연락처 이름의 첫 글자를 뽑는다(첫 글자는 '이'다.)
2. 첫 글자에 해당하는 주소록 페이지를 펼친다('이'로 시작하는 이름을 모아 놓은 페이지를 펼친다.)
3. 이 페이지에서 찾으려는 이름을 가진 연락처를 찾는다('이해일'이라는 이름을 찾는다.)



<그림 8> 해시에서 데이타를 꺼내는 과정

주소록이 바로 해시 테이블이고 이름이 연락처의 키이고, 이름의 첫 글자가 해시 코드이다. 이름의 첫 글자를 알아내는 것이 해시 함수(hash function)가 하는 일이다.

Java에서 hashCode 메소드가 바로 해시 함수 역할을 한다. 따라서, 컬렉션 프레임워크의 해시는 키 객체의 hashCode 메소드로 버킷을 결정한다. 키 객체가 동등한데 해시 코드 값이 다르면 어떻게 될까? 동등한 키 객체를 가진 데이타가 다른 버킷에 들어가기 때문에 동일한(identical) 키 객체를 쓰지 않는다면 해시 테이블에 저장한 데이타를 찾아낼 방법이 없다. 따라서, equals 메소드를 재정의했으면 꼭 hashCode 메소드를 재정의해야 한다. 해시의 키로 쓰일 때 문제가 발생할 수 있기 때문이다.

그렇다면, hashCode 메소드는 어떻게 만드는 것이 좋을까? <그림 8>을 보면, 해시는 버킷을 찾을 때는 다루는 데이타의 개수와 상관없이 일정한 성능을 내지만, 한 버킷 안에서 슬롯을 찾을 때는 리스트의 검색과 같은 성능을 낸다. 따라서, 최대한 균등하게 해시 코드를 만들어 내는 것이 좋다. 다음과 같은 hashCode 메소드는 최악이다.

```
public int hashCode() {
    return 31;
}
```

이 hashCode 메소드는 구현 계약의 두 번째 조항은 만족하지만 모든 객체의 해시 코드 값이 같기 때문에 모두 한 버킷에 들어간다. 이 해시 테이블은 이제 해시 테이블이 아닌 O(N)의 성능을 보이는 연결 리스트일 뿐이다! 모든 객체가 서로 다른 해시 코드를 가진다면 성능은 가장 좋겠지만, 현실적으로 이런 해시 코드를 만들어내는 것은 거의 불가능하고 메모리 소모도 많으므로 적절한 지점에서 타협해야 한다. 모든 알고리즘이 그렇듯이 해시도 성능과 메모리 사이에서 적절한 균형을 이뤄야 한다. 간단하면서도 거의 균등한 해시 코드 값을 만드는 hashCode 메소드 구현법(이 방법은 Joshua Bloch의 *[Effective Java Programming Language Guide]*(Addison-Wesley)에 나온 것을 인용했다.)은 다음과 같다.

1. result라는 int 타입 변수에 0이 아닌 상수 값(예를 들면, 17과 같은 값)을 저장한다.
2. 객체의 주요 필드(equals 메소드에서 비교하는 필드)인 모든 f들에 대해 각각 다음 작업을 수행한다.
   a. 각 필드의 해시 코드 c를 다음과 같이 계산한다.
      i. f가 boolean인 경우, (f ? 0 : 1)를 계산한다.
      ii. f가 byte, char, short, int인 경우, (int)f를 계산한다.
      iii. f가 long인 경우, (int)(f ^ (f>>>32))를 계산한다.
      iv. f가 float인 경우, Float.floatToIntBits(f)를 계산한다.
      v. f가 double인 경우, Double.doubleToLongBits(f)를 계산한 후, 2.a.iii와 같이 계산한다.
      vi. f가 객체 참조이고, 이 클래스의 equals 메소드에서 비교하는 대상이라면, f의 hashCode 메소드의 리턴값을 계산한다. 만약, 더 복잡한 비교가 필요다면 f를 표준 형식(canonical form)으로 바꿔서 해시 코드를 계산한다. 만약, f가 null이면 0으로 계산한다(아무 상수나 괜찮지만 0으로 계산하는 것이 관례이다.)
      vii. f가 배열인 경우, 각 구성요소 하나하나를 하나의 필드처럼 처리한다. 위의 규칙을 적용하여 배열의 주요 구성요소의 해시 코드를 모두 계산한 후 2.b 단계에 따라 한 번 더 계산한다.
   b. 2.a에 따라 계산한 해시 코드인 c와 1에서 정의한 result를 다음과 같이 더한다.

```
result = 37*result + c;
```

   c. result를 리턴한다.

equals 메소드에서 비교하지 않는 필드는 해시 코드를 계산할 때 빼야 한다. 만약 이 필드들을 빼지 않으면 hachCode 메소드 구현 계약의 두 번째 조항을 어길 가능성이 커진다. 위 비법에서 쓴 17과 37은 임의의 소수이다. 정확한 이유는 밝혀지지 않았지만, 홀수인 소수를 쓰면 해시 코드 값이 균등해진다고 한다.

이 방법에 따라 전화번호를 표현하는 PhoneNumber 클래스의 hashCode 메소드를 다음과 같이 구현할 수 있다(이 예제는 Joshua Bloch의 *[Effective Java Programming Language Guide]*(Addison-Wesley)에서 나온 것을 인용했다.)

```
public final class PhoneNumber {
    private final short areaCode;
    private final short exchange;
    private final short extension;

    public PhoneNumber(int areaCode, int exchange, int extension) {
        rangeCheck(areaCode, 999, "area code");
        rangeCheck(exchange, 999, "exchange");
        rangeCheck(extension, 9999, "extension");
        this.areaCode = (short) areaCode;
        this.exchange = (short) exchange;
        this.extension = (short) extension;
    }

    private static void rangeCheck(int arg, int max, String name) {
        if (arg < 0 || arg > max)
            throw new IllegalArgumentException(name + ":" + arg);
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
```

```
            return false;
        PhoneNumber pn = (PhoneNumber)o;

        // extention, exchange, areaCode 필드가 equals의비교대상이다.
        // 따라서, 해시코드 값도 이 필드들을 써서 계산해야 한다.
        return pn.extension == extension && pn.exchange == exchange &&
                pn.areaCode == areaCode;
    }

    public int hashCode() {
        int result = 17;
        result = 37*result + areaCode;
        result = 37*result + exchange;
        result = 37*result + extension;
        return result;
    }
}
```

이 방식은 JDK 1.4 배포판에서 실제로 쓰고 있는 방식으로 다음은 java.lang.String의 hashCode 메소드이다.

```
public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++) {
            h = 31*h + val[off++];
        }
        hash = h;
    }
    return h;
}
```

이 메소드를 보면 String 객체를 구성하는 모든 문자가 해시 코드 값 계산에 쓰인다는 것을 알 수 있다. JDK 1.2에서는 성능을 높이려고 처음 16개 문자로만 해시 코드 값을 계산했는 데, URL과 같이 계층 구조를 가지는 문자열들을 다룰 때 심각한 문제가 있었다. 예를 들어, http://java.sun.com/j2se, http://java.sun.com/j2ee과 같이 http://java.sun.com으로 시작하는 모든 URL은 처음 16자가 같기 때문에 같은 해시 코드 값을 가진다. String 객체는 불변 객체이고 해시 코드 계산 비용이 많기 때문에 hashCode 메소드가 처음 호출될 때 해시 코드 값을 계산하는 늦은 초기화 기법을 쓴 것도 눈여겨 보기 바란다.

앞에 제시한 해시 코드 구현 방식은 간단하긴 하지만 성가신 작업이고 개선의 여지도 있다. 앞으로 나올 JDK 배포판에서는 모든 객체의 해시 코드 값을 구해주는 유틸리티 메소드가 제공되길 기대해 보자.

객체의 순서를 어떻게 정할까?

해야 할 일은 많은데 자원은 제한되어 있기 때문에 해야 할 일의 우선 순위를 결정해야 할 경우가 많다. 이처럼 순서를 결정하는 일은 실제 세상에서 자주 수행하는 작업이다. 따라서, 순서를 다루는 코드를 만드는 일도 굉장히 많다.

순서란 무엇일까? 순서란 어떤 기준에 따라 정한 선후 관계이다. 따라서, 순서에는 반드시 기준이라는 것이 따라다닌다. 순서를 정할 때 쓰는 기준에는 두 종류가 있다. 자연수는 1, 2, 3처럼 크기를 기준으로, 날짜는 2003년 8월 3일, 2003년 8월 4일, 2003년 8월 5일처럼 시간을 기준으로, 문자열은 Java, 이해일, 홍길동처럼 사전식 배열을 기준으로 순서를 결정하는 것이 자연스럽다. 이렇게 결정한 순서를 '자연스러운 순서(natural ordering)'라고 한다.

하지만, 상식을 벗어나는 순서가 필요하거나 순서를 생각하기 힘든 추상 개념의 순서를 정해야 한다면 사용자 맞춤 순서를 써야 한다. 예를 들어, 문자열을 사전 순서가 아닌 문자열 길이를 기준으로 순서를 정할 수도 있다. 이런 순서가 바로 '맞춤 순서(custom ordering)'이다.

Java에서는 Comparable과 Comparator라는 인터페이스로 객체 순서 결정을 지원한다. 이 두 인터페이스를 쓰면 Arrays.binarySearch, Arrays.sort, Collections.binarySearch, Collections.max, Collections.min, Collections.sort와 같은 순서를 다루는 유틸리티 메소드와 TreeMap, TreeSet과 같이 자동 정렬 맵이나 집합을 다루는 컬렉션을 마음대로 쓸 수 있다.

Comparable 인터페이스

Comparable 인터페이스는 객체들 사이의 자연스러운 순서를 결정할 때 쓴다. 이 인터페이스에는 compareTo라는 메소드 하나밖에 없다. 이 메소드의 시그니처는 다음과 같이 아주 단순하다.

```
public int compareTo(Object o)
```

이 메소드는 this 객체와 인자로 받은 o를 비교하여 this가 o보다 순서가 앞서면 양수를, 순서가 같으면 0을, 순서가 뒤지면 음수를 리턴한다. 이렇게 간단한 작업만 해주면 Arrays, Collections, TreeMap, TreeSet이 제공하는 유용한 기능을 모두 쓸 수 있다니 정말 즐거운 일이 아닌가? 하지만, 이 때도 몇 가지 구현 계약을 지켜야 한다. compareTo의 구현 계약은 다음과 같다 *(java.lang.Comparable의 compareTo 메소드 명세).*

> this 객체와 인자로 받은 객체 사이의 순서를 비교한다. this 객체가 인자로 받은 객체보다 크면 양의 정수를, 같으면 0을, 작으면 음의 정수를 리턴한다. 만약, 인자로 받는 객체의 타입이 this 객체와 비교할 수 없는 것이라면, ClassCastException을 던진다.
>
> 다음 설명에서 sgn(expression)이라는 표현은 수학의 signum 함수와 같은 것이다. 이 함수는 표현식(expression)의 값이 음수면 -1, 영이면 0, 양수면 1을 리턴한다.

1. 모든 x, y에 대해 sgn(x.compareTo(y)) == -sgn(y.compareTo(x))임을 보장해야 한다(이 식은 y.compareTo(x)가 예외를 던질 때만 x.compareTo(y)도 같은 예외를 던져야 한다는 뜻이다.)
2. 추이성 관계를 보장해야 한다. (x.compareTo(y) > 0 && y.compareTo(z) > 0)라는 것은 x.compareTo(z) > 0라는 뜻이다.
3. x.compareTo(y) == 0이면, 모든 z에 대해 sgn(x.compareTo(z)) == sgn(y.compareTo(z))라는 뜻이다.
4. 엄격한 요구사항은 아니지만, (x.compareTo(y) == 0) == (x.equals(y))인 것이 좋다. 만약, 어떤 클래스가 Comparable 인터페이스를 구현하면서 이 조항을 지키지 않는다면 반드시 다음과 같이 이 사실을 알려 주는 것이 좋다. '주의 : 이 클래스의 자연스러운 순서는 equals 메소드와 맞지 않는다.'

compareTo 메소드의 구현 계약은 equals 메소드와 비슷하지만 compareTo 메소드는 다른 클래스의 객체를 비교할 수 없다. 다른 클래스의 객체를 비교하려고 하면 ClassCastException이 발생한다. 언어 차원에서 강제로 서로 다른 클래스의 객체를 비교하지 못하게 할 수는 없지만, JDK 1.4 배포판부터는 Java 플랫폼 라이브러리의 모든 클래스는 다른 클래스의 객체를 비교하는 compareTo 메소드를 제공하지 않는다.

어떤 클래스가 hashCode 메소드의 구현 계약을 어겼을 때 해시 알고리즘을 쓰는 컬렉션이 제대로 동작하지 않을 수 있는 것처럼, compareTo 메소드의 구현 계약을 어긴다면 객체의 순서를 비교하는 TreeSet, TreeMap과 같은 자동 정렬 컬렉션이나 Collections, Arrays와 같이 정렬과 검색을 처리하는 유틸리티 메소드를 제공하는 클래스들이 제대로 동작하지 않을 것이다.

1번, 2번, 3번 조항은 당연한 것이기 때문에 자세히 살펴보지는 않겠지만 compareTo 메소드는 동등성 검사를 내포하고 있으므로 equals 메소드의 구현 계약이 따라야 하는 계약조항(반사성, 대칭성, 추이성, null과 다름)과 제약을 그대로 따라야 한다는 것을 기억해야 한다. equals 메소드와 같이 구체 클래스를 상속받아 새로운 필드를 추가하면서 구현 계약을 지키는 compareTo 메소드를 만드는 것은 불가능하다. 이 문제는 equals 메소드와 똑 같은 방식으로 피해야 한다. 상속 대신에 이 클래스의 인스턴스에 대한 참조를 멤버 필드로 가지는 클래스를 만들고 이 필드를 리턴하는 뷰 메소드를 제공하는 것이 좋다.

4번 조항은 필수 조항이라기보다는 강력한 권고사항이다. compareTo 메소드와 equals 메소드의 동등성 검사 결과는 같은 것이 좋은데, 이것을 어기면 컬렉션을 쓸 때 문제가 생길 수 있다. 보통 컬렉션은 equals 메소드를 써서 구성요소의 동등성을 비교하지만, 자동정렬 컬렉션은 compareTo 메소드를 써서 구성요소의 동등성을 비교하기 때문에 일관성 없는 결과가 나올 수 있다. BigDecimal 클래스의 compareTo 메소드와 equals 메소드의 결과는 일치하지 않는다. HashSet 인스턴스를 하나 생성하여 new BigDecimal("1.0")과 new BigDecimal("1.00")을 넣으면 이 집합은 두 개의 원소를 가진다. BigDecimal 클래스의 equals 메소드는 new BigDecimal("1.0")과 new BigDecimal("1.00")을 '동등하다'고 판단하지 않기 때문이다. 하지만, HashSet 대신에 자동정렬 컬렉션인 TreeSet에 new BigDecimal("1.0")과 new BigDecimal("1.00")을 넣으면 단 하나의 원소만 가진다. TreeSet은 compareTo 메소드를 쓰고 BigDecimal의 compareTo 메소드는 두 BigDecimal 인스턴스를 '동등하다'고 판단하기 때문이다.

전화번호를 표현하는 PhoneNumber 클래스가 자연스러운 순서를 지원하게 만들어 보자(이 . 예제는 Joshua Bloch의 *[Effective Java Programming Language Guide]*(Addison-Wesley)에 나온 것을 인용했다.)

```
public final class PhoneNumber implements Comparable {
    private final short areaCode;
    private final short exchange;
    private final short extension;

    public PhoneNumber(int areaCode, int exchange, int extension) {
        rangeCheck(areaCode, 999, "area code");
        rangeCheck(exchange, 999, "exchange");
        rangeCheck(extension, 9999, "extension");
        this.areaCode = (short) areaCode;
        this.exchange = (short) exchange;
        this.extension = (short) extension;
    }
```

```
    private static void rangeCheck(int arg, int max, String name) {
        if (arg < 0 || arg > max)
            throw new IllegalArgumentException(name + ":" + arg);
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;

        return pn.extension == extension && pn.exchange == exchange &&
                pn.areaCode == areaCode;
    }

    public int hashCode() {
        int result = 17;
        result = 37*result + areaCode;
        result = 37*result + exchange;
        result = 37*result + extension;
        return result;
    }

    public int compareTo(Object o) {
    PhoneNumber pn = (PhoneNumber)o;

    // 전화번호는 지역번호-국번-내선번호 순서로 구성된다. 따라서 이 순서
대로 비교해야 한다.
    // 지역번호를 비교한다.
    if (areaCode < pn.areaCode) return -1;
    if (areaCode > pn.areaCode) return 1;

    // 지역번호가 동일하면 교환번호를 비교한다.
    if (exchange < pn.exchange) return -1;
    if (exchange > pn.exchange) return 1;

    // 지역번호와 교환번호가 동일한 경우에는 내선번호를 비교한다.
    if (extension < pn.extension) return -1;
    if (extension > pn.extension) return 1;

        // 모든 필드는 동일하다.
        return 0;
    }
}
```

여기서 compareTo 메소드가 리턴값으로 양수, 0, 음수만 정의했지 수의 크기를 정의하지 않았다는 점과 rangeCheck 메소드로 전화번호를 구성하는 번호가 절대 음수가 될 수 없다는 불변조건이 보장되므로, 다음과 같이 좀 더 빠르고 간단하게 compareTo 메소드를 구현할 수 있다.

```
public int compareTo(Object o) {
    PhoneNumber pn = (PhoneNumber)o;

    //지역번호를 비교한다.
    int areaCodeDiff = areaCode - pn.areaCode;
    if (areaCodeDiff != 0) return areaCodeDiff;

    //지역번호가 동일하면 국번을 비교한다.
    int exchangeDiff = exchange - pn.exchange;
    if (exchangeDiff != 0) return exchangeDiff;

    //마지막으로 내선번호를 비교한다.
    return extension - pn.extension;
}
```

이 방법은 비교하는 필드 값이 음수가 아니라는 불변조건이 보장될 때만 쓸 수 있다는 것을 다시 한 번 강조한다. 만약, 필드 값이 음수가 된다면 찾아내기 아주 어려운 버그로 남을 것이다.

Comparator 인터페이스

Comparator 인터페이스는 맞춤 순서를 정하고 싶을 때 쓴다. 이 인터페이스에는 compare라는 메소드 하나밖에 없다.

```
public int compare(Object o1, Object o2)
```

이 메소드는 서로 비교해야 할 객체 두 개(o1, o2)를 인자로 받아서 o1이 o2보다 순서가 앞서면 양수를, 같으면 0을, 뒤지면 음수를 리턴한다.

Comparator는 GoF의 전략 패턴에 나오는 전략 클래스로 정렬 순서를 결정하는 알고리즘을 제공하는 역할을 한다. 예를 들어, Collections.sort 메소드 중에서 Comparator를 쓰는 것의 시그니처를 살펴보자.

```
public static void sort(List l, Comparator c);
```

이 메소드는 정렬에 필요한 비교 알고리즘을 Comparator 타입의 인자

로 받는다. 예를 들어, 문자열 길이 순서대로 정렬할 필요가 있다고 하면 다음과 같이 Comparator 객체를 생성해서 제공하면 된다(이 예제는 Joshua Bloch의 *[Effective Java Programming Language Guide]*(Addison-Wesley)에 나온 것을 인용했다.)

```
public class StringLengthComparator implements Comparator {
    private StringLengthComparator() { }

    public static final StringLengthComparator
        INSTANCE = new StringLengthComparator();

    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.length() - s2.length();
    }
}

// 사용법. 문자열 리스트l을 문자열 길이 순서대로 정렬한다.
Collections.sort(l, StringLengthComparator.INSTANCE);
```

하지만, 보통 Comparator는 익명 클래스이거나 중첩 클래스로 구현한다.

```
// 익명 클래스로 구현하기(길이를 기준으로 순서를 정한다.)
Arrays.sort(stringArray, new Comparator() {
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.length() - s2.length();
    }
});

// 중첩 클래스로 구현하기(대소문자를 구분하지 않고 순서를 정한다.)
// java.lang.String
public class String implements Comparable, ... {
    ...
    public static final Comparator CASE_INSENSITIVE_ORDER
                    = new CaseInsensitiveComparator();
    private static class CaseInsensitiveComparator
            implements Comparator, java.io.Serializable {
        ...
    public int compare(Object o1, Object o2) {
        String s1 = (String) o1;
        String s2 = (String) o2;
        int n1=s1.length(), n2=s2.length();
        for (int i1=0, i2=0; i1<n1 && i2<n2; i1++, i2++) {
          char c1 = s1.charAt(i1);
          char c2 = s2.charAt(i2);
          if (c1 != c2) {
            c1 = Character.toUpperCase(c1);
            c2 = Character.toUpperCase(c2);
            if (c1 != c2) {
              c1 = Character.toLowerCase(c1);
              c2 = Character.toLowerCase(c2);
              if (c1 != c2) return c1 - c2;
            }
          }
        }
        return n1 - n2;
      }
    }
    ...
}
```

Comparator는 인자로 받은 두 객체를 비교하고 Comparable이 this 객체와 인자로 받은 객체를 비교한다는 점만 빼면 Comparator는 Comparable과 같다. 따라서, Comparator의 구현 계약은 Comparable의 구현 계약과 같다.

Comparable이나 Comparator를 쓰면 객체에 원하는 순서를 줄 수 있다. 자연스러운 순서가 필요하면 Comparable을 쓰고 맞춤 순서가 필요하면 Comparator를 쓰면 된다.

참고문헌

- Object-Oriented Analysis and Design, Grady Booch, Addison-Wesley
- Design Patterns, Erich Gamma et al., Addison-Wesley
- Effective Java Programming Language Guide, Joshua Bloch, Addison-Wesley
- Practical Java Programming Language Guide, Peter Haggar, Addison-Wesley
- Java Pitfalls, Michael C. Daconta et al., Wiley
- Java Rules, Duglas Dunn, Addison-Wesley
- Java Tutorial, http://java.sun.com/docs/books/tutorial/
- JDK 1.4.2 SDK Document, http://java.sun.com/j2se/1.4.2/docs/api/